Oracle Berkeley DB – Java Edition

Requisiti.

Una minima conoscenza del linguaggio di programmazione Java. IDE a piacere. Meglio senza. Un Sistema Operativo per cui esista una piattaforma Java 5 o superiore. Serve la base dati in argomento. Si scarica da:

http://www.oracle.com/technology/products/berkeley-db/je/index.html

Ai fini del tutorial interessa unicamente il file “je3.2.13.jar”, da usare come libreria per la compilazione dei sorgenti qui proposti.

Il record.

Vogliamo conservare degli oggetti che hanno due proprietà: un nome e una data. Il nome è una stringa di lunghezza imprecisata. La data è un oggetto java.util.Date. Usiamo il Direct Persistence Layer di Oracle Berkeley DB. Il DPL altro non è che un modo per salvare, recuperare, eliminare o aggiornare dei dati aggregati lavorando direttamente con l'oggetto Java che li contiene. Una versione preliminare del nostro Record potrebbe essere questa:

```
package jetutorial;

import java.util.Date;

public class Record {
   private Date time;
   private String name;
   private Long id;

   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }

   public Date getTime() { return time; }
   public void setTime(Date time) { this.time = time; }

   public String getName() { return name; }
   public void setName(String name) { this.name = name; }
}
```

Per salvare questo record nella base dati Berkeley DB dobbiamo apporre l'annotazione Entity alla classe e indicare quale tra le proprietà possedute sia la chiave primaria con l'annotazione PrimaryKey. La nostra chiave primaria è id.

```
package jetutorial;

import com.sleepycat.persist.model.PrimaryKey;
import com.sleepycat.persist.model.Entity;
import java.util.Date;

@Entity
public class Record {
   private Date time;
   private String name;

   @PrimaryKey(sequence="RecordId")
   private Long id;
```

```
   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }

   public Date getTime() { return time; }
   public void setTime(Date time) { this.time = time; }

   public String getName() { return name; }
   public void setName(String name) { this.name = name; }
}
```

L'attributo sequence di PrimaryKey stabilisce che la chiave primaria dei nostri record sarà generata automaticamente dalla base dati. Il valore “RecordId” assegnato all'attributo in parola è semplicemente un nome che identifica il generatore di sequenza creato dalla base dati per i Record. Potrebbe essere benissimo “Pippo” o “Pluto”, basta che ce ne sia uno.

Il DAO.

DAO è l'acronimo di Data Access Object. Un DAO contiene la dichiarazione dei metodi usati per l'interazione con la base dati. Un esempio di DAO per la gestione dei nostri Record potrebbe essere:

```
package jetutorial;

import java.util.Collection;

public interface Dao {

   void open();

   void close();

   void store(Record record);

   void update(Record record);

   void delete(Record record);

   Collection<? extends Record> getRecords();
}
```

Poiché la comunicazione con una base dati è comunemente soggetta a precondizioni esterne al codice del programma dichiariamo il rilascio di eccezioni ad hoc da parte dei metodi del nostro Dao. Definiamo prima il tipo di eccezione generata:

```
package jetutorial;

public class DaoException extends Exception {

   public DaoException() {}
   public DaoException(Throwable cause) { super(cause); }
   public DaoException(String message) { super(message); }

   public DaoException(String message, Throwable cause) {
      super(message, cause);
   }
}
```

per poi usarla nel DAO.

```
package jetutorial;

import java.util.Collection;

public interface Dao {

   void open() throws DaoException;

   void close() throws DaoException;

   void store(Record record) throws DaoException;

   void update(Record record) throws DaoException;

   void delete(Record record) throws DaoException;

   Collection<? extends Record> getRecords() throws DaoException;
}
```

Il DAO astrae tutto un insieme di dettagli che sono irrilevanti per quelle parti del programma che vogliano semplicemente leggere o scrivere Record. Un'utile estensione del concetto di DAO prevede che lo stesso sia considerato un produttore di eventi così che più componenti possano essere messi nelle condizioni di sapere quando un Record venga aggiornato, salvato o rimosso. L'evento prodotto dal Dao può essere fatto come segue.

```
package jetutorial;

import java.util.EventObject;

public class DaoEvent extends EventObject {
   public static enum Type {
      UPDATE,
      STORE,
      DELETE
   };

   private DaoEvent.Type type;
   private Record record;

   public DaoEvent(DaoEvent.Type type, Dao source, Record record) {
      super(source);
      this.type = type;
      this.record = record;
   }

   public Dao getSource() {
      return (Dao)super.getSource();
   }

   public Record getRecord() {
      return record;
   }

   public DaoEvent.Type getType() {
      return type;
   }

}
```

L'ascoltatore di eventi DaoEvent è la classica interfaccia figlia di EventListener.

```
package jetutorial;

import java.util.EventListener;

public interface DaoEventListener extends EventListener {

   void daoEventPerformed(DaoEvent e);
}
```

Il Dao diventa un produttore di eventi dichiarando il possesso di un metodo per la registrazione di un ascoltatore ed un metodo per la rimozione di un ascoltatore registrato.

```
package jetutorial;

import java.util.Collection;

public interface Dao {

   void open() throws DaoException;

   void close() throws DaoException;

   void store(Record record) throws DaoException;

   void update(Record record) throws DaoException;

   void delete(Record record) throws DaoException;

   Collection<? extends Record> getRecords() throws DaoException;

   void addDaoEventListener(DaoEventListener listener);

   void removeDaoEventListener(DaoEventListener listener);
}
```

Il Dao per il database Oracle Berkeley DB.

Forniamo a questo punto una versione concreta di Dao che usi il database oggetto del tutorial: chiamiamola JEDao. Il database richiede una cartella di base in cui saranno scritti i file necessari al suo funzionamento. Per questa ragione dichiariamo un costruttore che accetta come argomento un file.

```
package jetutorial;

import java.io.File;

/** Incompleto */
public class JEDao implements Dao {
   private File databaseDirectory;

   public JEDao(File databaseDirectory) {
      this.databaseDirectory = databaseDirectory;
   }
}
```

Nell'interfaccia Dao il metodo deputato all'apertura di una connessione con la base dati è open(). Se

la base dati non è già stata creata, la creiamo in questo stesso metodo.

```
package jetutorial;

import java.io.File;
import com.sleepycat.je.DatabaseException;

/** Incompleto */
public class JEDao implements Dao {
  private File databaseDirectory;

  public JEDao(File databaseDirectory) {
    this.databaseDirectory = databaseDirectory;
  }

  /** Apre la base dati. Se la base dati non è già stata creata, la
  crea. */
  public void open() throws DaoException {
    try {
      jeOpen();
    } catch(DatabaseException ex) {
      throw new DaoException(ex);
    }
  }

  private void jeOpen() throws DatabaseException {

  }
}
```

Il metodo open() rinvia a jeOpen() per un vezzo estetico. Incapsulando l'eccezione evitiamo che si propaghi nel programma la conoscenza di un tipo di eccezione, DatabaseException, specifica della concreta base dati usata.

Creare e connettersi alla base dati.

La creazione o la "connessione" ad Oracle Berkeley DB richiede un oggetto Environment. La creazione di un oggetto Environment richiede un oggetto EnvironmentConfig ed un File. Otterremo il File da chi costruirà un JEDao mentre generiamo nel metodo open() la configurazione.

```
package jetutorial;

import java.io.File;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;

/** Incompleto */
public class JEDao implements Dao {
  private Environment databaseEnvironment;
  private File databaseDirectory;

  public JEDao(File databaseDirectory) {
    this.databaseDirectory = databaseDirectory;
  }

  /** Apre la base dati. Se la base dati non è già stata creata, la
  crea. */
  public void open() throws DaoException {
```

```
      try {
        jeOpen();
      } catch(DatabaseException ex) {
        throw new DaoException(ex);
      }
    }

    private void jeOpen() throws DatabaseException {
      EnvironmentConfig environmentConfig = new EnvironmentConfig();
      environmentConfig.setAllowCreate(true);
      environmentConfig.setTransactional(true);

      databaseEnvironment = new Environment(
        databaseDirectory,
        environmentConfig);
    }
}
```

Dichiariamo un campo di tipo Environment perchè il riferimento creato in apertura della base dati dovrà poi essere usato per chiudere la stessa base dati nel metodo close(). La configurazione che usiamo per il nostro database esprime attraverso il metodo setAllowCreate la volontà di creare la base dati nel caso in cui non esista e attraverso il metodo setTransactional l'intento di lasciare alla base dati il compito di gestire la consistenza delle operazioni di lettura e scrittura.

Dopo aver creato il nostro DatabaseEnvironment, passiamo alla creazione di un EntityStore. L'EntityStore è la base del Direct Persistence Layer di Oracle Berkeley DB. Come per l'ambiente, anche l'EntityStore richiede una configurazione iniziale da definirsi con un oggetto StoreConfig.

```
package jetutorial;

import java.io.File;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.persist.EntityStore;
import com.sleepycat.persist.StoreConfig;

/** Incompleto */
public class JEDao implements Dao {
  private Environment databaseEnvironment;
  private EntityStore entityStore;
  private File databaseDirectory;

  public JEDao(File databaseDirectory) {
    this.databaseDirectory = databaseDirectory;
  }

  /** Apre la base dati. Se la base dati non è già stata creata, la
  crea. */
  public void open() throws DaoException {
    try {
      jeOpen();
    } catch(DatabaseException ex) {
      throw new DaoException(ex);
    }
  }

  private void jeOpen() throws DatabaseException {
    EnvironmentConfig environmentConfig = new EnvironmentConfig();
    environmentConfig.setAllowCreate(true);
    environmentConfig.setTransactional(true);
```

```
      databaseEnvironment = new Environment(
        databaseDirectory,
        environmentConfig);

      StoreConfig storeConfig = new StoreConfig();
      storeConfig.setAllowCreate(true);
      storeConfig.setTransactional(true);

      entityStore = new EntityStore(
        databaseEnvironment,
        "jetutorial",
        storeConfig);
    }
}
```

La comunicazione con l'EntityStore avviene attraverso un oggetto di tipo PrimaryIndex. Questo PrimaryIndex è generato dall'EntityStore e corrisponde vagamente all'idea di una tabella contenente tutti i Record. L'oggetto PrimaryIndex è parametrico rispetto alla chiave primaria del tipo di dato che gestisce. Poiché la comunicazione con la base dati avviene attraverso questo PrimaryIndex, lo dichiariamo come campo.

```
package jetutorial;

import java.io.File;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.persist.EntityStore;
import com.sleepycat.persist.PrimaryIndex;
import com.sleepycat.persist.StoreConfig;

/** Incompleto */
public class JEDao implements Dao {
  private Environment databaseEnvironment;
  private EntityStore entityStore;
  private File databaseDirectory;
  private PrimaryIndex<Long, Record> records;

  public JEDao(File databaseDirectory) {
    this.databaseDirectory = databaseDirectory;
  }

  /** Apre la base dati. Se la base dati non è già stata creata, la
  crea. */
  public void open() throws DaoException {
    try {
      jeOpen();
    } catch(DatabaseException ex) {
      throw new DaoException(ex);
    }
  }

  private void jeOpen() throws DatabaseException {
    EnvironmentConfig environmentConfig = new EnvironmentConfig();
    environmentConfig.setAllowCreate(true);
    environmentConfig.setTransactional(true);

    databaseEnvironment = new Environment(
      databaseDirectory,
      environmentConfig);
```

```
      StoreConfig storeConfig = new StoreConfig();
      storeConfig.setAllowCreate(true);
      storeConfig.setTransactional(true);

      entityStore = new EntityStore(
         databaseEnvironment,
         "jetutorial",
         storeConfig);

      records = entityStore.getPrimaryIndex(Long.class, Record.class);
   }
}
```

Questo completa l'apertura della base dati.

Chiusura della base dati.

Chiudiamo la base dati nel metodo close() del nostro JEDao. La chiusura è un'operazione molto breve e consiste nella chiusura dell'EntityStore seguita dalla chiusura dell'Environment. Il metodo close è il seguente.

```
   /** Chiude la base dati. */
   public void close() throws DaoException {
      try {
         jeClose();
      } catch(DatabaseException ex) {
         throw new DaoException(ex);
      }
   }

   private void jeClose() throws DatabaseException {
      try {
         entityStore.close();
      } finally {
         databaseEnvironment.cleanLog();
         databaseEnvironment.close();
      }
   }
```

Il sistema di notifica.

Prima di procedere con le operazioni di manipolazione della base dati approntiamo il meccanismo di notifica degli eventi. Il sistema di notifica richiede semplicemente un elenco di oggetti DaoEventListener ed un metodo per inviare ad ognuno di essi un evento DaoEvent. Per l'elenco di DaoEventListener usiamo un oggetto CopyOnWriteArrayList. La lista CopyOnWriteArrayList ci permette di aggiungere o rimuovere DaoEventListener mentre è in corso la notifica di eventi ciò che risulta utile sia in contesti concorrenti che in ambiti a thread singolo.

```
package jetutorial;

import java.io.File;
import java.util.concurrent.CopyOnWriteArrayList;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.persist.EntityStore;
```

```
import com.sleepycat.persist.PrimaryIndex;
import com.sleepycat.persist.StoreConfig;

/** Incompleto */
public class JEDao implements Dao {
  private CopyOnWriteArrayList<DaoEventListener> listeners =
    new CopyOnWriteArrayList<DaoEventListener>();
  private Environment databaseEnvironment;
  private EntityStore entityStore;
  private File databaseDirectory;
  private PrimaryIndex<Long, Record> records;

  public JEDao(File databaseDirectory) {
    this.databaseDirectory = databaseDirectory;
  }

  /** Apre la base dati. Se la base dati non è già stata creata, la
  crea. */
  public void open() throws DaoException {
    try {
      jeOpen();
    } catch(DatabaseException ex) {
      throw new DaoException(ex);
    }
  }

  private void jeOpen() throws DatabaseException {
    EnvironmentConfig environmentConfig = new EnvironmentConfig();
    environmentConfig.setAllowCreate(true);
    environmentConfig.setTransactional(true);

    databaseEnvironment = new Environment(
      databaseDirectory,
      environmentConfig);

    StoreConfig storeConfig = new StoreConfig();
    storeConfig.setAllowCreate(true);
    storeConfig.setTransactional(true);

    entityStore = new EntityStore(
      databaseEnvironment,
      "jetutorial",
      storeConfig);

    records = entityStore.getPrimaryIndex(Long.class, Record.class);
  }

  /** Chiude la base dati. */
  public void close() throws DaoException {
    try {
      jeClose();
    } catch(DatabaseException ex) {
      throw new DaoException(ex);
    }
  }

  private void jeClose() throws DatabaseException {
    try {
      entityStore.close();
    } finally {
      databaseEnvironment.cleanLog();
      databaseEnvironment.close();
    }
  }
```

```
  public void addDaoEventListener(DaoEventListener listener) {
    listeners.add(listener);
  }

  public void removeDaoEventListener(DaoEventListener listener) {
    listeners.remove(listener);
  }

  private void fireDaoEvent(DaoEvent e) {
    for(DaoEventListener listener : listeners) {
      listener.daoEventPerformed(e);
    }
  }
}
```

Ora quando JEDao riceverà una richiesta di creazione, eliminazione o aggiornamento di un Record creerà un DaoEvent che segnali l'operazioni richiesta e la notificherà tramite il suo metodo fireDaoEvent ad ogni DaoEventListener registrato. In questo modo oggetti diversi da quello che avrà richiesto l'operazione potranno accorgersi di ciò che è stato fatto.

Salvataggio di un nuovo Record.

Per salvare un Record nella base dati si usa uno tra i metodi put, putNoOverwrite o putNoReturn dell'oggetto PrimaryIndex. Se ricordate, noi abbiamo stabilito che la base dati generi automaticamente le chiavi primarie dei nuovi Record. Diciamo ora che la base dati genera una chiave primaria se il Record passato ad uno dei metodi put abbia un valore null o zero come chiave primaria. In caso di valori diversi la base dati andrà ad aggiornare il Record avente quella chiave. Il metodo store(Record) è il seguente.

```
  public void store(Record record) throws DaoException {
    try {
      record.setId(null);
      records.putNoReturn(record);
      DaoEvent storeEvent = new DaoEvent(
        DaoEvent.Type.STORE,
        this,
        record);
      fireDaoEvent(storeEvent);
    } catch(DatabaseException ex) {
      throw new DaoException(ex);
    }
  }
```

Nel metodo jeStore l'invocazione setId(null) è ciò che ci consente di far vedere alla base dati il Record come nuovo, a prescindere da un eventuale valore del suo campo id. Senza quel null lo store diventerebbe un possibile update, nel caso in cui la chiave fosse già presente. Non gestiamo la presenza di un valore id diverso da null come eccezione perchè il fatto che tale Record sia considerato sempre diverso da qualsiasi altro Record già immagazzinato è espresso dalla firma del metodo store e dal contratto di Dao che prevede un metodo update per il caso di aggiornamento.

Aggiornamento di un Record esistente.

L'aggiornamento è praticamente uguale all'inserimento di un nuovo Record. C'è un problema

“semantico”: come gestire la richiesta di aggiornare un Record che non esiste nella base dati? Scegliamo di trattare come questo fatto come errore perchè esso risulta contraddittorio rispetto al significato espresso nella firma del metodo e nel contratto di Dao. Il tentativo potrebbe inoltre essere il segnale di un problema di consistenza tra il programma ed il sottosistema di persistenza: qualcuno dispone di un Record che non dovrebbe più avere.

```
public void update(Record record) throws DaoException {
  try {
    if(records.contains(record.getId()) == false) {
      throw new DaoException("Cannot find a record to update");
    }
    records.putNoReturn(record);
    DaoEvent updateEvent = new DaoEvent(
      DaoEvent.Type.UPDATE,
      this,
      record);
    fireDaoEvent(updateEvent);
  } catch(DatabaseException ex) {
    throw new DaoException(ex);
  }
}
```

Eliminazione di un Record.

L'eliminazione di un Record richiede l'invocazione del metodo delete del PrimaryIndex. Ancora una volta il problema maggiore riguarda l'interpretazione di una richiesta di rimozione di un Record inesistente. Trattiamo il caso come eccezione per le ragioni già citate.

```
public void delete(Record record) throws DaoException {
  try {
    if(records.delete(record.getId()) == false) {
      throw new DaoException("No record to remove.");
    }
    DaoEvent deleteEvent = new DaoEvent(
      DaoEvent.Type.DELETE,
      this,
      record);
    fireDaoEvent(deleteEvent);
  } catch(DatabaseException ex) {
    throw new DaoException(ex);
  }
}
```

Caricamento dei Record.

Il caricamento dei Record, pur brevissimo, è un'operazione delicata. Propongo la restituzione di un insieme di Record per semplicità ma non è la strada che preferirei in un contesto reale perchè può consumare una quantità rilevante di memoria ed è raramente necessario. Presenterò in seguito un'alternativa. La scansione dei Record è realizzata usando un oggetto EntityCursor, ricavato dal PrimaryIndex. Essendo l'uso di EntityCursor passibile di rilasciare un'eccezione DatabaseException e nell'ottica di non voler comunicare all'esterno del Dao i dettagli relativi al funzionamento della base dati è necessario caricare i Record estratti dal database in una collezione a parte. Detto questo, il meccanismo è abbastanza sintetico.

```
  public Collection<? extends Record> getRecords() throws DaoException {
    try {
      ArrayList<Record> result = new ArrayList<Record>();
      EntityCursor<Record> cursor = records.entities();
      try {
        jeFill(result, cursor);
      } finally {
        cursor.close();
      }
      return result;
    } catch(DatabaseException ex) {
      throw new DaoException(ex);
    }
  }

  private void jeFill(ArrayList<Record> list, EntityCursor<Record> cursor)
    throws DatabaseException
  {
    for(Record r : cursor) {
      list.add(r);
    }
  }
```

Caricamento dei Record: alternativa.

Un'alternativa alla restituzione della lista di Record. Prima definiamo un'operazione applicabile ad un Record.

```
package jetutorial;

public interface RecordOperation {

  void initialize();

  void apply(Record record);

  void cleanup();
}
```

Poi dotiamo Dao della capacità di applicare questa operazione ad ogni Record con un metodo di questo tipo:

```
/* In Dao.java */
  void foreachRecord(RecordOperation op);
```

JEDao applicherebbe questa operazione in un modo simile a questo:

```
/* In JEDao.java */
  public void foreachRecord(RecordOperation op) throws DaoException {
    op.initialize();
    try {
      jeForeachRecord(op);
    } catch(DatabaseException ex) {
      throw new DaoException(ex);
    } finally {
      op.cleanup();
    }
  }
```

```
  private void jeForeachRecord(RecordOperation op) throws DatabaseException {
    EntityCursor<Record> cursor = records.entities();
    try {
      for(Record r : cursor) {
        op.apply(r);
      }
    } finally {
      cursor.close();
    }
  }
```

La differenza è che con RecordOperation possiamo compiere una scansione dell'intero insieme di Record contenuti nella base dati senza doverli caricare tutti quanti in memoria.

Come provare il sistema.

La classe che segue contiene un programma che usa le classi qui presentate per immagazzinare dei valori immessi dall'utente. Il codice è debitamente commentato. Il programma richiede che l'utente abbia a disposizione una cartella, preferibilmente vuota, da dedicare alla creazione della base dati. All'avvio, il programma chiede all'utente di specificare il percorso di quella cartella dopodichè propone un'interfaccia grafica minimale per inserire, modificare o eliminare degli oggetti Record.

```
package jetutorial;

import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;

public class Main {

  public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() { public void run() {
      new Main().start();
    }});
  }

  /* UN callback */
  private interface DaoEventProcessor { void process(DaoEvent e); };

  /* Callback invocato quando si verifichi un evento DaoEvent.Type.UPDATE */
  private DaoEventProcessor update = new DaoEventProcessor() {
    public void process(DaoEvent e) { recordUpdate(e); }
  };

  /* Callback invocato quando si verifichi un evento DaoEvent.Type.STORE */
  private DaoEventProcessor store = new DaoEventProcessor() {
    public void process(DaoEvent e) { recordStore(e); }
  };

  /* Callback invocato quando si verifichi un evento DaoEvent.Type.DELETE */
  private DaoEventProcessor delete = new DaoEventProcessor() {
    public void process(DaoEvent e) { recordDelete(e); }
  };

  /* Mappa evento-callback. Usata dall'ascolatore di eventi associato al Dao */
```

```
private Map<DaoEvent.Type, DaoEventProcessor> daoEventMap =
  new HashMap<DaoEvent.Type, DaoEventProcessor>();

/* Il dao, inizializzato nel metodo createDao(file) */
private Dao dao;

/* Trasforma un Date in stringa e una stringa in Date. Usato per mostrare
la data nella tabella. */
private SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");

/* IL modello della tabella dei record */
private DefaultTableModel recordModel = new DefaultTableModel();

/* La tabella dei record (usa il modello qui sopra) */
private JTable table;

/* La finestra della GUI. */
private JFrame window = new JFrame("JETutorial Test");

/** Invocato dal main. Chiede il percorso da usare come radice della base
dati e avvia l'ambaradan invocando createDao */
private void start() {
  String path = JOptionPane.showInputDialog(null,
    "Inserire il percorso della base dati.");
  File file = getDatabaseDir(path);
  if(file != null) {
    createDao(file);
  } else {
    JOptionPane.showMessageDialog(null, "La cartella " + path +
      " non esiste o non è accessibile");
  }
}

/* Crea o apre la base dati. Se tutto va bene, invoca createUI per
aprire la finestra del programma. Altrimenti sputa un messaggio di errore
in una finestra di dialogo. */
private void createDao(File file) {
  try {
    dao = new JEDao(file);
    /* Associa i callback ai tipi di evento per l'ascoltatore di eventi
    dao. */
    daoEventMap.put(DaoEvent.Type.UPDATE, update);
    daoEventMap.put(DaoEvent.Type.STORE, store);
    daoEventMap.put(DaoEvent.Type.DELETE, delete);

    /* Collega un ascoltatore di eventi al dao. */
    dao.addDaoEventListener(new DaoEventListener() {
      public void daoEventPerformed(DaoEvent e) {
        /* Usa la mappa dei callback su riempita per rimandare
        ai metodi di gestione dei singoli eventi (store, update,
        delete) */
        daoEventMap.get(e.getType()).process(e);
      }
    });

    /* Apre la base dati */
    dao.open();

    /* Se siamo qui è andato tutto bene. */
    createUI();
  } catch(DaoException ex) {
    showErrorDialog(ex);
  }
}
```

```
/** Crea l'interfaccia grafica. */
private void createUI() {
  /* La tabella avrà tre colonne di nome Id, Data e Nome */
  recordModel.setColumnIdentifiers(new String[] {"Id", "Data", "Nome", });

  /* Imposta il modello per la tabella. */
  table = new JTable(recordModel);

  /* Aggiunge la tabella ad un pannello a scorrimento. */
  JScrollPane tableContainer = new JScrollPane(table);


  /* Evita il comportamento predefinito di chiusura della finestra */
  window.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

  /* Aggiunge alla finestra un ascoltatore di eventi che chiude la finestra
  e chiude la base dati. */
  window.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) { closeApp(); }
  });

  /* Pulsante per la creazione di un nuovo record. */
  JButton newRecord = new JButton(new AbstractAction("New record") {
    public void actionPerformed(ActionEvent e) { newRecord(); }
  });

  /* Pulsante per cambiare i dati di un record. */
  JButton changeRecord = new JButton(new AbstractAction("Change record") {
    public void actionPerformed(ActionEvent e) { changeRecord(); }
  });

  /* Pulsante per eliminare un record. */
  JButton deleteRecord = new JButton(new AbstractAction("Delete record") {
    public void actionPerformed(ActionEvent e) { deleteRecord(); }
  });

  /* Contenitore dei pulsanti su creati. */
  JPanel buttons = new JPanel();
  /* Aggiunge i pulsanti al contenitore */
  buttons.add(newRecord);
  buttons.add(changeRecord);
  buttons.add(deleteRecord);
  /* Aggiunge il contenitore dei pulsanti alla finestra. */
  window.add(buttons, BorderLayout.NORTH);
  /* Aggiunge il contenitore della tabella alla finestra. */
  window.add(tableContainer, BorderLayout.CENTER);
  /* Assegna alla finestra le dimensioni necessarie e sufficienti a
  proiettare correttamente il suo contenuto */
  window.pack();
  /* Apre la finestra sullo schermo. */
  window.setVisible(true);
  /* Carica i record della base dati nella tabella */
  loadRecords();
}

/** Invocato alla pressione del pulsante New Record. Chiede all'utente
un nome e usa quel nome per creare un Record. Il Record così creato viene
inviato al Dao. Il dao lo passa alla base e dati e spara una notifica
STORE. La notifica viene intercettata dal listener che abbiamo registrato
presso il dao nel metodo createDao. Il listener, tramite la mappa dei
callback, "vede" l'evento STORE e invoca recordStore (metodo che si trova
più in fondo in questo sorgente) */
private void newRecord() {
```

```
    String name = JOptionPane.showInputDialog(window, "Inserire il nome");
    if(name != null) {
      Record record = new Record();
      record.setName(name);
      record.setTime(new Date());
      try {
        dao.store(record);
      } catch(DaoException ex) {
        showErrorDialog(ex);
      }
    }
}

/** Mostra una finestra di dialog con un messaggio che segnala l'eccezione
in argomento. */
private void showErrorDialog(Throwable ex) {
    JOptionPane.showMessageDialog(window, "Errore: " + ex.getMessage());
}

/** Il modello della tabella contiene delle righe di stringhe. Ogni riga
rappresenta un record. La prima stringa è l'id, la seconda una data, la
terza un nome. Questo metodo prende le stringhe della riga selezionata e
le impacchetta in un record. E' usato dai metodi deleteRecord e
changeRecord. */
private Record getSelectedRecord() {
    int selectedRow = table.getSelectedRow();
    if(selectedRow >= 0) {
      String id = table.getValueAt(selectedRow, 0).toString();
      String date = table.getValueAt(selectedRow, 1).toString();
      String name = table.getValueAt(selectedRow, 2).toString();
      Record record = new Record();
      try {
        record.setId(Long.parseLong(id));
        record.setTime(dateFormat.parse(date));
        record.setName(name);
        return record;
      } catch(Exception ex) {
        showErrorDialog(ex);
        return null;
      }
    } else {
      return null;
    }
}

/** Cambia i dati del record selezionato. In verità cambia solo il nome ma
è solo un esempio. */
private void changeRecord() {
    /* Ottiene il Record selezionato nella tabella. */
    Record record = getSelectedRecord();
    if(record != null) {
      /* Richiede un nuovo nome */
      String newName = JOptionPane.showInputDialog(window,
        "Inserire il nuovo nome");
      if(newName != null) {
        /* Imposta quel nome come nome del record. */
        record.setName(newName);
        try {
          /* Tramite il dao invia alla base dati una richiesta:
          "aggiorna un po' i dati di questo record. La base dati
          compie la stessa trafila indicata nel commento al metodo
          newRecord, solo che l'evento qui prodotto è un UPDATE. */
          dao.update(record);
        } catch(DaoException ex) {
```

```
            showErrorDialog(ex);
          }
        }
      }
    }

    /** Elimina il record selezionato nella tabella. Stessa manfrina di
    newRecord ma con un evento DELETE. */
    private void deleteRecord() {
      Record selectedRecord = getSelectedRecord();
      if(selectedRecord != null) {
        try {
          dao.delete(selectedRecord);
        } catch(DaoException ex) {
          showErrorDialog(ex);
        }
      }
    }

    /** Chiude l'applicazione, vale a dire chiude la finestra in argomento
    e chiude la base dati. */
    private void closeApp() {
      window.dispose();
      try {
        dao.close();
      } catch(DaoException ex) {
        ex.printStackTrace();
      }
    }

    /** Metodo invocato appena la finestra compare sullo schermo. Prende tutti
    i record contenuti nella base dati (tramite il dao) e li aggiunge alla
    tabella. */
    private void loadRecords() {
      Collection<? extends Record> records;
      try {
        records = dao.getRecords();
      } catch(DaoException ex) {
        showErrorDialog(ex);
        return;
      }
      for(Record r : records) {
        Object[] data = {
          String.valueOf(r.getId()),
          dateFormat.format(r.getTime()),
          r.getName(),
        };
        recordModel.addRow(data);
      }
    }

    /** Arriviamo qui tramite il callback "update", invocato dall'ascoltatore
    di eventi registrato presso il dao nel metodo iniziale "createDao". In
    pratica il dao ci ha appena detto che un Record è stato aggiornato. Noi
    rispondiamo a questa notifica aggiornando i dati che la tabella mostra in
    corrispondenza di quel record. */
    private void recordUpdate(DaoEvent e) {
      Record record = e.getRecord();
      for(int row = 0; row < table.getRowCount(); row++) {
        long id = Long.parseLong(table.getValueAt(row, 0).toString());
        if(record.getId().equals(new Long(id))) {
          recordModel.setValueAt(record.getId(), row, 0);
          recordModel.setValueAt(dateFormat.format(record.getTime()), row, 1);
          recordModel.setValueAt(record.getName(), row, 2);
```

```
      }
    }
  }

  /** E qui siamo arrivati grazie al callback "delete", invocato
  dall'ascoltatore di eventi registrato... come per recordUpdate, via.
  Stavolta eliminiamo la riga della tabella che contiene i dati del record
  eliminato dalla base dati. */
  private void recordDelete(DaoEvent e) {
    Record record = e.getRecord();
    for(int row = 0; row < table.getRowCount(); row++) {
      long id = Long.parseLong(table.getValueAt(row, 0).toString());
      if(record.getId().equals(id)) {
        recordModel.removeRow(row);
        return;
      }
    }
  }

  /** Come prima, più di prima, ti ameroooo... */
  private void recordStore(DaoEvent e) {
    Record record = e.getRecord();
    String[] data = {
      String.valueOf(record.getId()),
      dateFormat.format(record.getTime()),
      record.getName(),
    };
    recordModel.addRow(data);
  }

  /** Controlla se la stringa in argomento identifica un percorso di
  directory. In caso affermativo restituisce un File con quel percorso.
  Altrimenti restituisce null. E' usato nel metodo start, subito dopo aver
  chiesto all'utente di inserire il percorso della base dati, tanto per
  evitare struffolate. */
  private File getDatabaseDir(String path) {
    File file = null;
    boolean ok =
      path != null &&
      (file = new File(path)).exists() &&
      file.isDirectory();
    return ok ? file : null;
  }
}
```

Idea.

Un bel Dao generico?