Look and feel: imposta la grafica dei bottoni, etc
Swing: è pieno di designpatter (soluzioni piccole per problemi ricorrenti).
AWT: crea la finestra (contorni) e swing disegna i pixel al suo interno.
Dobbiamo svincolare l'interfaccia dall'applicazione.
Avrò un modello anche per la GUI.
- L'interfaccia è simile ad una classe ma nell'interfaccia posso creare metodi senza body come ad esempio: “public void metodo();” in una classe invece devo scriverlo così “public void metodo() {//vuoto}”.
E' “sottinteso” che un'interfaccia ha una classe derivata che ne implementa i metodi.

Ordine di creazione classi:
//GESTIONE STUDENTE
- Studente: dovrò lavorare sugli studenti quindi creo questa classe con lo scopo di poter utilizzare un oggetto "Studente" (con nome, cognome, matricola).

(- ModelloException)
- Modello: devo creare oggetti studente sfruttando i dati del DB quindi creo questa interfaccia con metodi che verranno implementati dalla classe seguente:
- DatabaseManager: gestisce la comunicazione con il DB

//GESTIONE INTERFACCIA
- ModelloTabella: classe che modifica la tabella fornita da swing integrandola con il Modello dello Studente.

- Vista: classe (interfaccia) che crea l'interfaccia dell'applicazione.
Contiene un solo metodo, che verrà implementato dalla classe seguente:
- VistaSwing: essendo l'interfaccia dell'applicazione, questa classe va creata assieme ad “Applicazione” ed “Esercizio2”.

//APPLICAZIONE
- Esercizio2: main che crea un'Applicazione
- Applicazione: classe che mette insieme il tutto.

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'Applicazione, aggiungendo ad ogni passo ciò che serve migliorandola, così si può anche testarne il funzionamento prima che sia completa.

ECLIPSE
Esercizio2
src
- it.unipr.informatica.esercizio2
  - Applicazione.java // Controller
  - Esercizio2.java // Main da lanciare
- it.unipr.informatica.esercizio2.database
  - DatabaseManager.java
- it.unipr.informatica.esercizio2.modello
  - Modello.java // Interface (che è come una classe) per accedere al DB
  - ModelloException.java
  - Studente.java
- it.unipr.informatica.esercizio2.swing
  - ModelloTabella.java
  - VistaSwing.java
- it.unipr.informatica.esercizio2.vista
  - Vista.java // Interface

Studente.java

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

public class Studente{
  protected int matricola;
  protected String cognome
  protected String nome;

  // Destro, source, “genera costruttore con i campi” (si può fare in molti altri casi, anche nei successivi)
  public Studente(int matricola, String cognome, String nome) {
    this.matricola = matricola;
    this.cognome = cognome;
    this.nome = nome;
  }
  // Destro, source, “genera Getter e Setters” → selezionare tutti e tre i campi; verrà generato in automatico tutto il codice seguente:
  public int get Matricola() {
    return matricola;
  }
  public String getCognome() {
    return cognome;
  }
  public String getNome() {
    return nome;
    // In java gli oggetti sono immutabili (una stringa non si può modifcare)
  }


  public void setNome(String nome) {
    this.nome = nome;
  }
  public void setCognome(String cognome) {
     this.cognome = cognome;
  }
}
```

ModelloException.java

```
// Classe usata per indicare le eccezioni trovate
package it.unipr.informatica.esercizio2.modello;

@SuppressWarnings("serial")  //sopprime un warming dicendo di non mostrarlo
public class ModelloException extends Exception {
  public ModelloException(String messaggio) {
    super(messaggio);
  }

  public ModelloException(Exception origine) {
    super(origine);
  }
}
```

Modello.java

```
package it.unipr.informatica.esercizio2.modello;
In automatico si aggiungono gli import

public interface Modello {  // Creo le funzioni che verranno implementate dalla classe “DatabaseManager” derivata di “Modello”
  public boolean aggiornaCognome(Studente studente, String cognome);
  public boolean aggiornaNome(Studente studente, String nome);
  public List<Studente> ricaricaStudenti() throws ModelloException;
}
```

DatabaseManager.java

```
package it.unipr.informatica.esercizio2.database;
In automatico si aggiungono gli import

public class DatabaseManager implements Modello {
  protected String url;

  //carico il driver
  public DatabaseManager() {
    try {
      ResourceBundle bundle = ResourceBundle.getBundle("configurazione");  // aggiunge lui .properties; sceglie anche in automatico la lingua (cosa che potrei impostare io)
      String clazz = bundle.getString("database.class");  // estrae la stringa che indica la classe del DB
      url = bundle.getString("database.url");  // estrae la stringa che indica l'url del DB
      Class.forName(clazz);  // carica la classe
    } catch(Throwable throwable) {
                throwable.printStackTrace();
    // se siamo in una applicazione per utente singolo, si può chiudere
    // altrimenti dico solo che non funziona niente, ma non chiudo
    }
  }

  @Override
  public boolean aggiornaCognome(Studente studente, String cognome) {
    Connection connessione = null;
    PreparedStatement statement = null;
    try {
      connessione = connetti();
      statement = connessione.prepareStatement("UPDATE STUDENTI SET COGNOME = ?
                              WHERE MATRICOLA = ?");
      statement.setString(1, cognome);
      statement.setInt(2, studente.getMatricola());
      statement.execute();

    } catch(Throwable throwable) {
      throwable.printStackTrace();
      disconnetti(connessione, statement, null);
      return false;
    }

    disconnetti(connessione, statement, null);
    return true;
  }
```

```java
@Override
public boolean aggiornaNome(Studente studente, String nome) {
  Connection connessione = null;
  PreparedStatement statement = null;

  try {
    connessione = connetti();
    statement = connessione.prepareStatement("UPDATE STUDENTI SET NOME = ?
                                WHERE MATRICOLA = ?");
    statement.setString(1, nome);

    statement.setInt(2, studente.getMatricola());
    statement.execute();

  } catch(Throwable throwable) {
    throwable.printStackTrace();
    disconnetti(connessione, statement, null);
    return false;
  }

  disconnetti(connessione, statement, null);
  return true;
}

@Override
public List<Studente> ricaricaStudenti() throws ModelloException {
  List<Studente> risultato = new ArrayList<Studente>();

  // le dichiaro fuori dal try perché altrimenti non si vedono
  // tutte le variabili che si vogliono usare nel catch vanno dichiarate fuori
  Connection connessione = null;
  PreparedStatement statement = null;
  ResultSet resultSet = null;

  try {
    connessione = connetti();
    statement = connessione.prepareStatement("SELECT * FROM STUDENTI");
    resultSet = statement.executeQuery();

    while(resultSet.next()) {  //estraggo e stampo i dati
      int matricola = resultSet.getInt("MATRICOLA");
      String cognome = resultSet.getString("COGNOME");
      String nome = resultSet.getString("NOME");
      Studente studente = new Studente(matricola, cognome, nome);
      risultato.add(studente);
    }

  } catch(Throwable throwable) {
    throwable.printStackTrace();
    disconnetti(connessione, statement, resultSet);
    throw new ModelloException("Impossibile ricaricare la tabella");
  }

  disconnetti(connessione, statement, resultSet);
  return risultato;  //return new ArrayList<Studente>();
}
```

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

  protected void disconnetti(Connection connessione, Statement statement, ResultSet resultSet) {
    //dobbiamo chiudere tutto nell'ordine in cui abbiamo aperto
    try {
      resultSet.close();
    } catch(Throwable throwable) {
      // Vuoto
    }

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

    try {
      connessione.close();
    } catch(Throwable throwable) {
      // Vuoto
    }
  }
  //serve un file di configurazione e abbiamo due opzioni quindi o lo mettiamo nella cartella resurces (res) o nella cartella dei sorgenti che poi si ritrova copiato uguale nella cartella class (lo metto in src; è un file .propertis)
}
```

ModelloTabella.java

```
package it.unipr.informatica.esercizio2.vista;
In automatico si aggiungono gli import

public class ModelloTabella implements TableModel {
  protected List<Studente> dati;
  protected Modello modello;  //valore che poi passerò come parametro e ciò genererà parecchi errori da sistemare


  public ModelloTabella(Modello modello) {
    this.modello = modello;
    this.dati = new ArrayList<Studente>();  // arraylist si appoggia ad un array; linkthelist è una lista concatenata; meglio l'array per questo caso perché io ricevo l'indice della riga ma poi devo "viaggiare" nella lista dei dati delle colonne
  }

  public ModelloTabella(Modello modello, List<Studente> dati) {
    this.modello = modello;
    this.dati = dati;
  }

  @Override
  public int getColumnCount() {
    return 3;
  }
```

```
  @Override
  public String getColumnName(int indice) { // Scelgo nome colonne
    switch(indice) {
      case 0:
        return "Matricola";
      case 1:
        return "Cognome";
      case 2:
        return "Nome";
      default:
-       throw new RuntimeException( "Indice non valido: " + indice);  //le runtimeException NON sono nella segnatura all'inizio quindi un utente potrebbe dimenticarsi di catturarla
-       throw new InvalidParameterException( "Indice non valido: " + indice);
        // Il parametro è l'indice, l'argomento è il valore dell'indice.
        throw new IllegalArgumentException("Argomento indice non valido: " + indice);
    }
  }

  @Override
  public Class<?> getColumnClass(int indice) {  // generato;  con le parentesi <> si usa un template (tipo dato generico); posso anche non scrivere <?> ed erediterei da object (classe da cui derivano tutte le classi) ma avrei un errore scrivendo object, quindi ho 3 casi di generico:  <?>, <object>, <> ; con il ? è il migliore (tutto deriva da object anche se ad esempio la classe Array non deriva da object e quindi con un array non funzionerebbe)
    switch(indice) {
      case 0:
        return int.class;  // Classe che rappresenta int come classe perché int non è una classe ma un dato primario (con la nuova sintassi funziona anche int.class)
      case 1:  // Se non scrivo niente, il case 1 risulta come il case 2.
      case 2:
        return String.class;  // Utilizzo (ritorna) un oggetto della classe;
      default:
-       throw new InvalidParameterException( "Argomento indice non valido: " + indice);
        throw new IllegalArgumentException("Argomento indice non valido: " + indice);
    }
  }

  @Override
  public boolean isCellEditable(int riga, int colonna) {  //da usare per editare; la matricola non la edito quindi nel caso colonna = 1, restituisco false

    if(colonna == 0)
      return false;
    return true;
  }

  @Override
  public void setValueAt(Object valore, int riga, int colonna) {
    Studente studente = dati.get(riga);

    if(colonna == 1) {
      String cognome = (String)valore;  //faccio un cast (specifico) a stringa perché la prima colonna non c'è quindi sono sicuramente stringhe i valori modificabili
      if(modello.aggiornaCognome(studente, cognome))  // "modello.aggiornaCognome" svolge le istruzioni SQL di aggiornamento e restituisce true se è riuscito ad aggiornare
        studente.setCognome(cognome);
    } else {
```

```
      String nome = (String)valore;
      if(modello.aggiornaNome(studente, nome))
        studente.setNome(nome);
    }
  }

  @Override
  public int getRowCount() {
-   return 0;  //da usare quando faccio il test per vedere se va, poi va tolto
    return dati.size();
  }

  @Override
  public Object getValueAt(int riga, int colonna) {
-   return null;  // Da usare nella prova intermedia, poi si tolglie

    if(riga > dati.size() || riga<0)
      throw new InvalidParameterException( "Argomento riga non valido: " + riga);

    Studente studente = dati.get(riga);
    switch(colonna) {
      case 0:
        return studente.getMatricola();
      case 1:
        return studente.getCognome();
      case 2:
        return studente.getNome();
      default:
        throw new InvalidParameterException( "Argomento colonna non valido: " + colonna);
    }
  }

  @Override
  public void addTableModelloListener( TableModelListener listener ) {
    // Vuoto
  }

  @Override
  public void removeTableModelloListener( TableModelListener listener ) {
    // Vuoto
  }
  // I listener rilevano quando avviene una modifica, ma dato che qui non ne abbiamo, li lasciamo vuoti.
  // Si usano molto poco su basi di dati
}
```

Vista

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

public interface Vista {
  public void apriFinestraPrincipale();
}
```

VistaSwing.java

```
package it.unipr.informatica.esercizio2.swing;
In automatico si aggiungono gli import

public class VistaSwing implements Vista {
  protected Applicazione applicazione;

- final JTable tabella = new JTable(); // Rende un riferimento immutabile (puntatore costante in C++) si fa per usare "tabella" nella inner class successiva.
  protected JTable tabella;  // Al posto del final;

  protected JFrame finestra;
  protected Icon iconaAggiorna;
  protected Modello modello;

  public VistaSwing(Applicazione applicazione) {
    try {
      UIManager.setLookAndFeel("javax.swing.plaf.nimbus.NimbusLookAndFeel");  //imposta il look and feel SE C'E (per migliorare lo stile)
    } catch(Throwable throwable) {
      // Vuoto
    }

    this.applicazione = applicazione;
    this.modello = applicazione.getModello();
    this.tabella = new JTable();

    tabella.setModel(new ModelloTabella(modello));
      //creo un'icona con un immagine contenuta nel class pack; uso l'URL dell'immagine utilizzando una funzione che mi compone l'URL
    URL url = getClass().getResource("/aggiorna.gif");  //facendo getClass ottengo l'oggetto classe che è collegato al class loader; questa chiamata è quindi chiamata sul class loader che ritorna un URL (viene restituito null)
    if(url != null)
      iconaAggiorna = new ImageIcon(url);
    else
      iconaAggiorna = null;
  }
    // non c'è il setColonne, etc perchè la JTable tiene separata la vista dei dati dai dati veri (si usano i model); la J è messa solo per evitare la sovrapposizione dei nomi con quelli di AWT


  @Override
  public void apriFinestraPrincipale() {
    JButton bottone = new JButton();
    bottone.setText("Aggiorna Tabella");

    bottone.addActionListener(
      new ActionListener(new ActionListener) {  // Dovrei costruire una classe esterna, ma in questo modo la creo dentro come oggetto (inoltre, questa inner class è senza nome e ciò mi è comodo
        @Override
        public void actionPerformed(ActionEvent event ) {  // Ricarico la tabella con i dati, ma io sono nella vista quindi per usare i dati devo passare per il controller dell'applicazione
-         Vista.this.applicazione...  //Esplicito perché le inner class sono contenute ed ho una parentela di classi
          aggiornaTabella();
        }
      } );
```

```
    JPanel pannelloInBasso = new JPanel();
    pannelloInBasso.setLayout(new FlowLayout(FlowLayout.CENTER));
    pannelloInBasso.add(bottone);
-   pannelloInBasso.setOpaque(true);  // Non trasparente
-   pannelloInBasso.setColor(new Color(255,255,0));

    // I bottoni dei tool non vengono selezionati con space o tab dal focus.
    // Posso costruire un icon tramite lettura di immagini:
    JButton toolAggiornaTabella;
    if(iconaAggiorna == null)
      toolAggiornaTabella = new JButton("Aggiorna Tabella");
    else
      toolAggiornaTabella = new JButton(iconaAggiorna);

    toolAggiornaTabella.setFocusable(false);
    toolAggiornaTabella.addActionListener( new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent evento) {
          aggiornaTabella();
        }
      }
    );

    JToolBar toolBar = new JToolBar("Sistema di Gestione Studenti");
    toolBar.add(toolAggiornaTabella);

    JPanel pannelloPrincipale = new JPanel();
    pannelloPrincipale.setLayout(new BorderLayout());
    pannelloPrincipale.add(toolBar, BorderLayout.NORTH);
    pannelloPrincipale.add(pannelloInBasso, BorderLayout.SOUTH);
    pannelloPrincipale.add(new JScrollPanel(tabella), BorderLayout.CENTER );  // Creo uno scroll per la tabella
dato che sarà lunga e la piazzo al centro (le scoll compaiono solo se il contenuto sborda)
    // Lo scroll contiene un solo contenuto

    JMenuItem aggiornaTabellaMenuItem = new JMenuItem("Aggiorna tabella");
    aggiornaTabellaMenuItem.setMnemonic(KeyEvent.VK_A);
    aggiornaTabellaMenuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_F5, 0));

    aggiornaTabellaMenuItem.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent evento) {
          aggiornaTabella();
        }
      }
    );

    JMenu menuTabella = new Jmenu("Tabella");  //il Menu è standard in tutti i programmi quindi lo aggiungo
    menuTabella.setMnemonic(KeyEvent.VK_T);
    menuTabella.add(aggiornaTabellaMenuItem);
    //potrei mandare un thread quando aggiorno la tabella che non mi blocca la GUI e che blocca i tasti aggiorna
finché non ho aggiornato.

    JMenuItem esciMenuItem = new JmenuItem("Esci");  //aggiungo metodi
    esciMenuItem.setMnemonic(KeyEvent.VK_E);
    esciMenuItem.addActionListener(new ActionListener() {
```

```
      @Override
      public void actionPerformed(ActionEvent evento) {
        esci();
      }
    }
  );

  JMenu menuFile = new JMenu("File");
  menuFile.setMnemonic(KeyEvent.VK_F);
  menuFile.add(esciMenuItem);

  JMenuBar menuBar = new JMenuBar();
  menuBar.add(menuFile);
  menuBar.add(menuTabella);

  finestra = new JFrame(); // Frame da cui si parte
  finestra.getContentPane().add(pannelloPrincipale);
  finestra.setTitle("Sistema di Gestione Studenti");
  // Swing parte dagli oggetti e la pagina si adatta; HTML parte dalla pagina e allinea quello che c'è dentro
  // Certe pagine internet fissano la pagina e ciò ha senso; lasciare invece la pagina libera, crea problemi
  // Dobbiamo quindi fissare la dimensione della finestra in base a quanto è grande il monitor (noi diamo una
dimensione iniziale senza chiedere al monitor quanto è grande)
  // Scegliamo circa 600x600
  finestra.setBound(0,0,600,600);  // (in alto a destra) x,y, larghezza,altezza
  // Swing apre un thread di AWT e quando chiudo la finestra, il programma rimane aperto perché quel thread resta
aperto
- finestra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  //Chiudendo la finestra termina tutto il
programma/processo
  // Posso inserire "// TODO ..." per ricordarmi cosa devo fare guardando la tasks list

  // TODO da rivedere
  finestra.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
  finestra.addWindowListener(new WindowAdapter() {
  //diciamo cosa fare nei vari casi
  //per evitare di mettere eventi vuoti, uso WindowAdapter che mi permette di scrivere solo ciò che userò
      @Override
      public void windowClosing(WindowEvent evento) {
        esci();
      }
    }
  );

  finestra.setJMenuBar(menuBar);
  finestra.setVisible(true);  // Mostro la finestra
 }

 protected void aggiornaTabella() {
  try {
    List<Studente> dati = applicazione.ricaricaStudenti();

    TableModel modelloTabella = new ModelloTabella(modello, dati);

    tabella.setModel(modelloTabella);
  } catch(Throwable throwable) {
-   JOptionPanel.showMessageDiaLog(finestra [blocca pa finestra], "impossibile aggiornare finestra"
                    [messaggio], "Errore" [titolo], JOptionPane.ERROR_MESSAGE [tipo di errore] );
```

```
      //in questo statement ci sono parti che posso riutilizzare in altre situazioni e qui solo la stringa varia quindi si può fare una funzioncina che riceve una stringa e che utilizzerò tutte le volte al posto di questa lunga riga
      errore("Impossibile aggiornare la tabella");
    }
  }

  protected void esci() {
    if(seiSicuro())
      applicazione.esci();
  }

  protected boolean seiSicuro() {  //richiesta che fa quando provo a chiudere il programma (finestra)
    int risposta = JOptionPane.showConfirmDialog(finestra, "Sei sicuro?", "Messaggio",
JOptionPane.YES_NO_OPTION);
    return risposta == JOptionPane.OK_OPTION;
  }

  protected void errore(String messaggio) {
    JOptionPane.showMessageDialog(finestra, messaggio, "Errore", JOptionPane.ERROR_MESSAGE);
  }
}
```

Esercizio2.java

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

public class Esercizio2 {
  public static void main(String[] args) {
    new Applicazione().run();
  }
}
```

Applicazione.java

```
package it.unipr.informatica.esercizio2;
In automatico si aggiungono gli import

public class Applicazione implements Runnable {
  protected Vista vista;
  protected Modello modello;

  @Override   // Sto sovrascrivendo qualcosa (metodo) della classe base o per implementarlo (funziona per versioni 5 o superiori)
  public void run() {   // Deve creare un modello, una vista e attivare la vista
    modello = newDatabaseManager(); // Cambia il modello? cambia solo questo
    // Iniziamo dalla vista (finestra) dopo averla creata
    vista = new VistaSwing(this); // Cambia la vista? cambio solo questo

    vista.apriFinestraPrincipale();
  }

  public Modello getModello() {
    return modello;
  }
```

```
  public void esci() {
    System.exit(0);
  }

  public List<Studente> ricaricaStudenti() throws ModelloException {  // Creo la connessione
    return modello.ricaricaStudenti();
  }
}
```

configurazione.properties

```
database.class = org.apache.derby.jdbc.ClientDriver
database.url = jdbc:derby://localhost/Ateneo  //posso modificare derby e il driver a seconda della macchina, cambiando il DB
```