

HOMework 6

Buzea Vlad-Calin

30422

Table of Contents

1. Objective of the assignment.....	3
2. Problem analysis.....	3
2.1. Modelling.....	3
2.2. Use cases	5
2.3. Scenarios.....	5
3. Design.....	8
3.1. UML Class Diagram	8
3.2. Data structures	10
3.3. Algorithms.....	12
3.4. Class design	12
3.5. Interface	18
4. Implementation and testing.....	26
5. Conclusions, what has been learned, further developments	30
6. Bibliography.....	30

1. Objective of the assignment

Homework number 6 has as target the development of an application that implements two design patterns and connects to a Database. Due to the fact that a certain domain has not been specified, I have chosen to implement an application that supports a school's grading system. Today, there is a great tendency to move the written school organization systems into a common access digital system. This process is held back due to the relatively reduced access of ordinary people to such digital systems (Computer, Tablet, Smartphone etc.). By access I also refer to the incapability of utilizing digital systems. However, it is clear to everyone that in about a couple of decades this will not be an issue anymore. Thus, digitalization of written systems will be a great trend in the future. Experience in this domain will bring great advantages to developers. The main purpose of the assignment is to create an application that allows teachers to grade students that follow their courses and the students to see these grades in real time. Deletion of mistaken grades should also be possible.

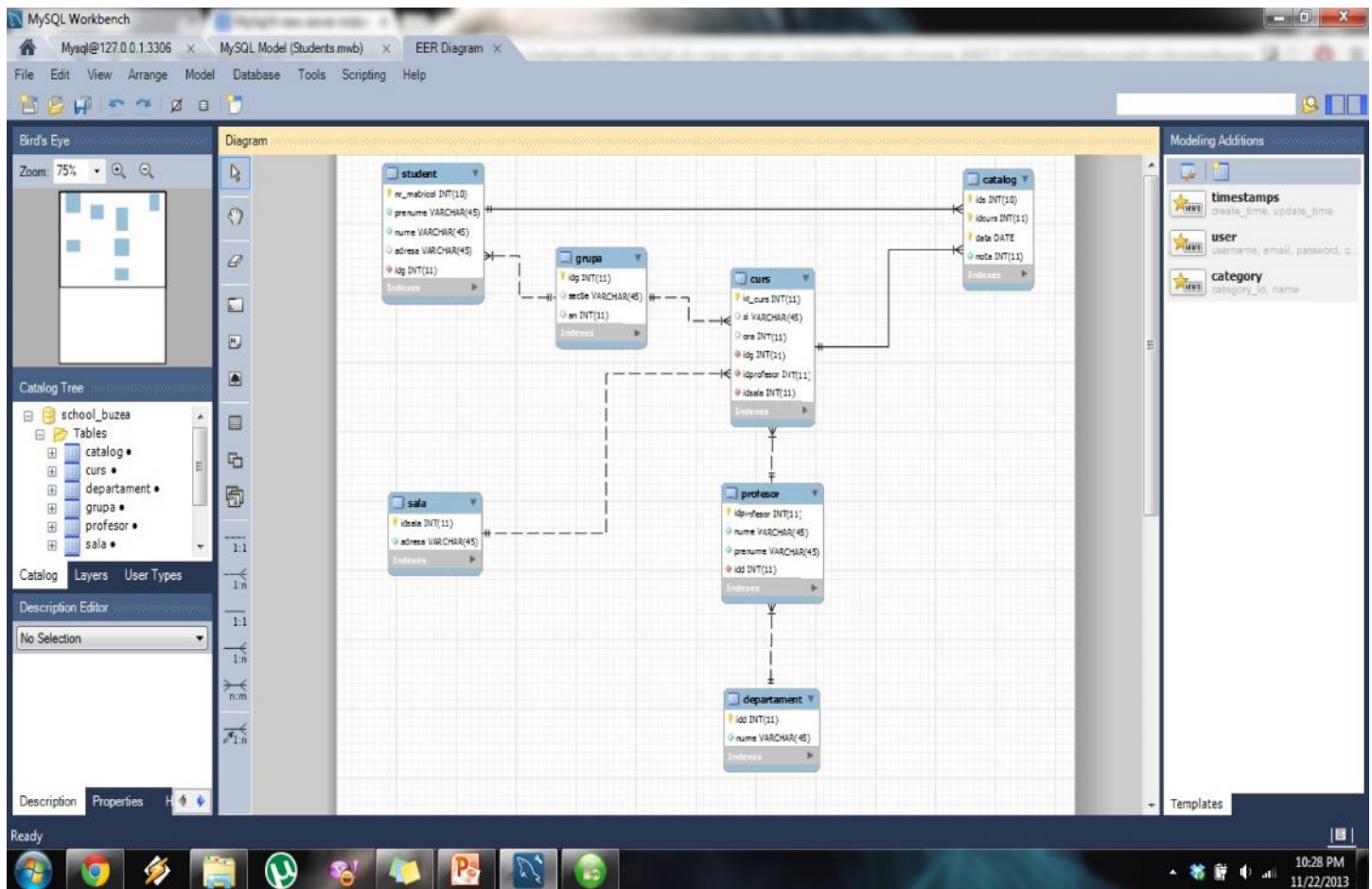
2. Problem analysis

A closer look at the problem shows the following aspects that concern us:

- We must store Students, Teachers, Grades and Courses
- Courses are composed of Group of students that auditions a Professor in a Room
- All data must be stored, such that a restart of the application will not involve loss of data
- Only Teachers must be able to give grades
- Students can see only their own grades (personal information)
- We must solve the concurrent access issue
- Grades should be displayed in real time

2.1 Modeling

To gain a small advantage I have chosen to reuse the database model built in the first semester of this year at the Database Design course. By connecting my application to the database the data storage issue will be solved. Also, minimum integrity constraints will be assured. So, I will model the application to wrap around the database model. A picture of the initial database will be further presented, mentioning the fact that it suffered small alterations during the programs development, due to applicability reasons. A first observation to be made is that we need an "Accounts" table to store accounts for login. The second observation is that the database is modeled in Romanian.



The database schema

A highly recommended practice when using databases in Java applications is to apply the Singleton design pattern. Thus we can check one design pattern off our list, leaving further discussions for the Implementation section.

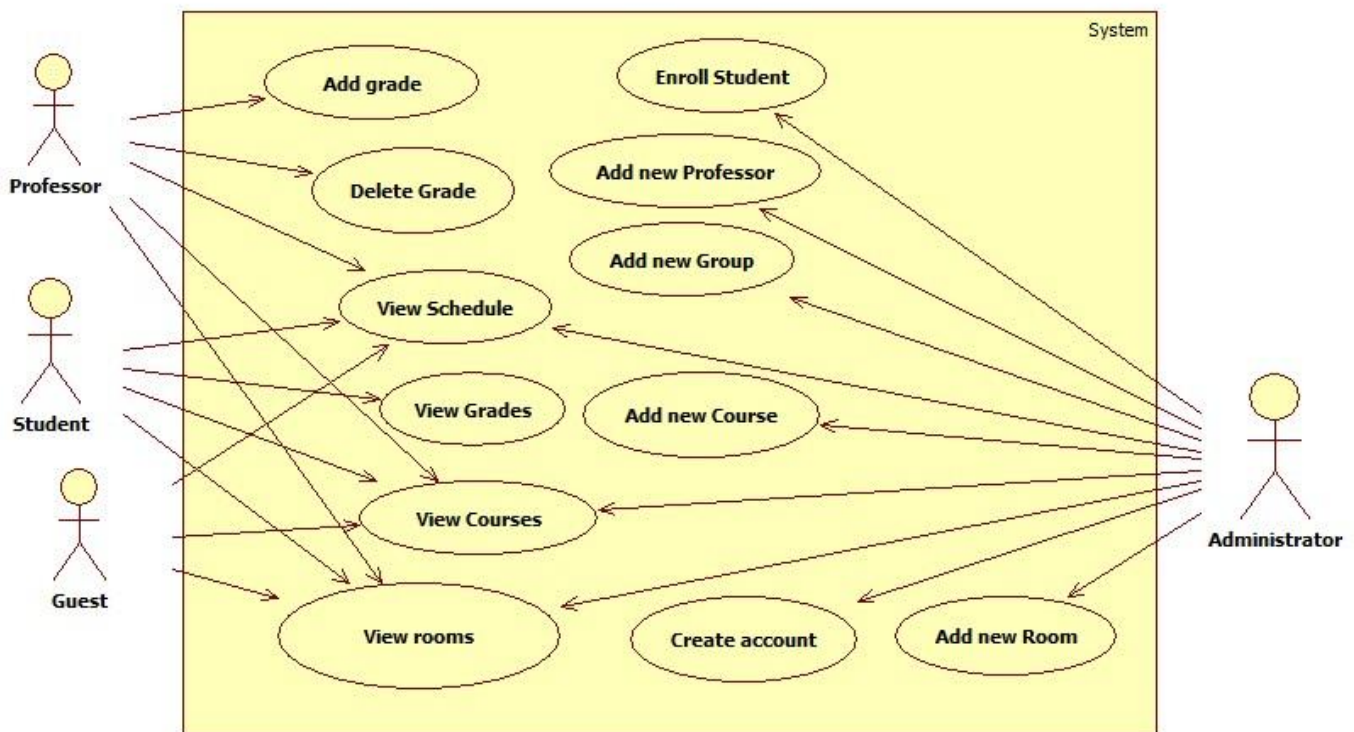
In the preliminary phase, the application will be run on only one device, but the main purpose will be to run the same application on more devices and platforms simultaneously. In this way the professors can transmit information through the application in real time, using the database that will be stored online.

It becomes more and more clear that there should be two main sections in the application: the Professor section and the Student section. Also, common information, such as Course Schedule and room addresses, can be shared with both categories and even the general interested public. Not so obvious, initially, is the administration section. To greater illustrate the main functionalities of the system, we must shallowly implement administration functions such as the addition of new students, courses, teachers, groups, rooms and especially user accounts.

2.2 Use cases

A use case defines the interactions between external actors and the system under consideration to accomplish a goal. Actors must be able to make decisions. According to this definition we can identify the main actor to be the user of the application.

A use case represents specifying a sequence of actions that the system can perform by interacting with the system's actors. The designed "Catalog" system has four actors: the professor, the student, the administrator and the guest. For a better understanding of the ideas stated above, I have created a use case diagram which describes the relationships between the actors and the system, as well as specifies the use cases.



2.3 Scenarios

A use case must have a clearly identifiable beginning and ending. We must also specify possible variants, such as successful scenario or alternative variants. A scenario represents a particular succession of sequences, which is being run from the beginning until the end of a use case.

Moving on, we will discuss in detail two separate use cases, taking into consideration the interaction of the professor with the system and then the one of the student.

Scenario I

a) Identification summary

Title: Professor grades a student

Summary: This use case describes the steps a professor must follow in order to give a grade to a student

Actor: professor

b) Flow of events

Preconditions: The user interface did not malfunction and the user input data was correct.

Main success scenario:

1. The professor starts the Catalog application.
2. The professor clicks the „Professor” button.
3. The application opens a new frame
4. The application requests a username and a password for authentication.
5. The professor introduces the username and password corresponding to his account.
6. The professor presses the „Login” button.
7. The application verifies correctness of input.
8. The application offers the specific interface .
9. The professor selects a specific course from a list.
10. The system generates a second list with students following the selected course.
11. The professor selects the student to be graded.
12. The professor introduces the date of the grade, respecting the „yyyy-MM-dd” format.
13. The professor introduces the mark from 1 to 10 using the spinner.
14. The professor clicks the „Submit” button.
15. The system validates the input and registers the grade.

Alternative scenarios:

A1) The input password or/and username is incorrect or there is no input: step 7 kicks in

In this case, the application informs the user of the incorrectness of the input data by showing a message:



If this happens, the scenario returns to step 4.

Error sequences:

E1) While performing an operation, the application somehow freezes (for example it enters an infinite loop): the user will have to restart the application in this case.

Post-conditions: none.

Scenario II

a) Identification summary

Title: Students views grades

Summary: This use case describes the steps a student must follow in order view his personal grades

Actor: student

b) Flow of events

Preconditions: The user interface did not malfunction and the user input data was correct.

Main success scenario:

1. The student starts the Catalog application.
2. The student clicks the „Student” button.
3. The application opens a new frame
4. The application requests a username and a password for authentication.
5. The student introduces the username and password corresponding to his account.
6. The student presses the „Login” button.
7. The application verifies correctness of input.
8. The application offers the specific interface .
9. The student can now view his grades and courses he is enrolled in.

Alternative scenarios:

A1) The input password or/and username is incorrect or there is no input: step 7 kicks in

In this case, the application informs the user of the incorrectness of the input data by showing a message:



If this happens, the scenario returns to step 4.

Error sequences:

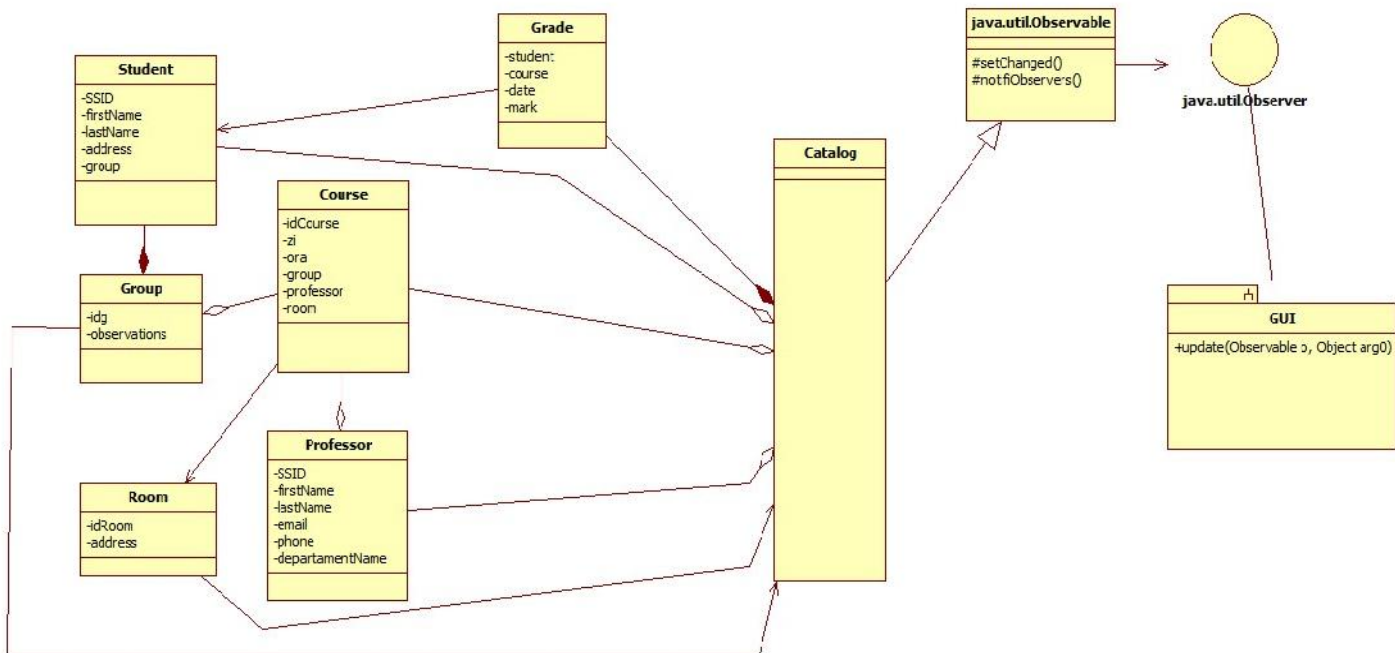
E1) While performing an operation, the application somehow freezes (for example it enters an infinite loop): the user will have to restart the application in this case.

Post-conditions: none.

3. Design

3.1 UML Class Diagram

For better understanding of the project, firstly we will present the contracted UML Class Diagram, that better illustrates the relationship between entities. To reduce complexity, the entire set of graphical user entities (JFrames, JPanels, JTables, etc.) will be grouped under the GUI entity, thus making the following diagram one of conceptual level.



A few small observations regarding the contracted UML Class Diagram are: only some classes in the GUI subsystem implement the Observer interface, the line connecting the GUI to the interface represents implementation and not all attributes are represented in this diagram. It is also clear that the UML wraps the database ERD.

Further on, we will present the extended Class Diagram, mentioning the fact that relationships are only represented as associations and dependencies, but the real relationship between entities is the one illustrated above.



3.2 Data Structures

The data structures used in this application are implemented by the Java Collections Framework and are the Hash Maps and Array Lists.

public class HashMap<K,V>

extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

An instance of HashMap has two parameters that affect its performance: initial capacity and load factor. The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table.

Note that this implementation is not synchronized. If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the Collections.synchronizedMap method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

```
Map m = Collections.synchronizedMap(new HashMap(...));
```

The iterators returned by all of this class's "collection view methods" are fail-fast: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

public class ArrayList<E>

extends AbstractList<E>

implements List<E>, RandomAccess, Cloneable, Serializable

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.) The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation.

Note that this implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the Collections.synchronizedList method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
List list = Collections.synchronizedList(new ArrayList(...));
```

The iterators returned by this class's iterator and listIterator methods are fail-fast: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

3.3 Algorithms

The Algorithms used in this project are those corresponding to hashtables:

- Hashtable insert : compute value of hash function(corresponding to transmitted key) and insert value at the end of the linked list corresponding to that table entry.
- Hashtable retrieve: compute value of hash function(corresponding to transmitted key) and search the value in linked list corresponding to that table entry
- Hashtable delete: compute value of hash function(corresponding to transmitted key) and search the value in linked list corresponding to that table entry; if found, delete it from the linked list.

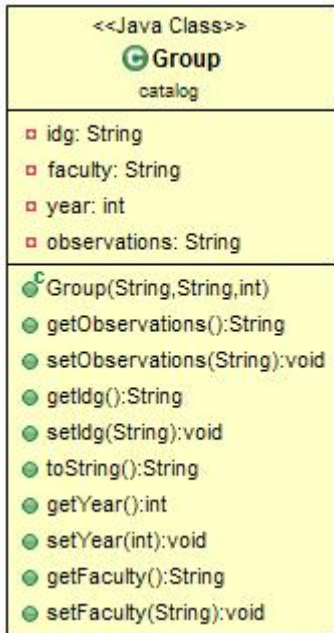
3.4 Class Design

CRC CARDS

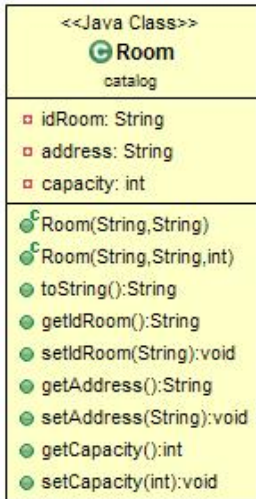
Student	
Attributes, Functions:	Collaborator classes
Has SSID as unique identifier. Has Name and address Belongs to a Group	Group



Group	
Attributes, Functions:	Collaborator classes
References a real word group that must have students. Has a year of study and a specialty. May have attached observations, to prevent professors.	Student



Room	
Attributes, Functions:	Collaborator classes
Has name, called also ID Has address Has capacity	

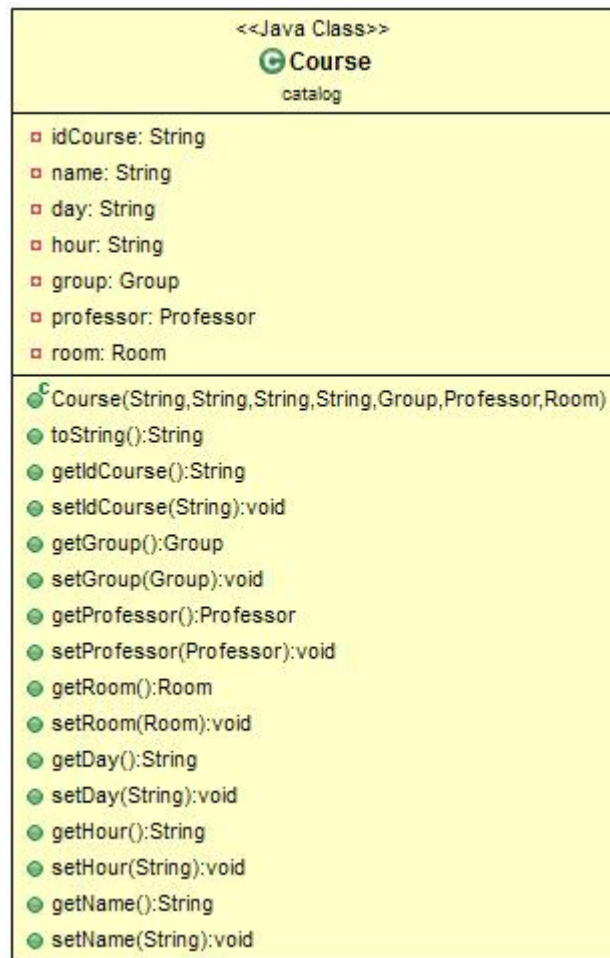


Professor

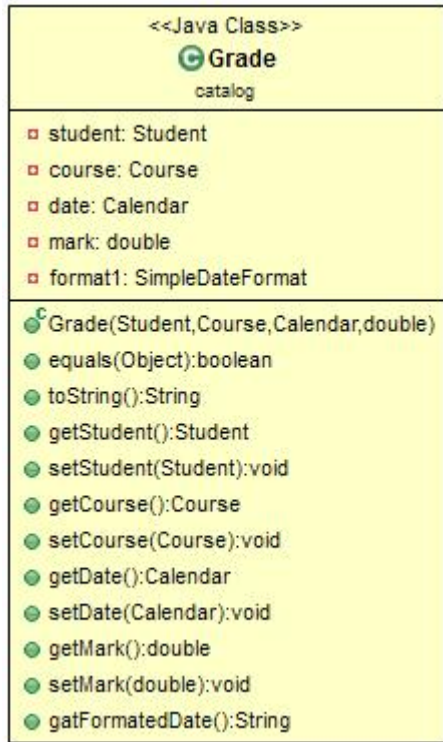
Attributes, Functions:	Collaborator classes
Has SSID as unique identifier Has Name Has contact information: Email, phone Belongs to a departament	



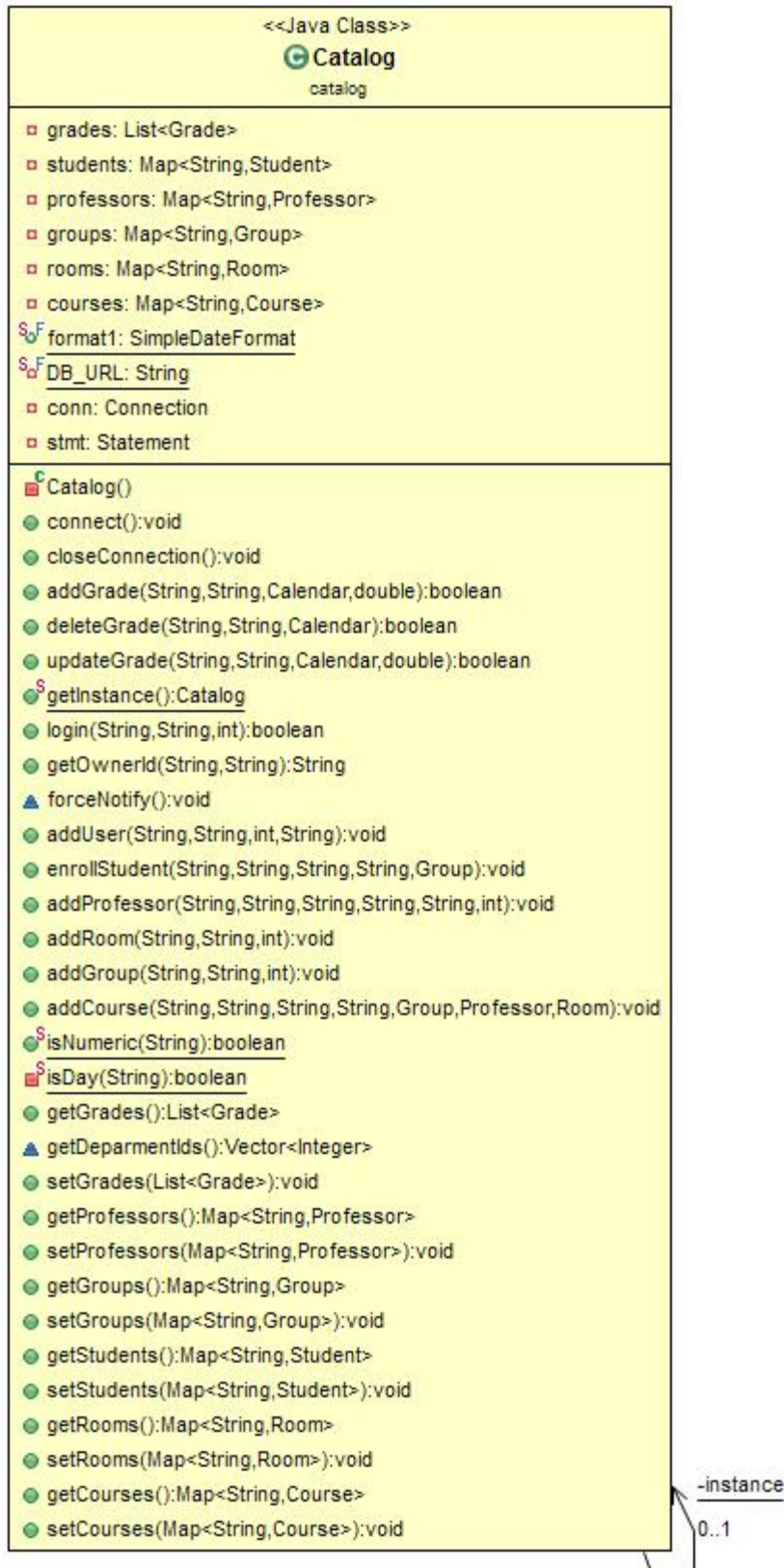
Course	
Attributes, Functions:	Collaborator classes
Represents a group of students that studies a subject taught by a single professor. Is held in a room. Is held once a week at a specific hour.	Group Professor Room



Grade	
Attributes, Functions:	Collaborator classes
Associates a Student with a mark received at a Course Has a date	Student Course



Catalog	
Attributes, Functions:	Collaborator classes
Wraps all previous classes and maps necessary functions. Connects to the database Adds grades, students, professors, rooms, courses and groups Deletes Grades Provides data for GUI	Grade Student Course Professor Group Room

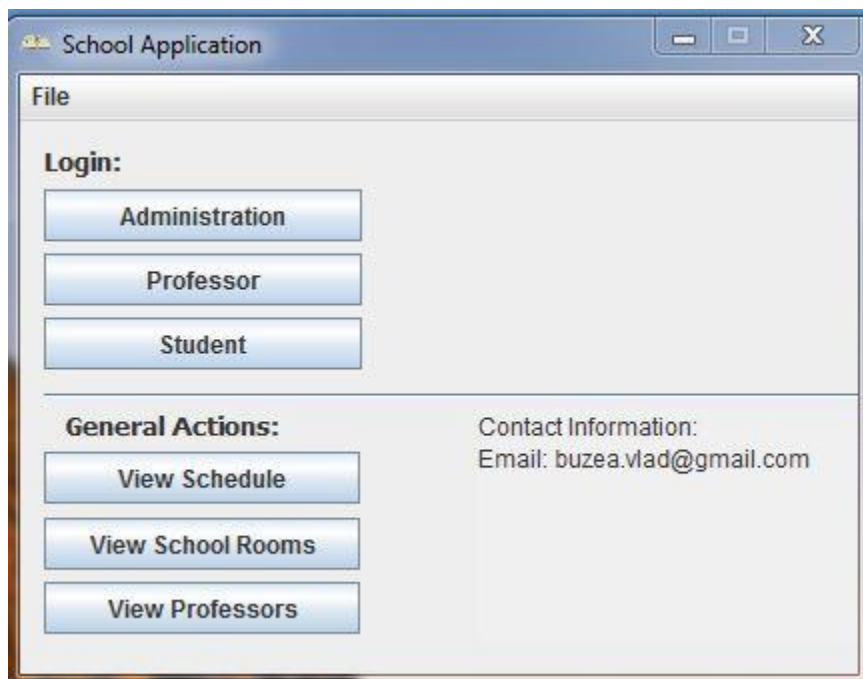


The remaining classes are GUI classes and have very specific functionality, so they will be presented with bullets:

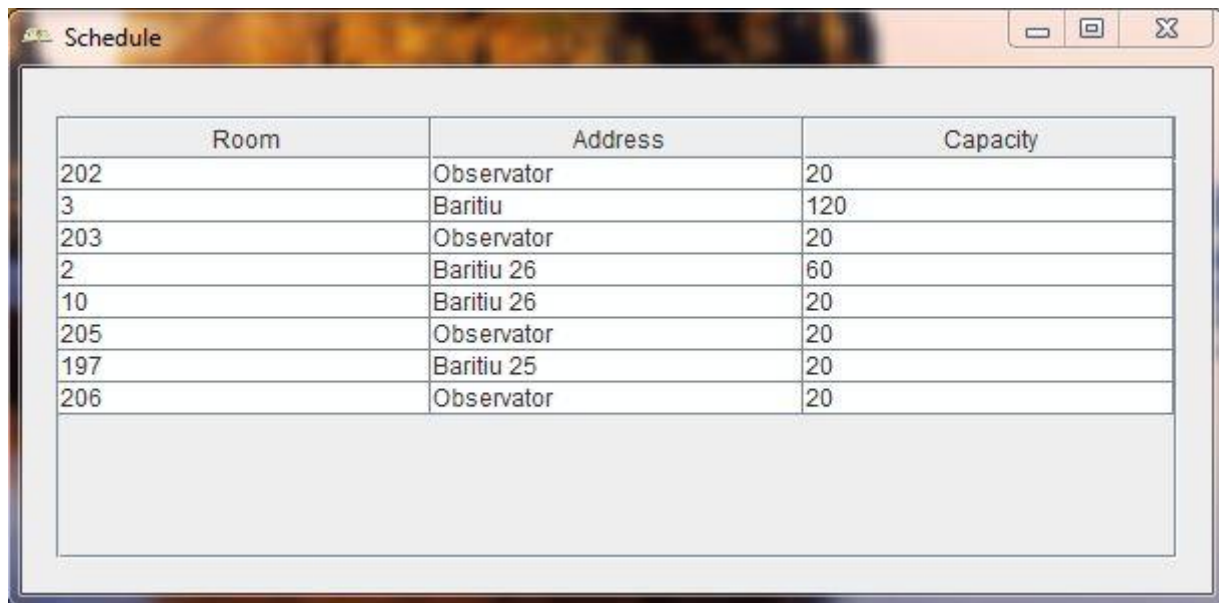
- AccountAddPanel – panel that offers content for registering a new account
- AdminFrame – container for LoginPanel and AdminPanel
- AdminPanel – admin menu options
- ApplicationWindow – the root of the application, contains links to all other GUI classes
- CourseAddPanel - panel that offers content for registering a new course
- GroupAddPanel - panel that offers content for registering a new group
- LoginPanel – used for account login; if valid login, moves CardLayout to “Menu” card;
- ProfessorAddPanel - panel that offers content for registering a new professor
- ProfessorMenuPanel – professor menu options: Add Grade, Delete Grade and View Courses
- ProfessorWindow – container for LoginPanel and ProfessormenuPanel
- RoomAddPanel - panel that offers content for registering a new room
- ScheduleFrame – show a table with courses or rooms
- StudentEnrollPanel - panel that offers content for registering a new student
- StudentFrame – container for LoginPanel and StudentPanel
- StudentPanel – student menu options: view grades and courses
- TableFrame – a JFrame that contains a table who’s content is specified in the constructor

3.5 Interface

We will start by presenting the ApplicationWindow, the root of the program. When this window is closed, the entire process is stopped.

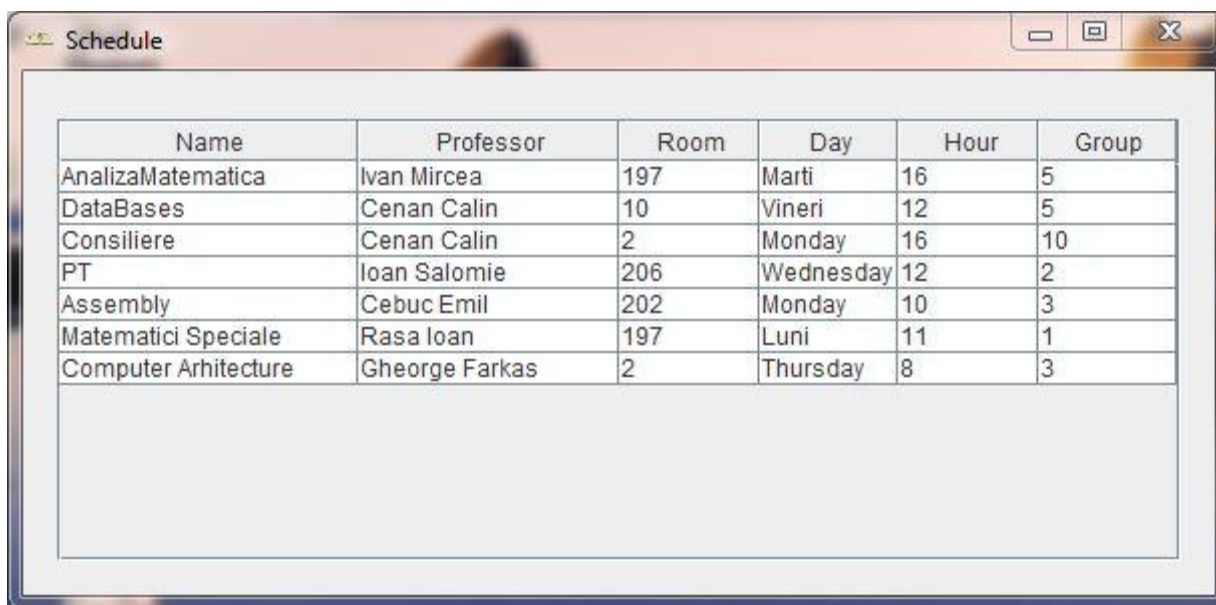


In the upper side we see the Login Buttons specific to each account type, and at the bottom side we see the general purpose commands. These general purpose commands will be next presented.



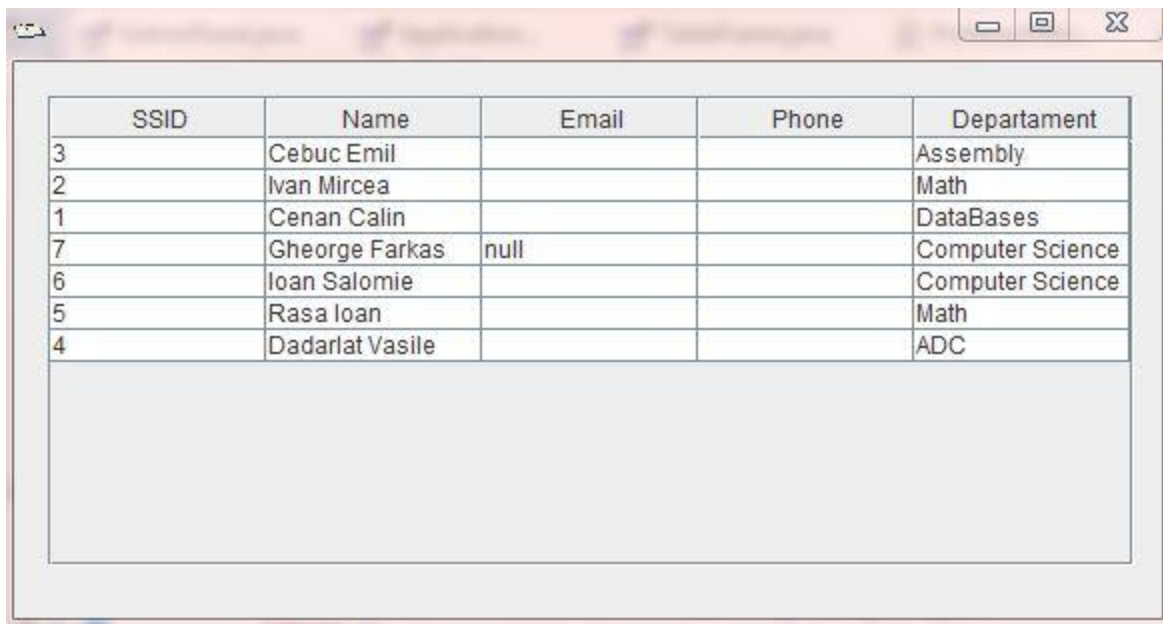
Room	Address	Capacity
202	Observator	20
3	Baritiu	120
203	Observator	20
2	Baritiu 26	60
10	Baritiu 26	20
205	Observator	20
197	Baritiu 25	20
206	Observator	20

Rooms table



Name	Professor	Room	Day	Hour	Group
AnalizaMatematica	Ivan Mircea	197	Marti	16	5
DataBases	Cenan Calin	10	Vineri	12	5
Consiliere	Cenan Calin	2	Monday	16	10
PT	Ioan Salomie	206	Wednesday	12	2
Assembly	Cebuc Emil	202	Monday	10	3
Matematici Speciale	Rasa Ioan	197	Luni	11	1
Computer Arhitecture	Gheorge Farkas	2	Thursday	8	3

Course Schedule

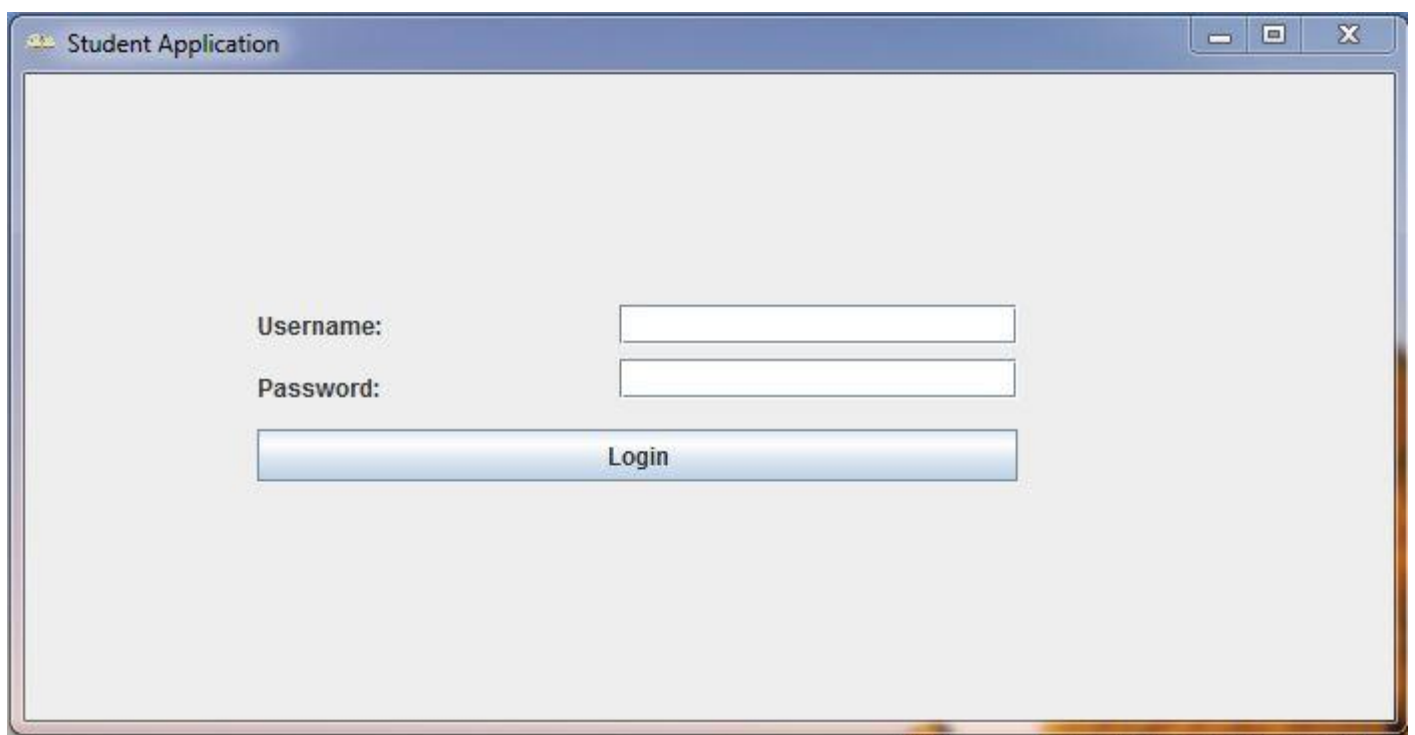


A screenshot of a window displaying a table of professor information. The table has five columns: SSID, Name, Email, Phone, and Departament. The rows are numbered 3, 2, 1, 7, 6, 5, and 4 from top to bottom. The 'Email' column contains a 'null' value for the row with SSID 7. Below the table is a large empty rectangular area.

SSID	Name	Email	Phone	Departament
3	Cebuc Emil			Assembly
2	Ivan Mircea			Math
1	Cenan Calin			DataBases
7	Gheorge Farkas	null		Computer Science
6	Ioan Salomie			Computer Science
5	Rasa Ioan			Math
4	Dadarlat Vasile			ADC

Professor Information

Next up, we will take a look at the LoginPanel:



A screenshot of a window titled 'Student Application'. Inside the window is a login panel with the following elements: a 'Username:' label followed by a text input field, a 'Password:' label followed by a text input field, and a 'Login' button below them.

LoginPanel

After passing the LoginPanel the Student, Professor and Administrator have different menus:

Student Application

Load Tables **Student: Vlad Buzea Group: 5**

My Grades:

Course	Grade	Date
DataBases	9.0	2013-11-20
DataBases	10.0	2014-05-06
AnalizaMat...	10.0	2014-02-09

My Courses:

Name	Professor	Room	Day	Hour	Group
AnalizaMa...	Ivan Mircea	197	Marti	16	5
DataBases	Cenan Ca...	10	Vineri	12	5

Student Menu

Professor Application

Welcome Cenan Calin!

Add new Grade:

Course: **DataBases 10 5 Vineri:12** ▼

Student: **Raul Jantea Group: 5** ▼

Date:

Date format : "yyyy-MM-dd"

Mark: 5

Submit **Delete Grade**

My Courses:

Name	Professor	Room	Day	Hour	Group
DataBases	Cenan Calin	10	Vineri	12	5
Consiliere	Cenan Calin	2	Monday	16	10

Professor Menu



Administrator Menu.

The Administrator menu has 6 different subsections, each one corresponding to a use case:

A screenshot of the "Admin Application" window, now displaying the "Add New Student" form. The "Administration actions:" menu is still visible on the left. To the right of the menu, there are input fields for "SSID:", "First Name:", "Last Name:", and "Address:". Below these is a "Group:" dropdown menu currently showing "3 AC 1". At the bottom of the form are "Submit" and "Cancel" buttons. The window title bar and standard controls are also visible.

Add New Student

The screenshot shows a window titled "Admin Application" with a standard Windows-style title bar (minimize, maximize, close buttons). Inside the window, on the left, is a vertical list of buttons under the heading "Administration actions:". The buttons are: "Enroll Student", "Add new Professor" (which is highlighted with a blue border), "Add new Group", "Add new Course", "Create new Account", and "Add new Room". To the right of these buttons is a form for adding a new professor. It includes labels and input fields for "SSID:", "First Name:", "Last Name:", "Email:", and "Phone:". There is also a "Department ID:" label followed by a dropdown menu currently showing the value "1". At the bottom right of the form are "Submit" and "Cancel" buttons.

Add New Professor

The screenshot shows the same "Admin Application" window. In this view, the "Add new Group" button is highlighted with a blue border. The form to the right is for adding a new group. It includes labels and input fields for "ID:", "Address:", and "Capacity:". At the bottom of the form are "Submit" and "Cancel" buttons.

Add new Group

Admin Application

Administration actions:

Enroll Student

Add new Professor

Add new Group

Add new Course

Create new Account

Add new Room

Name:

Professor: Cebuc E... ▼

Room: 202 (20)... ▼

Day: Monday ▼

Hour: 8 ▼

Group: 3 AC 1 ▼

Submit Cancel

Add new Course

Admin Application

Administration actions:

- Enroll Student
- Add new Professor
- Add new Group
- Add new Course
- Create new Account
- Add new Room

Username:

Password:

Re-type Password:

Access: ☐ Student
☒ Admin
☐ Teacher

Admin Application

Administration actions:

- Enroll Student
- Add new Professor
- Add new Group
- Add new Course
- Create new Account
- Add new Room

Username:

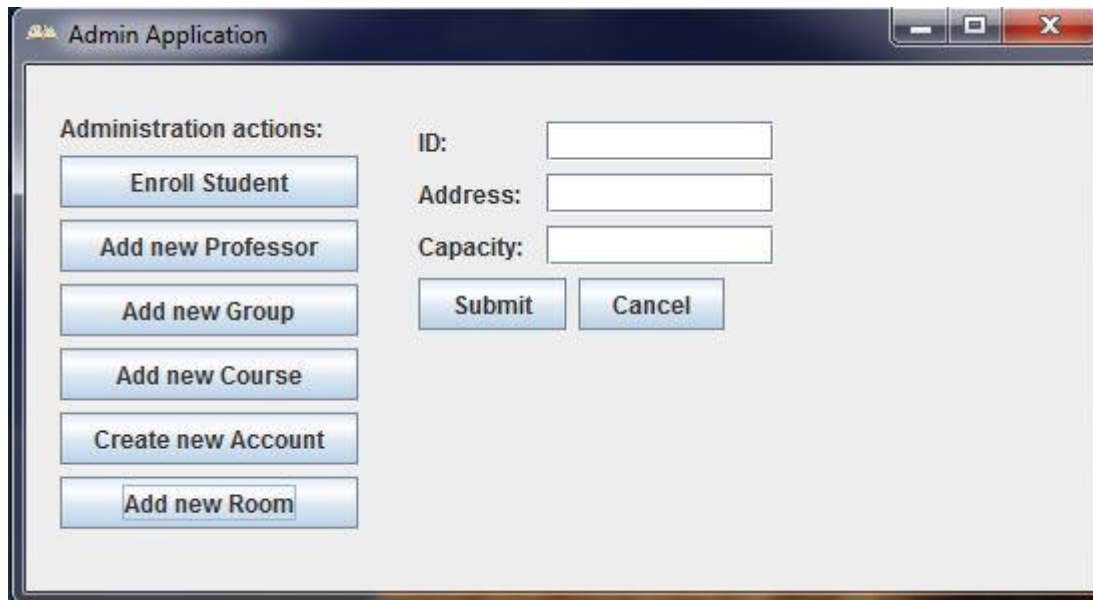
Password:

Re-type Password:

Access: ☐ Student
☐ Admin
☒ Teacher

Teacher SSID: ▼

Add new Account



Add New Room

4. Implementation and Testing

From the perspective of implementation, this application uses the Creational Design Pattern Singleton and The Behavioral Design Pattern Observer.

Singleton Pattern

Sometimes it's important to have only one instance for a class. For example, in a system there should be only one window manager (or only a file system or only a print spooler). Usually singletons are used for centralized management of internal or external resources and they provide a global point of access to themselves.

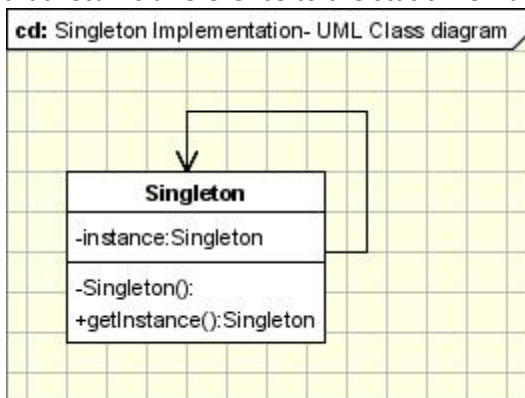
The singleton pattern is one of the simplest design patterns: it involves only one class which is responsible to instantiate itself, to make sure it creates not more than one instance; in the same time it provides a global point of access to that instance. In this case the same instance can be used from everywhere, being impossible to invoke directly the constructor each time.

Intent:

- Ensure that only one instance of a class is created.
- Provide a global point of access to the object.

Implementation:

The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member.



Singleton Implementation - UML Class Diagram

The Singleton Pattern defines a getInstance operation which exposes the unique instance which is accessed by the clients. getInstance() is responsible for creating its class unique instance in case it is not created yet and to return that instance.

Adaptation

In our application the Singleton Class is the Catalog Class, which has its constructor private and the static method getInstance() that behaves as describe above.

Observer Pattern

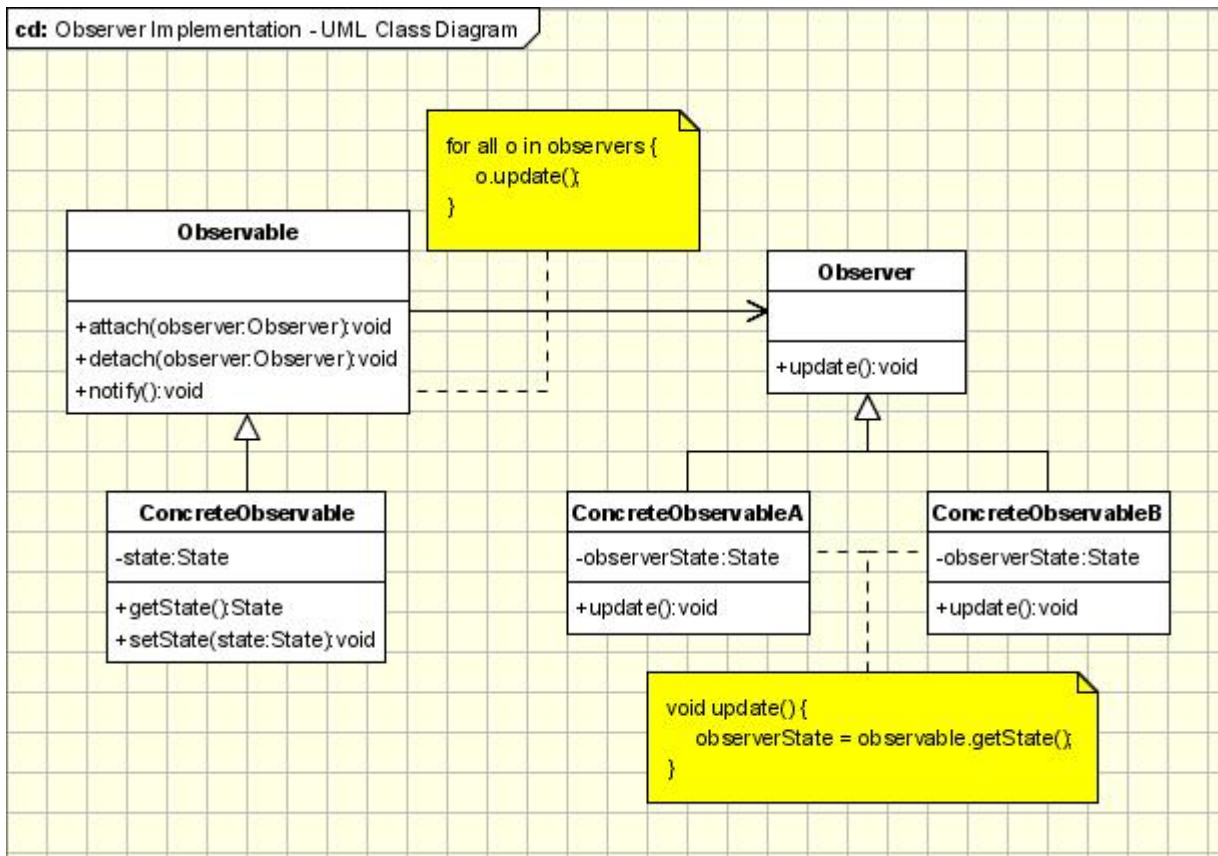
Motivation

We can not talk about Object Oriented Programming without considering the state of the objects. After all object oriented programming is about objects and their interaction. The cases when certain objects need to be informed about the changes occurred in other objects are frequent. To have a good design means to decouple as much as possible and to reduce the dependencies. The Observer Design Pattern can be used whenever a subject has to be observed by one or more observers.

Let's assume we have a stock system which provides data for several types of client. We want to have a client implemented as a web based application but in near future we need to add clients for mobile devices, Palm or Pocket PC, or to have a system to notify the users with sms alerts. Now it's simple to see what we need from the observer pattern: we need to separate the subject(stocks server) from it's observers(client applications) in such a way that adding new observer will be transparent for the server.

Intent

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Observer Design Pattern Implementation - UML Class Diagram

Implementation

The participants classes in this pattern are:

Observable - interface or abstract class defining the operations for attaching and de-attaching observers to the client. In the GOF book this class/interface is known as Subject.

ConcreteObservable - concrete Observable class. It maintain the state of the object and when a change in the state occurs it notifies the attached Observers.

Observer - interface or abstract class defining the operations to be used to notify this object.

ConcreteObserverA, ConcreteObserver2 - concrete Observer implementations.

The flow is simple: the main framework instantiate the ConcreteObservable object. Then it instantiate and attaches the concrete observers to it using the methods defined in the Observable interface. Each time the state of the subject it's changing it notifies all the attached Observers using the methods defined in the Observer interface. When a new Observer is added to the application, all we need to do is to instantiate it in the main framework and to add attach it to the Observable object. The classes already created will remain unchanged.

Adaptation

In our application we use the Observer interface and Observable class implemented in java.util. The Catalog class extends the Observable class and Observer interface is implemented by ScheduleFrame, StudentFrame and ProfessorWindow. In this way, any changes made in the Catalog will be immediately reflected into the JTables contained by these classes.

Further on we will shortly describe the most relevant methods implemented, all contained by the Catalog class:

- void addCourse(java.lang.String idCourse, java.lang.String name, java.lang.String zi, java.lang.String ora, Group group, Professor professor, Room room)

Adds a new course in the system

- boolean addGrade(java.lang.String idStudent, java.lang.String idCourse, java.util.Calendar date, double mark)

Adds a new grade.

- void addGroup(java.lang.String id, java.lang.String faculty, int year)

Registers a new group in the system

- void addProfessor(java.lang.String SSID, java.lang.String firstName, java.lang.String lastName, java.lang.String email, java.lang.String phone, int departamentId)

Adds new Professor in the system.

- void addRoom(java.lang.String id, java.lang.String address, int capacity)

Registers a new room in the system

- void addUser(java.lang.String username, java.lang.String password, int level, java.lang.String onwerId)

Adds a new user to the system

- void closeConnection()

Closes the connection to the DataBase

void connect()

Connects to the DataBase

- boolean deleteGrade(java.lang.String idStudent, java.lang.String idCourse, java.util.Calendar date)

Deletes a grade from the catalog

- void enrollStudent(java.lang.String SSID, java.lang.String firstName, java.lang.String lastName, java.lang.String address, Group group)

Adds new student to the DataBase

- static boolean isNumeric(java.lang.String str)

Returns true if the transmitted String is a number

- boolean login(java.lang.String username, java.lang.String password, int level)

Verifies login credentials

- boolean updateGrade(java.lang.String idStudent, java.lang.String idCourse, java.util.Calendar date, double newMark)

Updates the mark regarding a specific student, course and date

Testing has been done in the TestDriver class, but no relevant information has emerged from it, that should be mentioned.

5. Conclusions, what has been learned, further developments

This application only scratches the surface of what must be done in order to transfer traditional registry systems into the digital environment, but it has been a great exercise in understanding the steps that must be followed.

As result we have an application that permits real time grading, without the need of manually refreshing pages. It also manages relatively well collisions in the database accessing.

Further developments will not be mentioned, because the application barely enters its domain and the unimplemented area overshadows the accomplishments.

6. Bibliography

- j2se7
- <http://www.oodeesign.com/>