UNIVERSITÀ DI PISA

Department of Information Engineering

Information Systems
Task 1 Documentation

Students:
Adriano Botti
Antonio Le Caldare
Francesco Merola
Giacomo Ponziani

Contents

List of Figures

Application Specifications

The goal of the application that we implemented is to provide a way for both students and professors to manage the registration process for exams. More specifically, we want the application to exert the following functionalities:

- For Students:
    1. Check past exams results
    2. Register to an exam date
    3. Delete an exam registration
- For Professors:
    1. Add grades to an exam
    2. Create a new exam date

Requirements and Use Cases

Functional Requirements

The Professor:

1. shall be able to insert an exam, associated with a course he holds, in a date of choice
2. shall not be able to inster an exam for a date precedent to the current date
3. shall be able to insert the corresponding grade for a student in his registration for that exam.
4. shall insert all grades in the exact date of the exam

The Student:

1. shall be able to check the results of past exams
2. shall be able to register to an exam not yet took
3. shall not be able to register to an exam after the exam date.
4. shall be able to register to more successive exams for the same course
5. If the student registered to successive exams for the same course he just got a mark for, then those future registrations shall be deleted
6. shall be able to deregister from an exam he was previously registered to
7. shall not be able to deregister from an exam alredy took
8. shall not be able to deregister from an exam after the exam date.

Non-Functional Requirements

For the application we identified Consistency and Availability as the two most important non-functional requirements.

Use Case Diagram

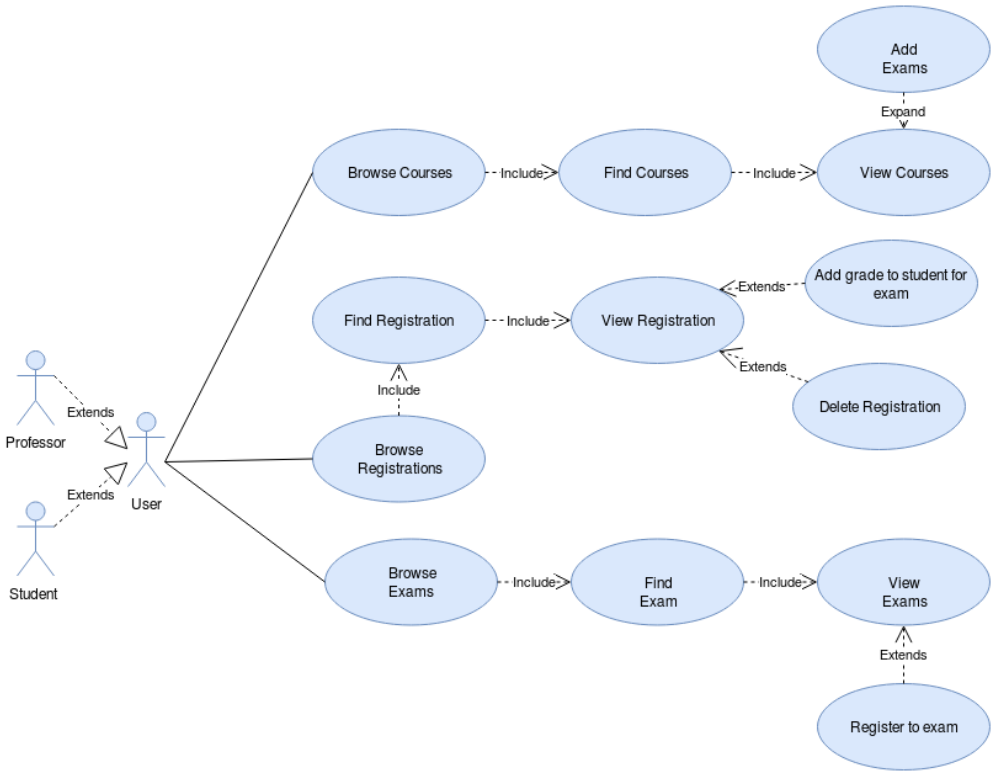

Figure 1: Use Case Diagram

- **Add Exam** : This operation can only be performed by a Professor.
  He/She will be able to browse his/her tenured courses, select the one to add a new exam to and then add it.
- **Add Grade** : This operation can only be performed by a Professor.
  He/She will be able to browse the registrations for his/her tenured courses, select the registration to add a grade to and then add it.
- **Register to Exam** : This operation can only be performed by a Student.
  He/She will be able to browse all the exams to which is possible to register, select one and perform the registration.
- **Deregister to Exam** : This operation can only be performed by a Student.
  He/She will be able to browse his/her active registrations, select one and deregister from it.
- **See Grades** : This operation can only be performed by a Student.
  He/She will be able to browse his/her registrations and show only the one with a sufficient.

For more detail and a step-by-step description of the different scenarios, see chapter *User's Manual*.

UML Class and Entity-Relationship Diagrams

UML Class Diagram

The UML Class Diagram shows the main objects involved in our application, with the respective logical relationships. From this model we derived an Entity-Relationship Diagram, where the

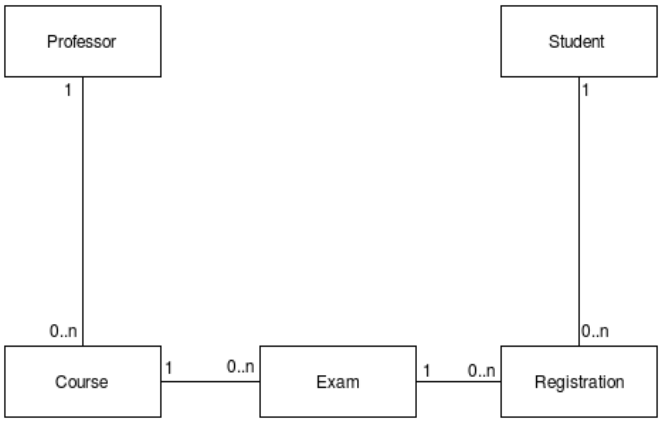

Figure 2: UML Class Diagram

dependecies are more deeply identified. Finally we will show the final database schema implemented on the MySQL RDBMS.

ER Diagram

Names definition

Here we define in detail the terms for the main entites and relationships we will use in the following:

- *Student*
  A student is an entity which is able to perform the operations already defined in the requirements. He's an actor for our application.
- *Professor*
  A professor is an entity which is able to perform the operations already defined in the requirements. He's an actor for our application.
- *Course*
  A course is held by one and only one professor. The course object only includes information about its name, cfu and the helding professor. It holds no information about when exams for that course will take place.
- *Exam*
  An exam represents the actual date of the examination for a course.
- *Exam Result* An exam result relates an exam to all the students who registered to that exam, adding the information of the grade, if meaningful.

Entities

Entity	Description	Attributes
Student	Holds all the information related to the students	• $\underline{id}$ • $name$ • $surname$
Professor	Holds all the information related to the professors	• $\underline{id}$ • $name$ • $surname$
Course	Holds the information related to the courses	• $\underline{id}$ • $name$ • cfu • $professor$
Exam	Holds all the new and past exams	• $\underline{course(ext)}$ • $\underline{date}$

Relationships

Relationship	Description	Participants	Attributes
Teaching	Links Professors to their held courses	• Professor(1,N) • Course(1,1)	
Exam Result	Links students to exams, with the respective grade	• Student(0,N) • Exam(0,N)	• *grade*
Exam Date Creation	Links the courses to the exams through a date	• Course(0,N) • Exam(1,1)	

ER Diagram

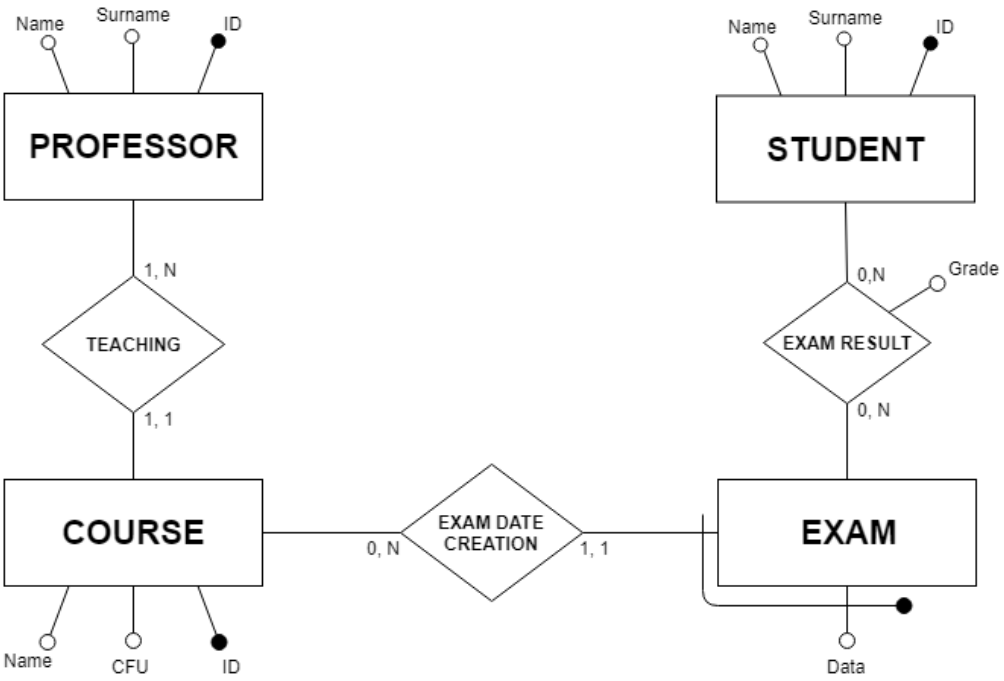

Figure 3: Entity - Relationship Diagram

Database Implementation

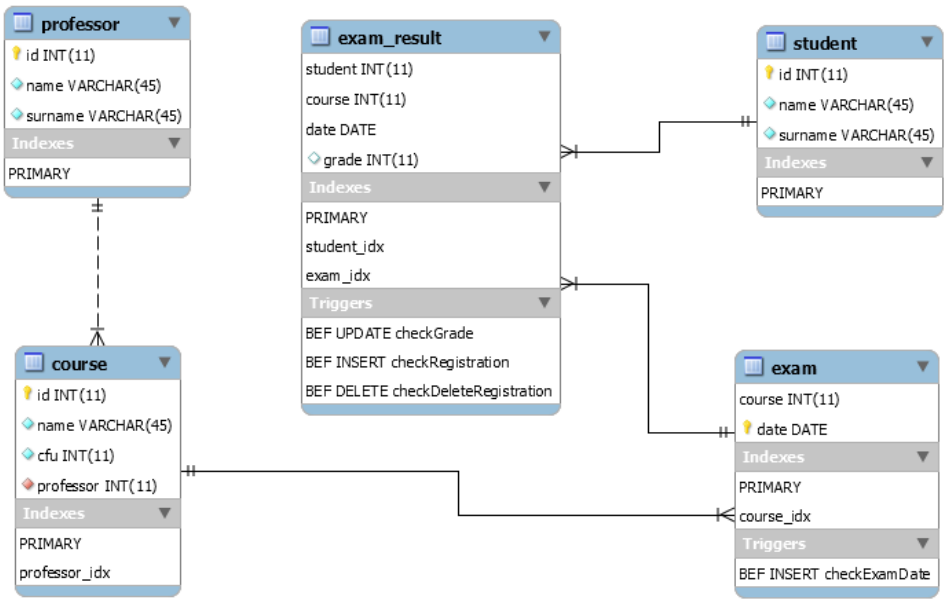

Figure 4: MySQL Schema

Implementation

Project implementation is based on two main packages, *task.db* and *task.gui*, and database class objects implementation.

task.db package

This package includes only one class, *DBManager*, which is based on Singleton design pattern. Each method implements a CRUD operation on MySQL database using jpa api methods. A list of all the methods:

- public List<Course> findCourse(int profID);
  find all courses given a professor ID
- public List<Exam> findExam(int studId);
  find all exams registered by a student indentified by its ID
- public List<Registration> findRegistrationProfessor(int profId);
  find all registrations opened by professor identified by the parameter ID
- public List<Registration> findRegistrationStudent(int studentId, boolean toDo);
  find all registrations applied to student identified by student ID. If toDo is true, show only registrations without grade, otherwise return all registration with grade
- public void insertExam(int courseID, LocalDate date);
  add exam for specified courseID at specified date
- public void insertRegistration(int studentId, Exam examDetached, @Nullable Integer grade);
  insert registration for specified student and exam (can be detached from jpa context). grade can be null, in that case it means that a registration is inserted without a grade
- public void updateRegistration(Registration reg, int grade);
  update grade to a registration
- public void deleteRegistration(int studentId, Exam exam);
  delete registration given its key (composite key of studentid + exam)

Implementation of findRegistrationStudent(int studentId, boolean toDo)

This implementation and the following one use JPQL language instead of SQL.

```
public List<Registration> findRegistrationStudent(int studentId, boolean toDo) {
List<Registration> resultList;
   try {
      entityManager = factory.createEntityManager();
      Query query =
         entityManager.createQuery("SELECT r FROM Registration r " +
         "WHERE r.student.id = :studentId AND r.grade IS "
         + ((toDo) ? "NULL" : "NOT NULL"));
      query.setParameter("studentId", studentId);
      resultList = query.getResultList();
   } catch (Exception ex) {
      throw ex; // Let GUI show the exception
   } finally {
      entityManager.close();
```

```
    }

    return resultList;
}
```

Implementation of findExam(int studId)

```
public List<Exam> findExam(int studId) {
    List<Exam> resultList;
    try {
        entityManager = factory.createEntityManager();
        Query query = entityManager.createQuery("SELECT e FROM Exam e "+
            "WHERE (SELECT count(r) FROM Registration r " +
            "WHERE r.student.id = :studId AND r.exam.id = e.id " +
            "OR (r.exam.course = e.course AND r.grade IS NOT NULL) " +
            ") = 0");
        query.setParameter("studId", studId);
        resultList = query.getResultList();
    } catch (Exception ex) {
        throw ex;
    } finally {
        entityManager.close();
    }

    return resultList;
}
```

task.gui package

Includes classes related to gui implementation, implemented using JavaFX API.

Database object files

Database objects implementation derived from Task0 have been converted to a JPA implementation. Although this phase should have been a simple annotation task, it caused some issues with composite keys implementation. JPA requires that each composite key should be implemented as an @Embeddable class and instanced into the main object class with a @EmbeddedID reference (check tutorial for more details). In addition to this, each composite key member should have a duplicated reference inside the main class, annotated with @MapsID, which realizes the relation of composite key members. Also, JPA specification requires that each entity class must implement a costructor with no parameters. We choosed also to use every annotation on getter and setter methods instead of entity class members, although it does not make any difference, apart from be able to intercept every get and set operation made by JPA, which in any case we don't use. Student and Professor implementation is straighforward.

Student

```
import javax.persistence.*;

@Entity
@Table(name = "student")
public class Student {
    private int id;
    private String name;
    private String surname;

    public Student() {}

    public Student(int id, String name, String surname) {
```

```
        this.id = id;
        this.name = name;
        this.surname = surname;
    }

    @Column(name = "id")
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public int getId() { return id; }

    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getSurname() { return surname; }
    public void setSurname(String surname) { this.surname = surname; }
}
```

Professor implementation is similar to this.

Registration

```
import javax.persistence.*;
import java.io.Serializable;
import java.sql.Date;

@Entity
@Table(name = "exam_result")
public class Registration {

    @Embeddable
    public static class RegistrationId implements Serializable {
        private int student;
        private Exam.ExamID exam;

        public RegistrationId() {}

        public int getStudent() { return student; }
        public void setStudent(int student) { this.student = student; }
        public Exam.ExamID getExam() { return this.exam; }
        public void setExam(Exam.ExamID exam) { this.exam = exam; }

        @Override
        public boolean equals(Object obj) {
            if(obj == null || obj.getClass() != this.getClass())
                return false;

            RegistrationId regobj = (RegistrationId)obj;
            return student == regobj.getStudent()
                && exam.equals(regobj.getExam());
        }

        // hashcode should be implemented in case of using
        // @OneToMany relations using HashMaps
        @Override
        public int hashCode() {
            return super.hashCode();
        }
    }



    public Registration () {}
```

```
    public Registration (Student student, Exam exam, Integer grade) {
        this.exam = exam;
        this.grade = grade;
        this.student = student;
        this.id = new RegistrationId();
        this.id.setExam(exam.getId());
        this.id.setStudent(student.getId());
    }

    private RegistrationId id;
    @EmbeddedId
    public RegistrationId getId() { return id; }
    public void setId(RegistrationId id) { this.id = id; }

    // ===== Key fields =====
    private Student student;

    @MapsId("student")
    @JoinColumn(name="student", referencedColumnName="id")
    @ManyToOne
    public Student getStudent () { return student; }
    public void setStudent(Student student) { this.student = student; }

    private Exam exam;

    @MapsId("exam")
    // il join tra Registration e Exam va fatto su due campi contemporaneamente:
    // exam_result.course = exam.course AND exam_result.date = exam.date
    @JoinColumns({
          @JoinColumn(name="course", referencedColumnName="course"),
          @JoinColumn(name="date", referencedColumnName="date")
    })
    @ManyToOne
    public Exam getExam() { return this.exam; }
    public void setExam(Exam exam) { this.exam = exam; }

    // ===== Additional fields =====
    private Integer grade;
    @Column(name = "grade")
    public Integer getGrade() { return grade; }
    public void setGrade(Integer grade) { this.grade = grade; }
}
```

Exam class implementations is similar to Registration implementation. We showed this to show an example of use of an embedded reference inside an embedded class.

Key-Value Feasibility Study

The first version of the application stores all its data in a relational database managed by the MySQL RDBMS. The main non-functional requirements are availability and consistency ad they are both guaranteed by the relational database since the application handles a limited amount of data. Availability may be nomore guaranteed if the number of users grows large, i.e. the number of students. The system may responds slower when too many users are active at the same time since in this scenario the relational database has to perform a lot of complex operations to fullfill their demands resulting in not negligible waiting times.
Operations that work with exam registrations are the most expensive and time consuming as they require to perform a join on multiple tables of the database. Reducing this kind operations may be a possible solution to improve the system performance: it may be convenient to use a key-value database in parallel with the relational one and spread different operations between them.

The relational database still stores all the business data and the key-value database stores only data relative to registrations. The two copies registrations data make the storage system redundant but consistency is always guaranteed by the fact that write operations are committed on both the databases or canceled. When a write operation occurs both the databases update consistently their data and if one of two can not commit the update the other rollbacks and it is so possible to perform a write operation if and only if both databases are on. If a read operation regards single-table data it is redirected towards the relational database, if instead it regards registrations it is redirected towards the key-value database. In the latter case it may happen that the key-value database is anavailable, in this case the read operation is redirected towards the relational database.

The key is composed of:

- id of the course of the exam;
- date of the exam;
- id of the professor of the course;
- id of the student.

The values, stored separetly, are:

- name of the student;
- surname of the student;
- name of the professor;
- surname of the professor;
- name of the course;
- number of CFU of the course;
- grade assigned by the professor.

A key-value database is so used to take advantage of its semplicity and velocity to speedup read operations on regitrations data.

User's Manual

The application starts and shows a minimal user interface, fig. 5, composed of:

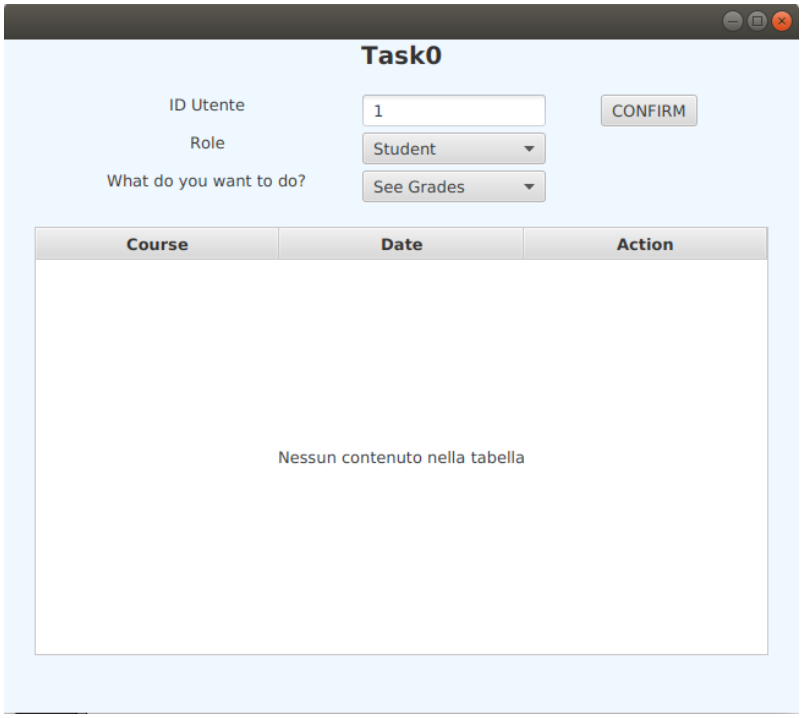

Figure 5: User Interface when the application starts

- *UserID* field, where the user specifies his id number;
- *Role* choise box, whith the option:
  1. Professor, in case the user is a Professor;
  2. Student, in case the user is a Student.
- *What do you want to do?* choise box, where the user specifies the action he wants to perform. The options change according to the role chosen. Professor can select among:
  1. *Add Exam* if he wants to add an exam;
  2. *Add Grade* if he wants to add a grade.

  A Student can select among:
  1. *Register to Exam* if he wants to register to an exam;
  2. *Deregister to Exam* if he wants to deregister to an exam;
  3. *See Grades* if he wants to see the grades he got.
- *Confirm* button;
- A table showing the results of the selected operation. The layout of the table can change according depending on the selected operation.

Professor

A Professor:

1. Inserts his ID number into the *UserID* field;
2. Selects *Professor* in the *Role* choise box;
3. Selects the action he wants to perform;
4. Pushes the *Confirm* button.

Add Exam

If the Professor select the *Add Exam* option, the table, fig. 6, is populated with a list of all the courses he is currently holding. Each element of the list is composed of:

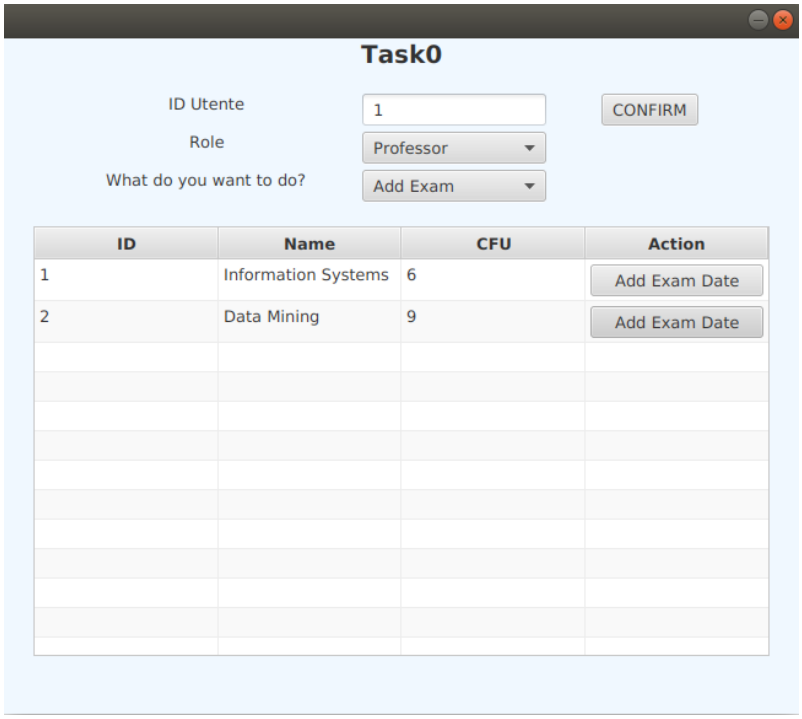

Figure 6: Example of Add Exam

- *ID*, the id of the course;
- *Name*, the name of the course;
- *CFU*, the number of credits assigned to the course;
- *Add Exam Date*, button the professor has to push in order to add an exam corresponding to the course.

If the Professor pushes one of the *Add Exam Date* buttons a confirm dialog is presented and it asks for the date of the exam to insert, fig. **??**. If the Professor wants to confirm he:

1. Selects the date using the datepicker;
2. Pushes the *Confirm* button in the dialog.

To go back and undo the operation the Professor just pushes the *Delete* button.

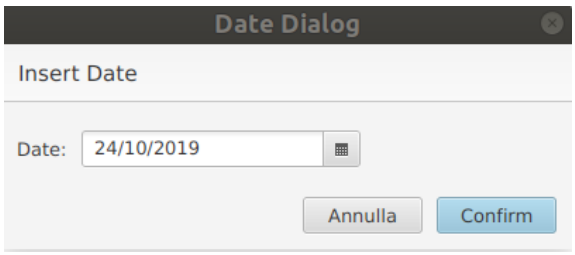

Figure 7: Example of Add Exam dialog

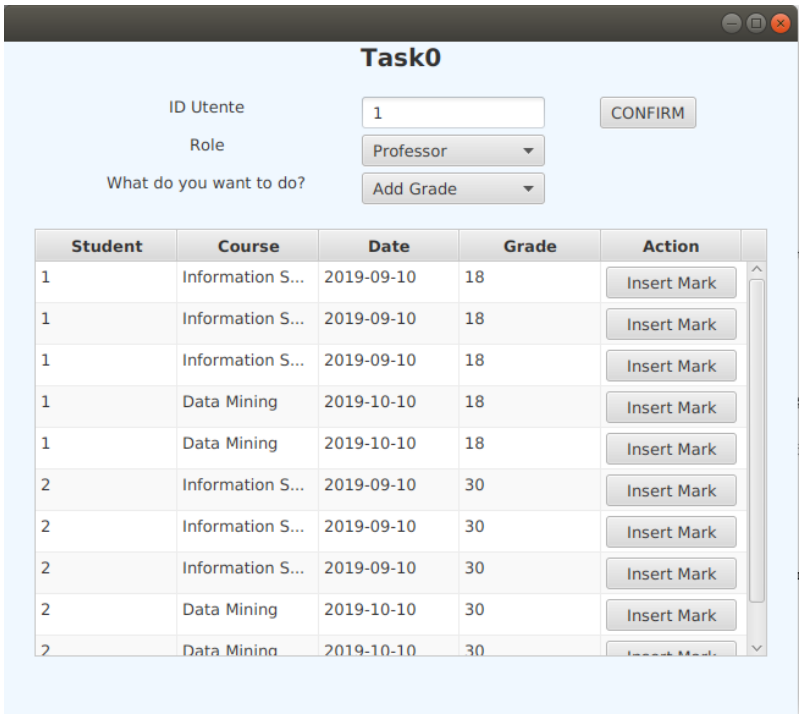

Figure 8: Example of Add Grade

Add Grade

If the Professor select the *Add Grade* option, the table, fig. 8, is populated with a list of all the registrations to the courses he is currently holding. Each element of the list is composed of:

- *Student*, the id of the student enrolled to the exam;
- *Course*, the name of the course;
- *Date*, the date of the exam;
- *Insert Mark* button that the professor has to push in order to insert a grade to the corresponding registration.

If the Professor pushes one of the *Insert mark* buttons a confirm dialog is presented and it asks for the grade of the exam to insert, fig. 9. If the Professor wants to confirm he:

1. Inserts the grade in the corresponding field;
2. Pushes the *Confirm* button in the dialog.

To go back and undo the operation the Professor just pushes the *Delete* button.

Student

A Student:

1. Inserts his ID number in the *UserID* field;

Figure 9: Example of Add Grade dialog

2. Selects *Student* in the *Role* choise box;
3. Selects the action he wants to perform;
4. Pushes the *Confirm* button.

Register to Exam

If the Student select the *Register to Exam* option, the table, fig. 10, is populated with a list of all the available exams. Each element of the list is composed of:

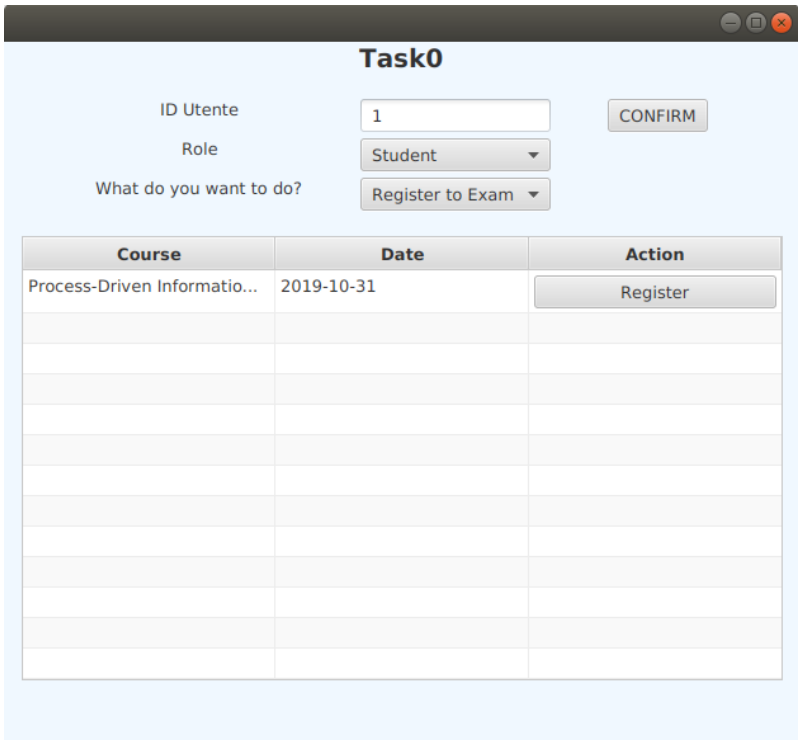

Figure 10: Example of Register to Exam

- *Course*, the name of the course;
- *Date*, the date of the exam;
- *Register*, button the student has to push in order to register to the selected exam.

If the Student pushes one of the *Register to Exam* the table is updated so that it shows all the exams to which the Student is not enrolled.

Deregister to Exam

If the Student select the *Deregister* option, the table, fig. 11, is populated with a list of all the registrations corresponding to the Student. Each element of the list is composed of:

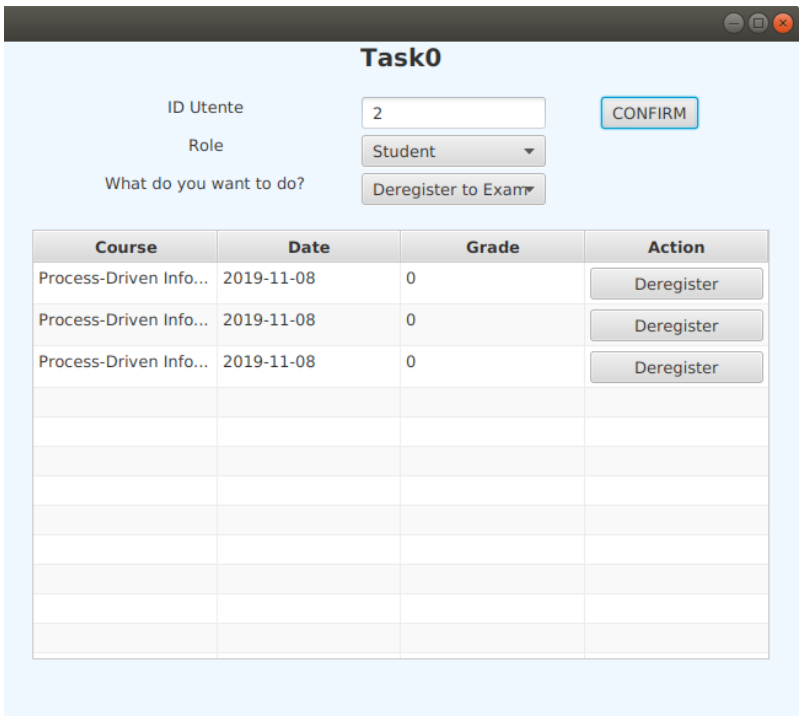

Figure 11: Example of Deregister to Exam

- *Course*, the name of the course of the exam the Student is enrolled to;
- *Date*, the date of the exam;
- *Deregister* button that the Stuedent has to push in order to do the deregistration.

If the Professor pushes one of the *Deregister* the table is updated so that it shows all the exams to which the Student is not enrolled.

See Grades

If the Student select the *See Grades* option, the table, fig. 12, is populated with a list of all the exams the student has done. Each element of the list is composed of:

- *Course*, the name of the course;
- *Date*, the date when the student passed the exam;
- *Grade*, the grade the Student got.

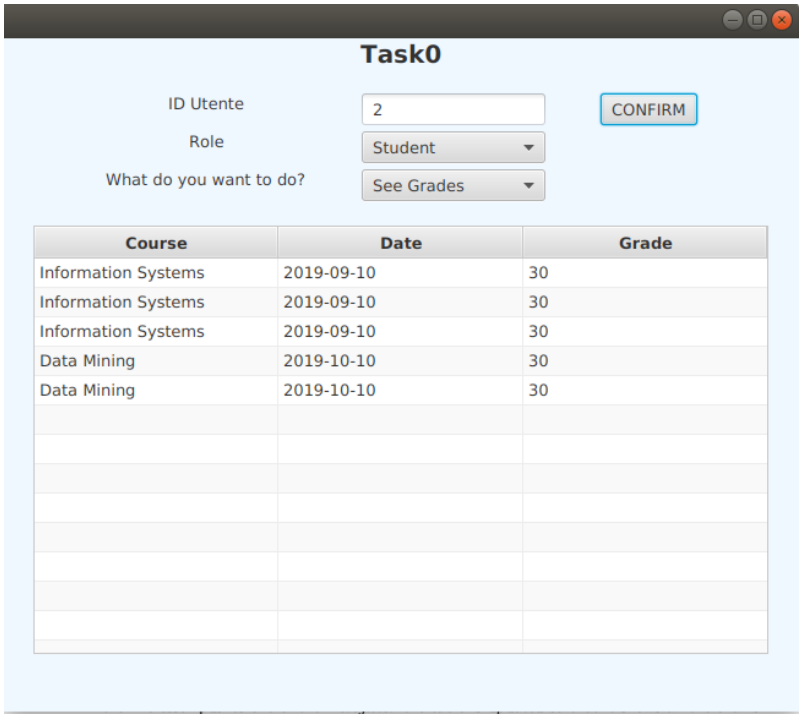

Figure 12: Example of See Grades