**Description**

**College Examination Management System**

**In this project, we will discuss what this project is, how it works, what are its components, and a description of the GUI used with some pictures.**

**First of all, what is this project?**

**This project is designed as a system for a college or educational institution such as the College of Computing to give the user good control over his account, as there are three types of users for this project, and we will discuss each of them in some detail.**

**The first type of user for this project is the admin, and his job is to connect to the accounts of other users, such as the student and the teacher. His tasks are summarized in the following:**

* 1. 1. Administrative Module
  2. a) Admin has username and password and can alter them.
  3. b) Admin manages students and lecturers (Add, Delete, update, List, search).
  4. c) Admin manages Subjects and assign them for students and lecturers.

In the beginning, I designed a form for logging in, where this form is called )LogIn), and if it is the first time, then it is necessary to log in to a form called (LogUp( to register for the first time the user sends his data consisting of three information: username, password, and user type, i.e. admin or student Or a teacher sends this data to a table in the Database called (users). Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Then after registering using an email to the admin, the next page is opened to show him three options, which are File, from which he can refer to Log in, Eidt and Subject

A picture containing text, computer

Description automatically generated

Graphical user interface, application

Description automatically generated

From Edit, you can enter to add, view or edit a student or teacher

Graphical user interface

Description automatically generatedTable

Description automatically generatedGraphical user interface

Description automatically generatedChart

Description automatically generatedGraphical user interface

Description automatically generated

When you choose to add a teacher, you can add a teacher, and when you choose List Lecturer, you can view your teacher’s teacher, and when you choose Update Lecture, you can update the teacher’s data.

The same thing happens when you add a student.

Our second part is

* 1. **2. Lecturer Module**
  2. **a) Lecturer manages and designs his exams (Add, Delete, update, List)**
  3. **b) Lecturer puts the exam of subject and its duration and true answer for some questions like single and multiple choice.**
  4. **c) System calculate the final degree for each student.**
  5. **d) Lecturer can make reports about students and their degrees.**

In this form, the teacher can manage exams, such as adding an exam, clearing an exam, updating exam data, or displaying exams that he created, in addition, because he can also have a new exam for a specific period and time, and the correct answer to the questions of this exam. In addition, the teacher also puts a report on the student’s grades or on the name The student and his grade. Graphical user interface

Description automatically generated

In addition, the teacher can modify his data by clicking on Update Information Account.

The third model, which is the student form, where the student can enter the exam for a registered subject only.

The student can enter the exam once, in addition to seeing his grades and the correct answer to the exam.

Graphical user interface, bar chart

Description automatically generated

Graphical user interface, text

Description automatically generated

* 1. **3. Student Module**
  2. **a) Student can access only exams for his registered subjects.**
  3. **b) Student can enter exam for once.**
  4. **c) Student can see his degree and corrected exam.**

**The fourth model and the users model. In this model, users can enter and exit the project, in addition to the fact that they can modify their data except ID.**

* 1. **4. User Module**
  2. **a) All Users can login and logout.**
  3. **b) Users can Update their Information except ID.**

**Design Pattern**

Best practice to solve common software problems

Solutions in the form of templates that may be applied to real world problems

***(SOLID)***

S è Single responsibility principle:

The class should solve only one problem it should have a single reason to change

Oè open/closed principle:

The class should be open for extension,

closed for modification

Lè liskov substitution principle:

If you substitute any type with one of its subtype, the behavior should not change

Iè interface segregation principle:

Avoid making general interface contains all methods

Dè dependency inversion principle:

Higher lever classes should not know

the implementation of low classes but depends on abstraction

# 

# (1) Single Responsibility Principle (SRP)

is a computer-programming principle that states that every [module](https://en.wikipedia.org/wiki/Modular_programming" \o "Modular programming), [class](https://en.wikipedia.org/wiki/Class_(computer_programming)" \o "Class (computer programming)) or [function](https://en.wikipedia.org/wiki/Function_(computer_programming)" \o "Function (computer programming)) in a [computer program](https://en.wikipedia.org/wiki/Computer_program" \o "Computer program) should have responsibility over a single part of that program's [functionality](https://en.wikipedia.org/wiki/Software_feature" \o "Software feature), and it should [encapsulate](https://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming)" \o "Encapsulation (object-oriented programming)) that part. All of that module, class or function's [services](https://en.wikipedia.org/wiki/Service_(systems_architecture)" \o "Service (systems architecture)) should be narrowly aligned with that responsibility.

[Robert C. Martin](https://en.wikipedia.org/wiki/Robert_C._Martin" \o "Robert C. Martin), the originator of the term, expresses the principle as, "A class should have only one reason to change, although, because of confusion around the word "reason" he also stated "This principle is about people. In some of his talks, he also argues that the principle is, in particular, about roles or actors. For example, while they might be the same person, the role of an accountant is different from a database administrator. Hence, each module should be responsible for each role.

And then, after reflecting on this principle, we have applied it to our project, which is in front of your eyes. On this class :

# (2) Open Closed Principle OCP (OCP)

In [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming" \o "Object-oriented programming), the **open–closed principle** states "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification* that is, such an entity can allow its behavior to be extended without modifying its [source code](https://en.wikipedia.org/wiki/Source_code" \o "Source code).

The name *open–closed principle* has been used in two ways. Both ways use generalizations (for instance, [inheritance](https://en.wikipedia.org/wiki/Inheritance_(computer_science)" \o "Inheritance (computer science)) or delegate functions) to resolve the apparent dilemma, but the goals, techniques, and results are different.

And then, after reflecting on this principle, we have applied it to our project, which is in front of your eyes. On this class :

# (3) Liskov Substitution Principle (LSP)

is a particular definition of a subtyping relation, called [strong behavioral subtyping](https://en.wikipedia.org/wiki/Behavioral_subtyping" \o "Behavioral subtyping), that was initially introduced by [Barbara Liskov](https://en.wikipedia.org/wiki/Barbara_Liskov" \o "Barbara Liskov) in a 1988 conference [keynote](https://en.wikipedia.org/wiki/Keynote" \o "Keynote) address titled *Data abstraction and hierarchy*. It is based on the concept of "substitutability" – a principle in [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming" \o "Object-oriented programming) stating that an [object](https://en.wikipedia.org/wiki/Object_(computer_science)" \o "Object (computer science)) (such as a [class](https://en.wikipedia.org/wiki/Class_(computer_programming)" \o "Class (computer programming))) and a sub-object (such as a class that extends the first class) must be interchangeable without breaking the program. It is a [semantic](https://en.wikipedia.org/wiki/Formal_semantics_of_programming_languages" \o "Formal semantics of programming languages) rather than merely syntactic relation, because it intends to guarantee semantic interoperability of [types](https://en.wikipedia.org/wiki/Data_type" \o "Data type) in a hierarchy, object types in particular. Barbara Liskov and [Jeannette Wing](https://en.wikipedia.org/wiki/Jeannette_Wing" \o "Jeannette Wing) described the principle succinctly in a 1994 paper

And then, after reflecting on this principle, we have applied it to our project, which is in front of your eyes. On this class :

# (4) Interface Segregation Principle ISP

In the field of [software engineering](https://en.wikipedia.org/wiki/Software_engineering" \o "Software engineering), the **interface segregation principle** (**ISP**) states that no code should be forced to depend on [methods](https://en.wikipedia.org/wiki/Method_(computer_programming)" \o "Method (computer programming)) it does not use. ISP splits [interfaces](https://en.wikipedia.org/wiki/Interface_(computing)" \o "Interface (computing)) that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called *role interface*s. ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy. ISP is one of the five [SOLID](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)" \o "SOLID (object-oriented design)) principles of object-oriented design, similar to the High Cohesion Principle of [GRASP](https://en.wikipedia.org/wiki/GRASP_(object-oriented_design)" \o "GRASP (object-oriented design)). Beyond object-oriented design, ISP is also a key principle in the design of distributed systems in general and microservices in particular. ISP is one of the six IDEALS principles for microservice design.

And then, after reflecting on this principle, we have applied it to our project, which is in front of your eyes. On this class :

# (5) Dependency Inversion Principle (DIP)

In [object-oriented design](https://en.wikipedia.org/wiki/Object-oriented_design" \o "Object-oriented design), the **dependency inversion principle** is a specific methodology for [loosely coupling](https://en.wikipedia.org/wiki/Coupling_(computer_programming)" \o "Coupling (computer programming)) software [modules](https://en.wikipedia.org/wiki/Modular_programming" \o "Modular programming). When following this principle, the conventional [dependency](https://en.wikipedia.org/wiki/Dependency_(computer_science)" \o "Dependency (computer science)) relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details. The principle states:

1. High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).
2. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

By dictating that *both* high-level and low-level objects must depend on the same abstraction, this design principle *inverts* the way some people may think about object-oriented programming.

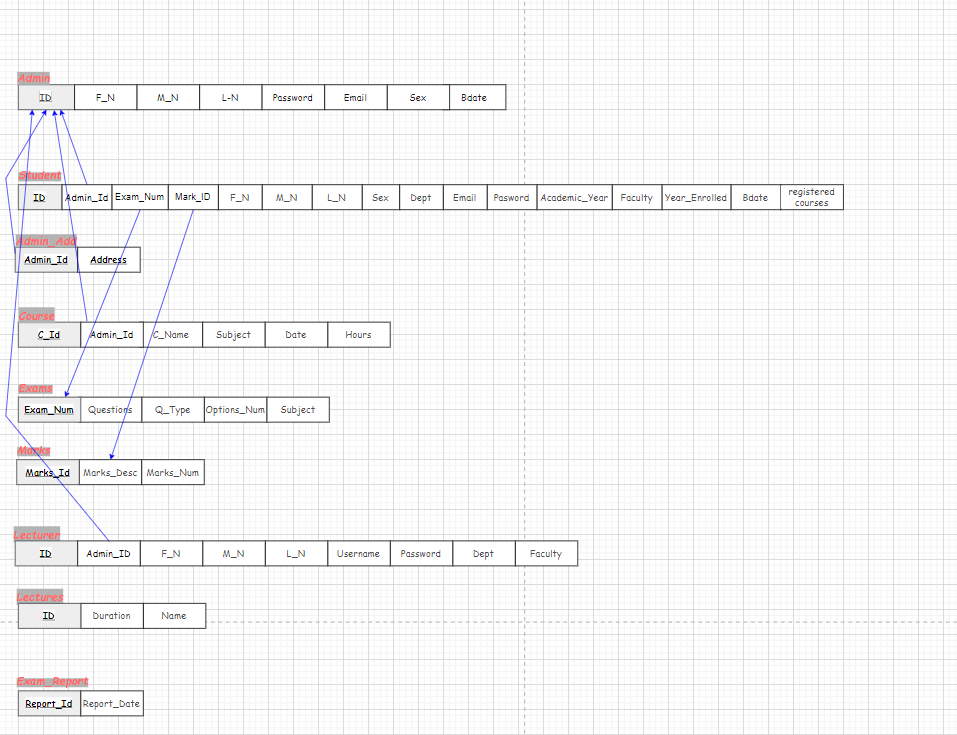
The idea behind points A and B of this principle is that when designing the interaction between a high-level module and a low-level one, the interaction should be thought of as an abstract interaction between them. This not only has implications on the design of the high-level module, but also on the low-level one: the low-level one should be designed with the interaction in mind and it may be necessary to change its usage interface.

In many cases, thinking about the interaction in itself as an abstract concept allows the coupling of the components to be reduced without introducing additional coding patterns, allowing only a lighter and less implementation-dependent interaction schema.

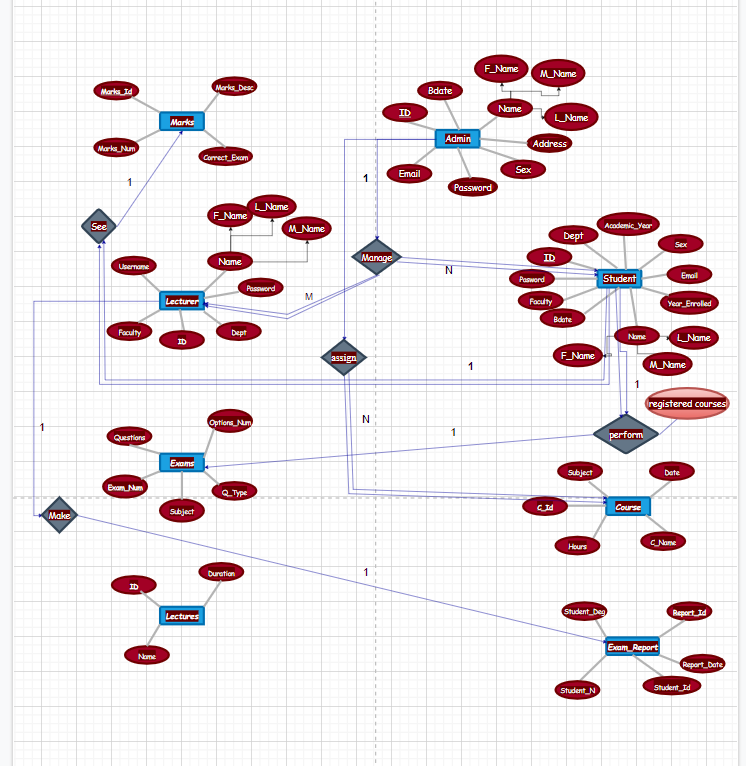
When the discovered abstract interaction schema(s) between two modules is/are generic and generalization makes sense, this design principle also leads to the following dependency inversion coding pattern.

And then, after reflecting on this principle, we have applied it to our project, which is in front of your eyes. On this class :

Database Schema



ER Diagram



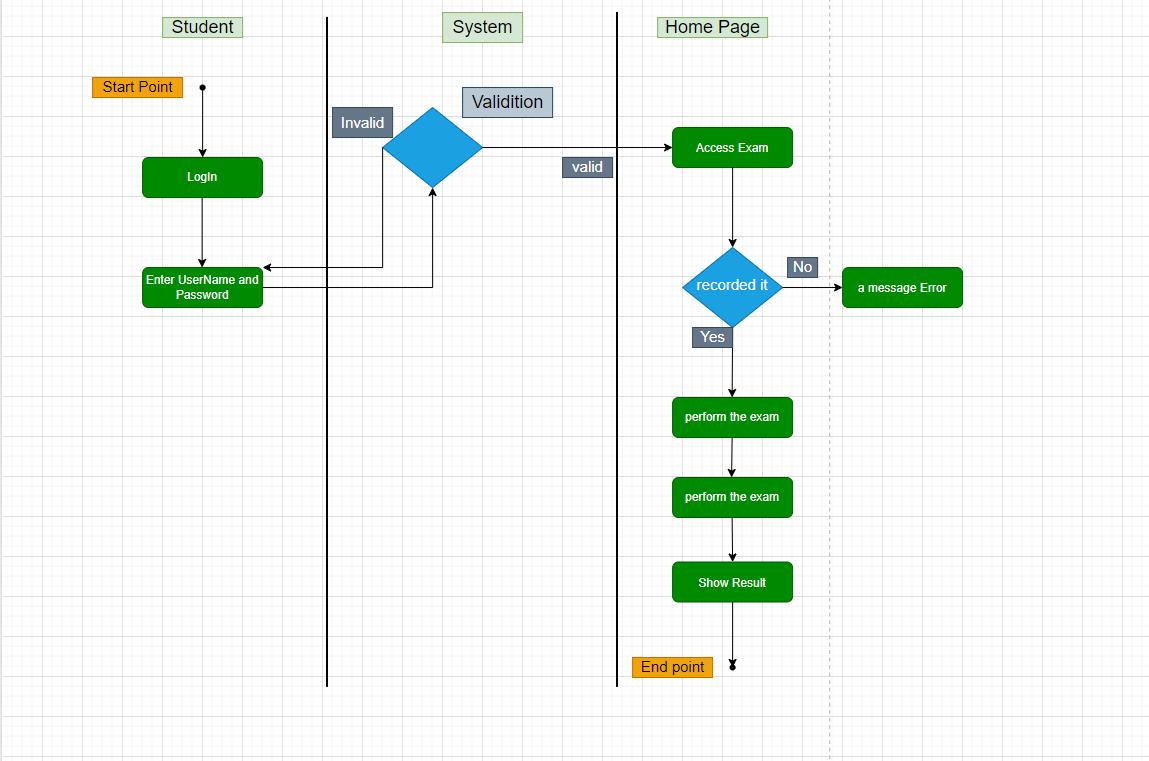
Activity Diagram

Admin

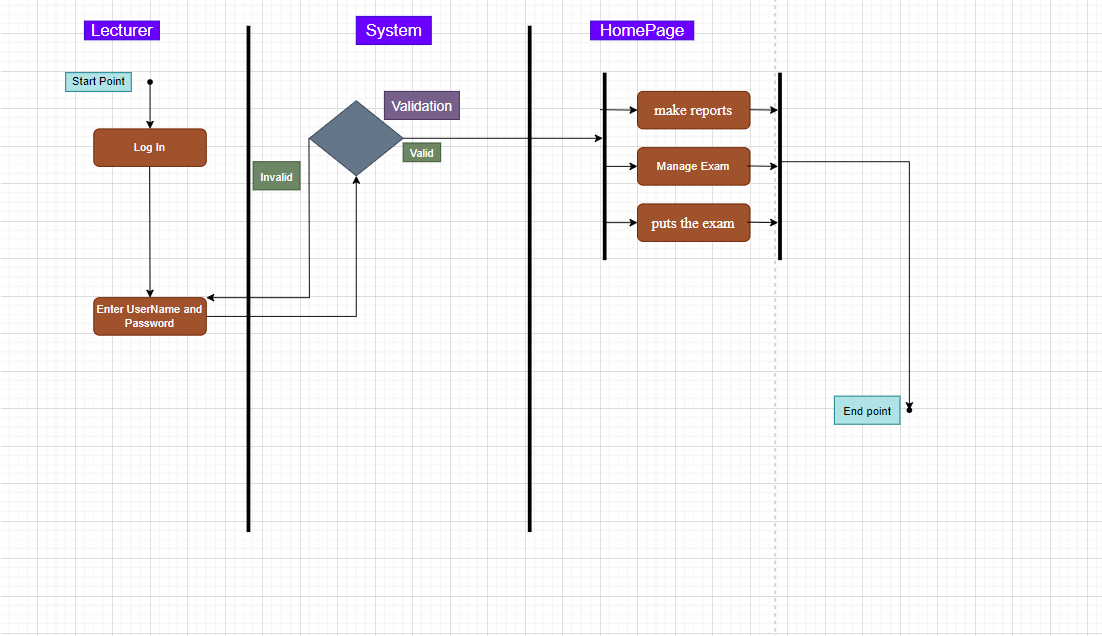
Diagram, engineering drawing

Description automatically generated

Student



Lecturer

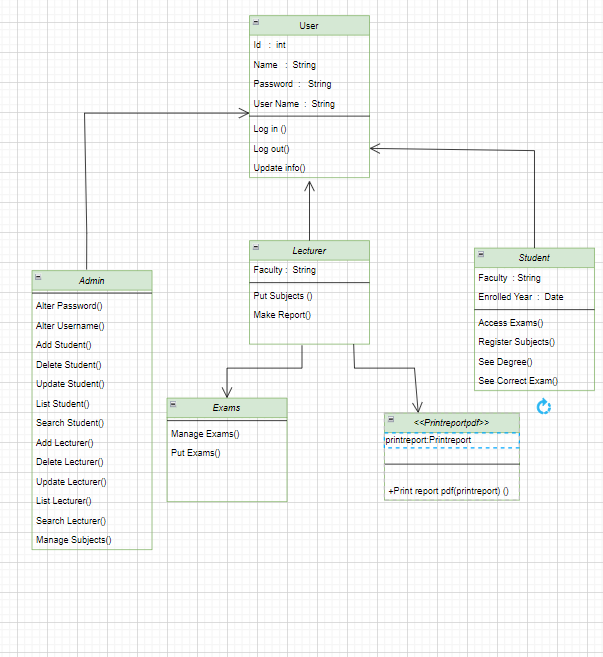


Sequence Diagram

Chart

Description automatically generated

Class Diagram



Use Case Diagram

