Prendo parte dell'esercizio 2 e lo modifico rendendolo generico.

Ordine di creazione classi:
//GESTIONE STUDENTE
- Studente e Record: due interfacce implementate dalla classe seguente:
- RecordStudenti: crea l'oggetto che rappresenta uno Studente.

(- ModelloException)
- DatabaseManager: gestisce la comunicazione con il server

//APPLICAZIONE
- Filtro: interfaccia che viene implementata in Esercizio3
- Esercizio3: main.

ATTENZIONE: ho usato questo ordine perché è molto sequenziale e quindi facile da spiegare in un testo ma non è il modo migliore (ed è applicabile se ho già ben chiaro tutto ciò che mi serve); il modo migliore sarebbe partire dall'Esercizio3, aggiungendo ad ogni passo ciò che serve, così si può anche testarne il funzionamento prima che sia completo.

ECLIPSE

Esercizio3
src
- it.unipr.informatica.esercizio3
  - Esercizio3.java
- it.unipr.informatica.esercizio3.database
  - DatabaseManager.java
  - Filtro.java
  - Record.java
  - RecordStudenti.java
- it.unipr.informatica.esercizio3.modello
  - ModelloException.java
  - Studente.java
- confiurazione.properties

Studente.java

```
// Rispetto all'esercizio 2, ho spostato i “get” e “set” in RecordStudenti.java
package it.unipr.informatica.esercizio2.modello;

public class Studente{
  protected int getMatricola;
  protected String getCognome
  protected String getNome;
}
```

Record

```
package it.unipr.informatica.esercizio3.database;

public interface Record {
}
```

RecordStudenti

```
package it.unipr.informatica.esercizio3.database;
import it.unipr.informatica.esercizio3.modello.Studente;

public class RecordStudenti implements Record, Studente {
  private int matricola;
  private String cognome;
  private String nome;

  public RecordStudenti() {
    // Vuoto
  }

  @Override
  public int getMatricola() {
    return matricola;
  }
  public void setMatricola(int matricola) {
    this.matricola = matricola;
  }

  @Override
  public String getCognome() {
    return cognome;
  }
  public void setCognome(String cognome) {
    this.cognome = cognome;
  }

  @Override
  public String getNome() {
    return nome;
  }
  public void setNome(String nome) {
    this.nome = nome;
  }
}
```

ModelloException.java

```
package it.unipr.informatica.esercizio2.modello;

@SuppressWarnings("serial")
public class ModelloException extends Exception {
  public ModelloException(String messaggio) {
    super(messaggio);
  }
  public ModelloException(Exception origine) {
    super(origine);
  }
}
```

DatabaseManager.java

```
package it.unipr.informatica.esercizio3.database;
In automatico si aggiungono gli import

public class DatabaseManager {
  protected String url;

  public DatabaseManager() {
    try {
      ResourceBundle bundle = ResourceBundle.getBundle("configurazione");
      String clazz = bundle.getString("database.class");
      url = bundle.getString("database.url");
      Class.forName(clazz);
    } catch(Throwable throwable) {
                              throwable.printStackTrace();
    }
  }

// Rispetto all'esercizio 2, qui abbiamo generalizzato rendendo il codice molto più breve
public <X, Y extends X> List<X> leggiTabella(Class<Y> classe, Filtro<X> filtro) throws
ModelloException {  // X è un Object
  Connection connessione = null;
  Statement statement = null;
  ResultSet resultSet = null;
  try {
    connection = connetti();
-   String nomeClasse = classe.getName();  // Si blocca al primo . ; devo usare getSimpleName()
     String nomeClasse = classe.getSimpleName();  // Prende il nome della classe di "classe" (dove lo
useremo noi sarà "RecordStudenti").
    String nomeTabella = nomeClasse.substring("Record".length());  // Dal nome della classe
("RecordStudenti") estrale la sottostriga escludendo "Record" quindi estrae "Studenti".
    String sql = "SELECT * FROM " + nomeTabella;  // Non uso più i ? (come ad esempio nel caso del
nome della tabella).
    // sql sarà "SELECT * FROM STUDENTI" se quando richiamo il metodo "leggiTabella", gli passo la
classe "RecordStudenti"; se io avessi un'altra tabella da stampare come ad es. "Professori", mi basterebbe
passare con classe "RecordProfessori", e senza bisogno di cambiare la classe "DatabaseManager", potrei
leggere anche la tabella professori.
    // Questa "generalità" viene ovviamente mantenuta anche nelle istruzioni successive:
    statement = connection.createStatement();
    resultSet = statement.executeQuery(sql);
    List<X> risultato = new ArrayList<X>();

    while(resultSet.next()) {
      X record = classe.newInstance();  // Costruisce un nuovo oggetto di classe 'classe'
      Method[] metodi = classe.getMethods();

      for (int i = 0; i < metodi.length; i++) {
        String nomeMetodo = metodi[i].getName();
```

```
        if(nomeMetodo.startsWith("set")) {  // Serve un costruttore senza parametri; se non c'è, dà errore
          String nomeCampo = nomeMetodo.substring("set".length());
          Object valore = resultSet.getObject(nomeCampo);
          metodi[i].invoke(record, valore);  // Il primo parametro è l'oggetto su cui si invoca il setMatricola ed è un nuovo oggetto che abbiamo creato e il valore è quello ricavato in precedenza; aggiungendo ad esempio 12,13 il metodo viene applicato anche su 12 e 13
          // Dice: passami un qualsiasi dato basta che sia del tipo giusto
          // Se, a compile time, non so quanti parametri ho, allora mi costruisco un vettore a run time
        }
      }

      if(filtro.applica(record))
        risultato.add(record);
      }

      disconnetti(connection, statement, resultSet);
      return risultato;

    } catch(Exception exception) {
      exception.printStackTrace();
      disconnetti(connection, statement, resultSet);
      throw new ModelloException(exception);
    }
  }

  protected Connection connetti() throws SQLException {
    return DriverManager.getConnection(url);
  }

  protected void disconnetti(Connection connessione, Statement statement, ResultSet resultSet) {
    try {
      resultSet.close();
    } catch(Throwable throwable) {
      // Vuoto
    }


    try {
      statement.close();
    } catch(Throwable throwable) {
      // Vuoto
    }

    try {
      connessione.close();
    } catch(Throwable throwable) {
      // Vuoto
    }
  }
}
```

Filtro

```
package it.unipr.informatica.esercizio3.database;

public interface Filtro<X> {
        public boolean applica(X valore);
}
```

Esercizio3.java

```
package it.unipr.informatica.esercizio3;
In automatico si aggiungono gli import;

public class Esercizio3 {
  public static void main(String[] args) {
    try {
      //Costruisco il database manager
      DatabaseManager databaseManager = new DatabaseManager();

      List<Studente> studenti = databaseManager.leggiTabella(
                // Devo passargli una classe e, se conosco il nome a compile time, scrivo RecordStudenti.class.
                // Se non la conosco a compile time uso Class.forname....
                // Info: Forname costruisce l'oggetto classe in base alla classe che passo come parametro,
inizializzando i metodi statici
                        RecordStudenti.class, new Filtro<Studente>() {
                          @Override
                          public boolean applica(Studente valore) {
                            String cognome = valore.getCognome();
                            return cognome != null && cognome.startsWith("B");
                          }
                        }
                                                                        );
      for(Studente studente : studenti) {  // Faccio un for che li stampa tutti
        System.out.println(studente.getMatricola() + " " + studente.getCognome()+ " "+
studente.getNome());
      }
    } catch(Throwable throwable) {
        throwable.printStackTrace();
    }
  }
}
```

configurazione

```
database.class = org.apache.derby.jdbc.ClientDriver
database.url = jdbc:derby://localhost/Ateneo

// Usando questa struttura generica, il programma è più lento ma più riutilizzabile.
// Java Enterprise edition chiede solo l'interfaccia e java costruisce a run time la classe Studente che
l'utente non vedrà mai.
// Usiamo molto Object e ciò si porta dietro il cast che è il goto del passato.
```

// Bisogna usare i generici con i quali però si fanno molti più controlli a run time (tutte le X sono object; poi le posso sostituire per eliminare i cast).
// I generici si possono usare come parametro della classe però ad essere generico non è la classe ma il metodo quindi è il metodo che deve avere il parametro generico.
// In java si usano anche metodi generici impliciti (nel C++ bisogna inserire <T>).
// Voglio passare le funzioni come parametri (lamba expression) e da java8 si può.