CARLETON UNIVERSITY

CUPID SYSTEM

System Design Document

Prepared by Team

GAME OF CODES

Submitted to:
Dr. Christine Laurendeau
COMP 3004 Object-Oriented Software Engineering
School of Computer Science Carleton University

Nov-23-2015

Contents

System Design Document

1. Introduction

The Carleton University Project Partner Identifier (cuPID) program is an innovative project that solves a crucial problem when building teams to work on projects. In order to create successful teams that will accomplish the goal of a project, members in a team must work in cohesion. That being said it is extremely difficult to evaluate candidates when forming teams, the cuPID program offers to sort potential candidates into teams. cuPID is a system that involves two kinds of users the Administrator and the Student (team candidates), where the Administrator user wants to form teams of Student users for any project. The program allows the administrators to create projects and students can register into these projects. The cuPID system uses a unique algorithm called the “Project Partner Identifier” (PPID) to sort students into teams based evaluating the questionnaire that each student is required to complete during their registration into the cuPID system.

The questionnaire evaluates the student’s personality and work ethics by means of 12 questions, the student fills out the same question keeping mind of the partner they are looking for. Once the students have registered into their desired projects, the administrator sets the team size and runs the PPID algorithm.

This Document leads through the process of understanding the D2 prototype design choices, proposes a new more efficient design for the program. Justifies why the new design is better and how it will be implemented, what services it would provide. This is an in depth analysis of both D2 prototype design and the Redesigned Program.

2. Subsystem Decomposition

2.1 Phase 1 Decomposition

The Phase 1 Decomposition decomposed the system that was handed in for D2. All the diagrams, in large format, for this section are indexed in the Appendix, from A-1 to A-3, which can be found at the end of this document. This section decomposes the D2 system into the 3-Tier Architectural Design. The Design consists of three subsystems which are the cuPID User Interface Subsystem, the cuPID Logic Subsystem and the cuPID Storage Subsystem. As is shown in *Figure 1*, this architecture promotes high coupling and low cohesion. There is a breakdown below of all the subsystems in this architectural style.

cuPID User Interface Subsystem

The cuPID User Interface Subsystem is responsible for handling all user interactions with the cuPID system. It is responsible for displaying information to the user and accepting user input to be sent to the cuPID Logic Subsystem. All of the interfaces (UI classes) are in this subsystem. The cuPID Logic Subsystem interacts with the user using the cuPID User Interface Subsystem. In the 3-tier style, this subsystem has very low cohesion as almost every UI class communicated with a control class in the cuPID Logic Subsystem.

cuPID Logic Subsystem

The cuPID Logic Subsystem is responsible for controlling all the features implemented in the cuPID system. It achieves this by contains all the control classes of the cuPID system. This subsystem is responsible for launching the correct user interfaces, retrieving information from storage and populating the UI, error checking all user inputs and passing user inputs to the Storage subsystem for storage. The cuPID Logic Subsystem is not a very cohesive subsystem and has very high coupling with both, the cuPID User Interface and the cuPID Storage, subsystems. The control classes in this subsystem allow the program progress in a linear fashion to allow the user use the features of the program.

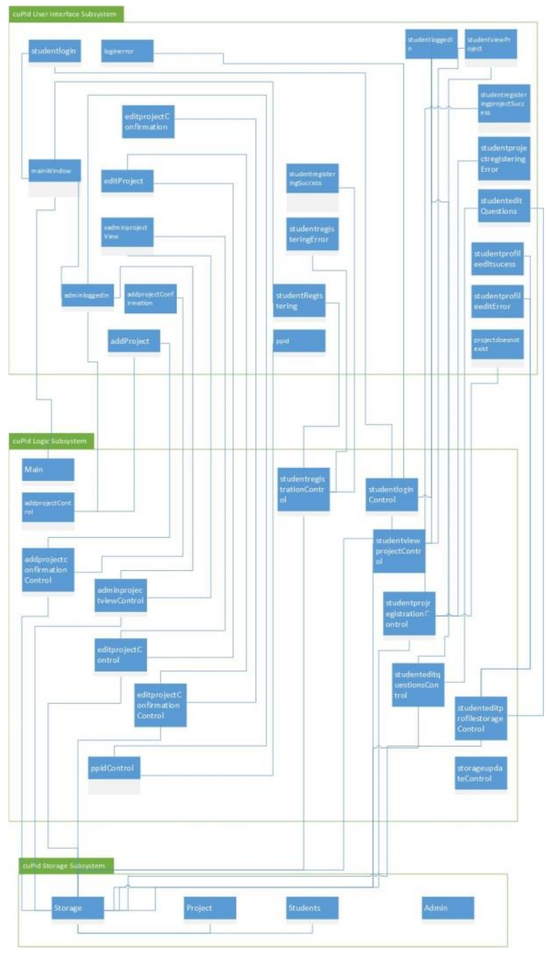


Figure 1

cuPID Storage Subsystem

The cuPID Storage Subsystem is responsible for the persistent storage of the data. It is the only subsystem with access to the persistent database. This subsystem is responsible for retrieving data from the database and passing it to the control class that requested. It also accepts incoming data and writes it to the database by making a new record or overwriting an existing record. This subsystem only interacts with classes in the cuPID Logic Subsystem. It also provides some error checking by ensuring certain values are unique to guarantee no duplication of records. The Project Entity Object was the only one used in this subsystem. Student information was passed back and forth by vectors.

2.2 System Decomposition

The System Decomposition displayed in *Figure 2* shows the new Redesign of the system. It shows a decomposition of the system into a Repository Architecture Design. With this style, the system now has four logical subsystems; cuPID Student Subsystem, cuPID Administrator Subsystem, cuPID Main Subsystem and cuPID Storage Subsystem. All the diagrams, in large format, are indexed in the Appendix, from B-1 to B-3, which can be found at the end of this document. This architecture style promotes high cohesion and low coupling within each logical subsystem. This is accomplished by grouping the functionality of the two types of users into the same subsystem, using façade classes to communicate between each subsystem and the cuPID Storage subsystem and using control classes to communicate between the cuPID main subsystem and the users' subsystems. It allows for the splitting of functional and non-functional requirements into each subsystem. It also mostly divides the use cases into classes that are mostly in one subsystem.

cuPID Student Subsystem

The Student Subsystem is in charge of all activities pertaining to the Student User. This subsystem takes control of whenever a student wants to register or login to the cuPID system. It retrieves information from the storage and updates the storage via the student storage façade class. The subsystem allows students to register, update their profiles and register in projects. This subsystem, along with storage handles all use cases for a student that were defined in D1. **UC-05, UC-06** & **UC-07** are mostly controlled by this subsystem. It also implements functional requirements **F-04, F-05, F-06, F-07 & F-08**.

cuPID Admin Subsystem

The Admin Subsystem is in charge of all activities pertaining to the Administrative User. It takes control whenever an administrator logs in. It allows the administrator to create and edit projects and run the PPID algorithm on projects to group students as stated in **UC-01, UC-02, UC-03** & **UC-04**. This subsystem updates the storage and retrieves information from the storage via the admin storage façade class. It implements **F-01, F-01-01, F-01-02, F-01-03, F-01-04, F-02, F-03, F-03-01, F-12** & **F-13.**

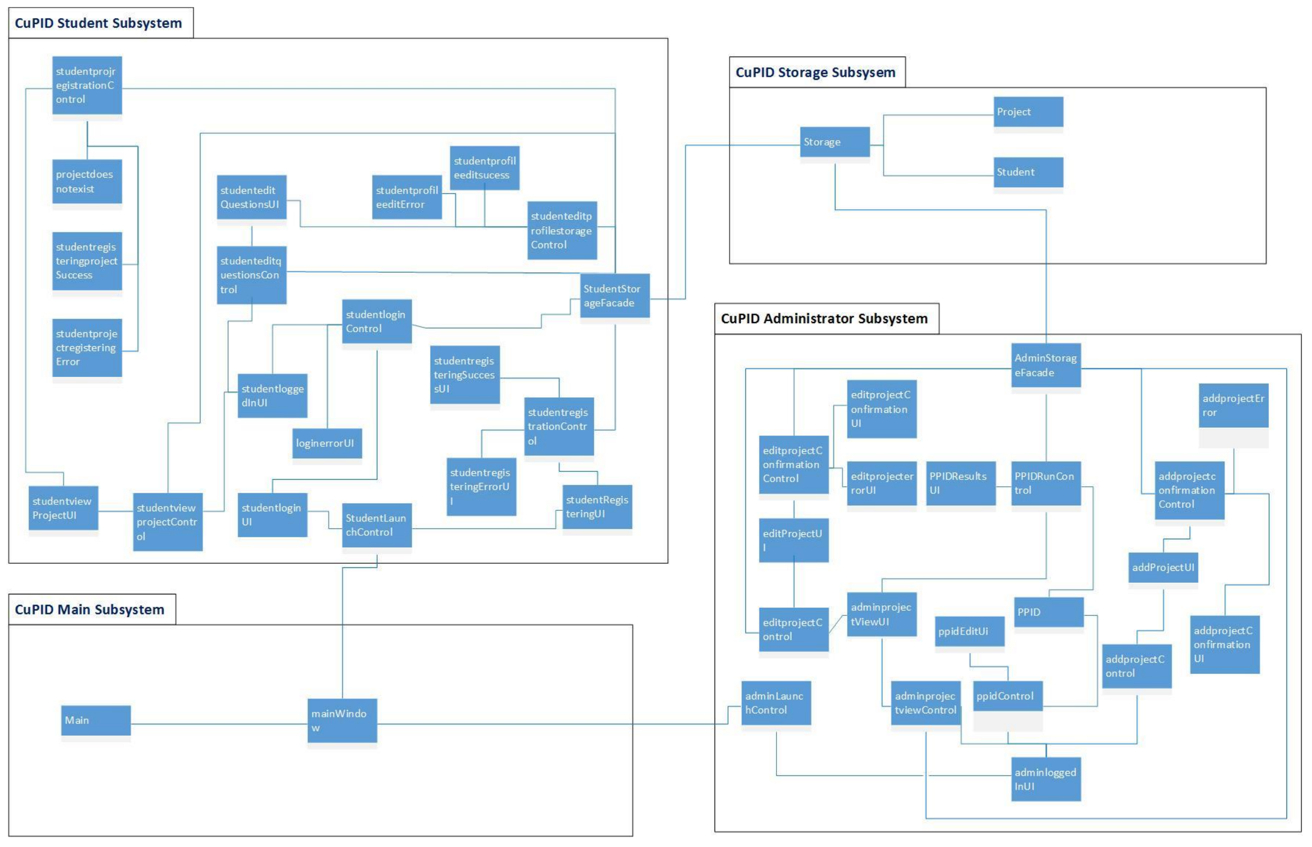


Figure 2

cuPID Main Subsystem

The Main Subsystem is the first subsystem to launch when the program is started and launches the main user interface. It allows students or administrators to login and passes control to the correct subsystem. It also launches the first instance of the storage class that is used by the rest of the system.

cuPID Storage Subsystem

The Storage Subsystem handles all activities pertaining to the storage of information from student and admin subsystem. It retrieves information from the database and passes it to subsystem that requested it. The Storage subsystem makes a copy of the information requested by the other subsystems and passes the copy to the requesting class. This ensures integrity of the data and verifies all incoming data is correct before updating the storage. It plays a role is most of the Use Cases identified in D1 and accomplishes functional requirements **F-09, F-10** & **F-11**.

2.3 Design Evolution

The prototype that was built for D2 was decomposed to a 3-tier architectural style. It had 3 layers and a closed architecture. There was an interface layer, an application layer and a storage layer. This decomposition had very high coupling and very low cohesion, as was made evident by the UML class diagram with packages in *Figure 1*. In the 3-tier architectural style, every UI class in the User Interface subsystem communicates with a control class or is launched by a control class in the cuPID logic subsystem. Many of the control classes in the cuPID logic subsystem communicate with the storage class in the storage subsystem. This also contributes to the low cohesion and high coupling as very few control classes in the logic subsystem communicate with each other. To improve our design, we tried a few architectural styles and found the best results using a Repository architectural style using the façade design pattern to communicate with other subsystems. The same disadvantages would apply to a four-tier system as well as the fact that the system is on one machine and does not require a network.

The MVC architectural style did not work because the system uses control classes to send and retrieve information and launch the UIs. Splitting these into MVC would break the cohesi on and make the coupling strong.

Client/server and peer-to-peer could not be used because the entire system is on one machine, the functional requirements did not request the system to be applicable over a network. Therefore the client/server architectural style would not work. The same

The Repository Architectural Style allowed the cuPID subsystem to be decomposed into 4 main subsystems. These subsystems all have high cohesion as they all implement very specific functionality that is related. All classes communicate within the subsystem and use façade classes to communicate with other subsystems. This allows for controlled and restricted access between subsystems to ensure unwanted behavior does not occur. This style works best because only one repository is being updated. All the subsystems are very independent and communicate through the repository subsystem. The style is recommended for complex data processing and allows new services to ask. Although the style is prone to high coupling with the repository class, using the façade design pattern lowers this by only allowing one class to communicate with the storage from each subsystem.

This design change of the system allows for a more modular and independent design of each subsystem. It allowed the Use Cases and the Functional requirements to be split between the cuPID Student, cuPID Administrator & cuPID Storage subsystems. These subsystems are more specialized than the previous decomposition and allow for new operations to be added without worrying about making changes to another subsystem.

3. Design Strategies

This section will outline how the system is configured and on how many machines. It will also break down how the database is used to store and retrieve information and accomplish the functional requirements and parts of different use cases. This section will end by explain the design pattern used and why others that were considered did not work.

3.1 Hardware/Software Mapping

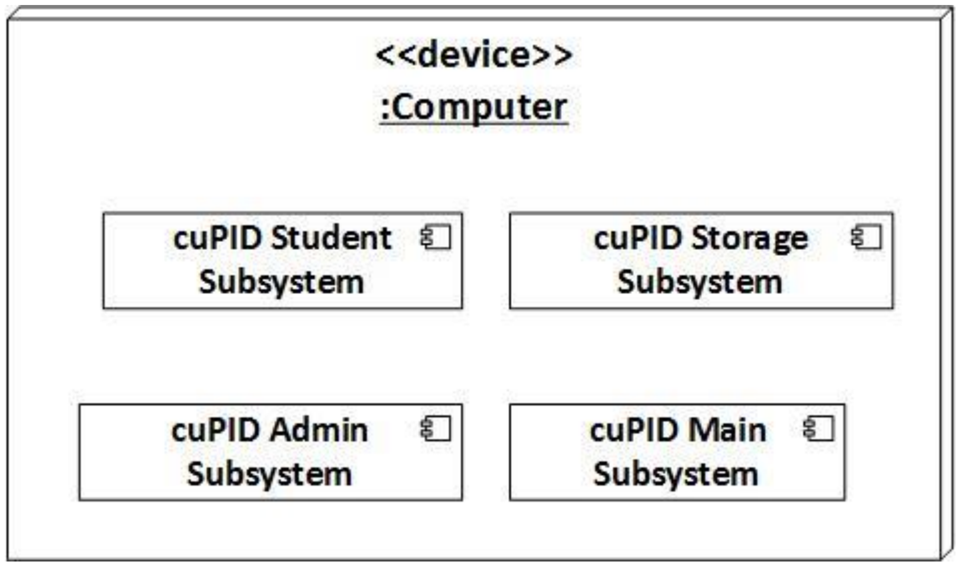


Figure 3, See Appendix C-1 for bigger size

The new design follows the Repository Architecture pattern. There are four subsystems that are created according to the programs functionality and requirements. The functionality of the program is based on the two types of users of the system. The Student user's functionality is preserved in the cuPID Student Subsystem and similarly the Admin's functionality is preserved in the cuPID Admin Subsystem. The remaining subsystems are Main and Storage. Main subsystem is responsible for launching the program, and the Storage subsystem is supposed to mimic the Repository in the Repository Architectural design pattern. The Storage subsystem stores the state of the program and any updates made by either subsystem. Information is retrieved according to the requests make by Student Facade Class in Student subsystem and Admin Facade Class in the Admin Subsystem and modifications are saved accordingly through the same Facade Classes in the respective subsystems. There is only one node in the system because the program runs only on one computer/ device and the program can execute independently without any relay from any other device.

3.2 Persistent Data Management

For storage purposes of this project we decided on using SQLite database implementation in QT (QSQL database). This database uses tables to store data, and each table has a column that is used as the key, i.e it is unique and cannot be duplicated. There are no objects stored in the persistent storage as data is passed and retrieved using vectors of entity object pointers.

For the cuPID system we have three tables. One for a student's profile, named Student. This table consists, of ID (student number) which in this case is unique for each student and thus we make it a key.

It also consists of the student's name and the respective values for the 24 questions the student answers in their profile, for his/her self and the partner he/she is looking for.

The second table is the Projects table, used to store the details of the projects uploaded by the administrator. Here we dedicate a column for ID, which would serve as the key (being unique and distinct), making it easier to retrieve and registering into a project based on a key for each project. We also store the projects name and description in this table. For D2, assigning ID (project ID) to be the key, it only makes the id to be unique allowing the project to have duplicate names, to eliminate this we will put a "unique" sql constraint on project name. The unique constraint will not allow duplicate project names, but will allow incrementing of the keys.

The last table in our database, is the Sturegs (students registered in projects) table. This table, consists of two columns (Student Id - sid and Project Id - pid), both being keys as both are distinct. We use the student's student number from the student table and the projects id from the projects table. This allows us to view projects a student registered into and also view all the students registered in a particular project.

3.3 Design Patterns

There were many design patterns that were considered for this system. The requirements forced us to look at the 3 categories of design patterns introduced in class to find the correct one. We considered the Abstract Factory Design Pattern from the Creational Design Patterns category, the Façade and Proxy Design Patterns from the Structural Design Patterns and the Observer Design Pattern is Behavioral Design Patterns.

We decided not to use the Abstract Factory Design Pattern because our two entity objects do not have any common data. Also, none of our entity objects are related and can exist independent of each other. All other classes are independent and have unique functionality; they cannot be abstracted to the factory because they are not related and do not have a common theme. The client is either an Administrator, who can only create a project object, or a Student, who can only create a student profiles. There is no need to provide an abstract interface when each user can only create one object.

The reason not to use the Proxy Design Pattern is because all of the data is stored in the database. The integrity of the data is ensured because the student object is created when a student is registered but the data is not overwritten in the database until the control classes and the database ensure the values passed are valid. The database checks that all student numbers are unique when a student is registering, and then the system does not allow a student to change their student number. The control classes ensure the values passed in are valid and in range and the system does not use open ended questions to ensure only valid data is passed. Updates are done as soon as data is submitted once it is checked for correctness. For these reasons, a proxy student object is not required.

Projects are created by administrators and passed to the database. The database gives each project a unique ID to ensure there is not duplication and verifies that all project names are unique. If a duplicate project name is passed, the database returns an error and the storage classes returns an error to the control notifying that the entry was not made. When an administrator or student asks for the project list, the database creates a vector of project objects and passes it to the appropriate control. The students do not have the option to edit the project. The administrator can edit a project but the control and the database check for unique title and valid entries to ensure the integrity of data. All changes are committed to the database as soon as they are made and verified. These measures ensure the integrity of the data.

The observer design pattern is one that works best with MVC. It requires that UIs, controls and entity object be separated. It is used when an object has dependents. We tried to decompose our system into an MVC to implement an observer design pattern in the Phase 1 Decomposition, but all of our objects only had one dependent. The system either had storage to UI dataflow, facilitated by control, or UI to storage dataflow, also facilitated by control. Although the system has a common communication pattern, to and from storage and UI, the route is though different control classes that request and send different types of data. These control classes are responsible for verifying data, depending on what the user has requested or sent. Because of this, the control classes cannot be merged. The intent of this design pattern is to update all of its dependents. Because there are not many dependents that need to be updated whenever an object changes state, this design pattern is not needed for the system. Also, this design pattern requires that observers be able to attach and detach from the subject, but this is not possible in the cupid system as each UI and control class need each other to ensure the system works well.

The system uses a façade design pattern. It is used by the Student and Admin subsystems to provide a simplified interface to the larger storage class. It is used to hide the complexity of the storage class from the Admin and Student subsystems and ensure only the correct operations are performed by each. These two subsystems are provided with an easier to use interface use the façade. Without the façade the Admin, Student and Storage subsystems would have very high coupling and dependencies. The façade classes allow the system to be more modular and portable. The façade also keep a level of security to the database. It does not allow direct access to the storage class and database, all the data passed goes through validity checks. For these reasons, the façade Design pattern was most applicable.

4. Subsystem Services

This section will outline the services provided by the different subsystems and to which subsystem. It uses ball-and-socket component diagrams to show the services provided.

`Main subsystem:

The *cuPID main subsystem* does not provide any service to other sub-systems. The main subsystem however, does provide means for the user of the program to interact with other subsystems. This subsystem uses the services of **logging the student in(*appendix*. E2)** and **student registering (UC-05) (*appendix*. E1)** from the cuPID Student Subsystem. It is the starting point of the Use Case **UC-05**. It also uses the **admin logging in (*appendix*. E3)** service from the cuPID Admin Subsystem.

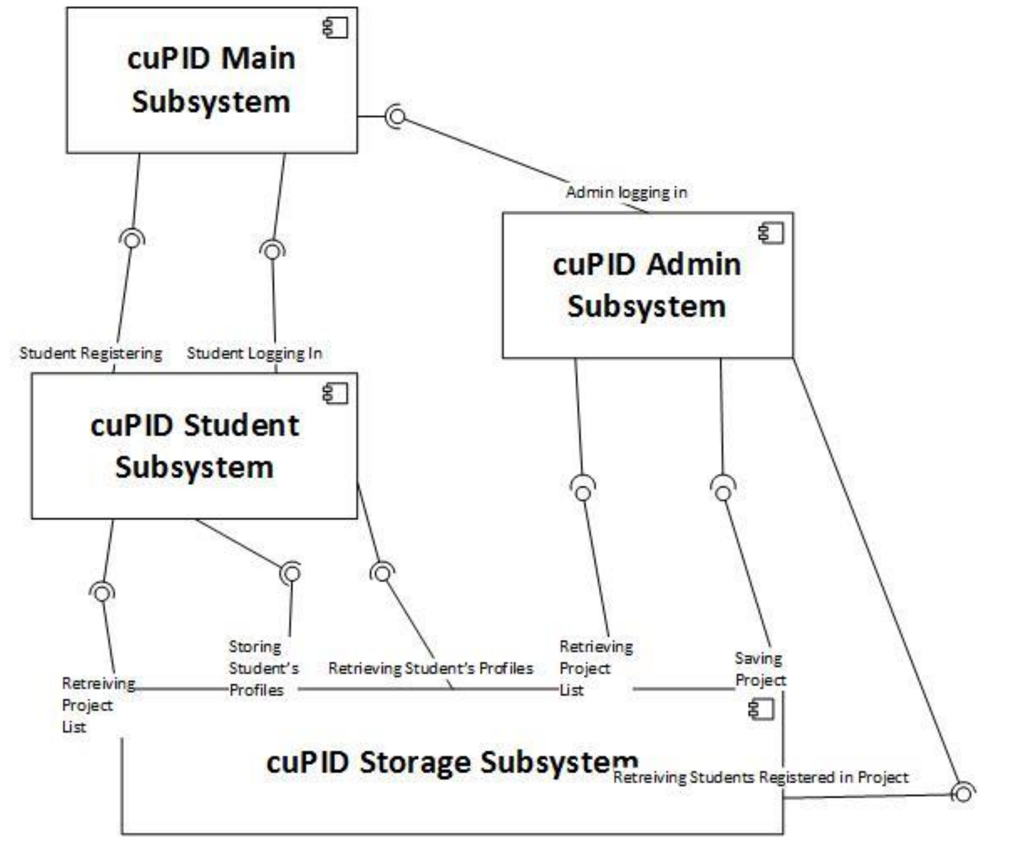


Figure 4, See Appendix D-1 for larger size.

cuPID Student subsystem:

The *cuPID student subsystem* provides various services to other subsystems in the cuPID system. Following is the list of subsystems along with a description of services provided by student subsystem to each one.

cuPID Student subsystem to cuPID Main Subsystem:

The cuPID student subsystem provides *Main subsystem* mechanism to check whether or not the student ID provided by a student to login into the cuPID system is valid. This service is called the ***Student***

Logging In service. It verifies the provided student number is registered in the system and launches the *Student Logged In UI* or prompts the user that the student name is invalid. This service is the launching point of Use Cases **UC-07, UC-08** & **UC-09.** The check is done through *storage subsystem* which provides this checking mechanism to the student subsystem via *retrieving student profile* service. These implement functional requirements **F-05**, **F-10**. The classes involved in this service are depicted in the class diagram shown in **(pg.23 *appendix* E2)**

If a student is not previously registered in the cuPID system they will be prompted to fill a registration form (**F-05 and UC-05)**. The *cuPID Student Subsystem* provides service to the *cuPID Main subsystem* to help the new student register into the cuPID via ***Student Registering*** service. The classes involved in this service are depicted in the class diagram shown in **(pg.22 *appendix* E1)**

After filling a form the *cuPID student subsystem* passes the student information to the *cuPID storage subsystem* (**UC-05, F-05, F-10)** which holds mechanism to check and verify that student who is registering hasn't registered before. The cuPID storage subsystem does this check using the *Storing Student profile* service (**F-10**).

cuPID Storage Subsystem:

The *cuPID Storage subsystem* provides various services to the subsystems within the cuPID program. It implements the functional requirements **F-09 & F-10.** Following is the outline of the services provided by the *cuPID Storage subsystem* to other subsystems with in the cuPID system.

cuPID storage subsystem to cuPID Admin Subsystem:

The cuPID storage subsystem enables the *cuPID admin subsystem* to store the newly created project. This is achieved by the *cuPID storage subsystem* via ***storing projects services*** (**F-09 and UC-01**). The classes involved in this service are depicted in the class diagram shown in **(pg.31 *appendix* E11).** The ***storing project*** service checks whether or not a project that is being stored has been previously registered. If a project has been created before it will disallow the Admin to re-create it.

The cuPID storage subsystem also provides the admin with the list of registered projects. This is achieved by the invocation of the ***retrieving projects* service (F-09)**. The classes involved in this service are depicted in the class diagram shown in **(pg.25, 30 *appendix* E6, E10).** The admin makes a request to view project which in return invokes the service to output the name and descriptions of all the previously registered projects in the *cuPID storage subsystem.*

The *cuPID storage subsystem* provides a list of students registered in a particular project to run the PPID. This is achieved my calling the ***Retrieving Students Registered in Project*** service (**F-09, F-10, F-11, F-12, F-13, UC-03**). The classes involved in this service are depicted in the class diagram shown in **(pg.24 *appendix* E4)**

It gets the list of students registered in a project and runs the PPID to group them.

cuPID storage subsystem to cuPID student Subsystem:

The cuPID storage subsystem provides the *cuPID student subsystem* with ability to store the newly created student's profiles (**UC-05, F-04, F-10)**. This is accomplished by calling the ***Storing Student's Profile*** service. The classes involved in this service are depicted in the class diagram shown in **(pg.22 *appendix* E1)** *The cuPID storage subsystem* has a check mechanism that checks and notifies the *cuPID student subsystem* whether or not the student is already registered. Two students with the same student number cannot be stored simultaneously and *cuPID Storage subsystem* is equipped with mechanism to prevent that from happening.

The cuPID storage subsystem also provides the *cuPID student subsystem* with ability to retrieve project lists that have been registered by the *cuPID Admin subsystem.* This is achieving via ***retrieving project list*** service (**F-09, UC-02**). The classes involved in this service are depicted in the class diagram shown in **(pg.25 *appendix* E5)**

The cuPID storage subsystem provides *cuPID student subsystem* with the ability to retrieve previously made profile. This is achieved by ***retrieving student profiles* service**. The classes involved in this service are depicted in the class diagram shown in **(pg.27 *appendix* E7).** ***Retrieving student profiles*** service is only invoked by a student who is already logged in thus check for an invalid request is eliminated.

The cuPID student subsystem directs a new student's information to be stored in the primary database hosted in the *cuPID storage subsystem*. This is facilitated by the ***Storing student profiles* service** (**UC-05, UC-07)**. The classes involved in this service are depicted in the class diagram shown in **(pg.29 *appendix* E9)**

The cuPID storage subsystem has a verification mechanism that checks and notifies the *cuPID student subsystem* whether or not the student is already registered. Two students with the same student number cannot be stored simultaneously and *cuPID Storage subsystem* is equipped with mechanism to prevent that from happening.

cuPID Admin subsystem:

cuPID admin subsystem to cuPID main subsystem:

cuPID admin subsystem provides the *cuPID main subsystem* with ability to login and view the admin control. There is no check for admin to login. This is done via ***admin logging in* service**. The classes involved in this service are depicted in the class diagram shown in **(pg.24 *appendix* E3)**

5. Class Interfaces

Student Registering Service: (Student)

If a new Student wants to register in the cuPID system, they are able to do so by clicking the registration button from the **MainWindowUI** window. This launches the **StudentLaunchControl** which in turn launches the **StudentRegisteringUI**. The student fills out the profile and clicks submit. Upon submission, the **StudentRegistrationControl** ensures all values entered are valid and sends it to Storage via the **StudentStorageFacade** class. If the Storage class returns no error, the **StudentRegistrationSuccessUI** is launched or else the **StudentRegistrationErrorUI** is launched. This service implements **UC-05,UC-12,F-05**. The classes involved in this service are depicted in the class diagram shown in **(pg.22 *appendix* E1)**

Student Login Service:

The student clicks the Login button on the **MainWindownUI**, This launches **StudentLaunchControl** which launches **StudentLoginUI.** The student enters her student number and clicks login. This launches the **StudentLoginControl**. This class passes the information to the **StudentStorageFacade**. The Façade class passes information to the storage. If the Storage Class returns and error, the façade class return an error. This causes the **StudentLoginControl** to launch an error message or else the **StudentLoginControl** launches the **StudentLoggedIn** Class. The classes involved in this service are depicted in the class diagram shown in **(pg.23 *appendix* E2)**

Admin Login Class:

The main windows provides admin login interface. The Admin doesn't need any check therefore upon click of the login button, admin is immediately granted access to the cuPID system. The admin is done welcomed with **AdminLoggedinUI** which provides admin range of options to choose from. The classes involved in this service are depicted in the class diagram shown in **(pg.24 *appendix* E3)**

Retrieve students in projects: (ADMIN)

Admin can launch the PPID on the students registered in a specific project through **AdminProjectViewUI** class. **AdminProjectViewUI** class launches the **PPIDRunControl** class which retrieves the set attributes fixed by the Admin in the PPID class. Equipped with the attributes retrieved from the PPID class, the **PPIDRunControl** class then uses the **AdminStorageFacade** to retrieve the students registered in the projects on which the PPID was launched. The **AdminStorageFacade** retrieves the list from Storage and passes it to the **PPIDRunControl**. The **PPIDRunControl** groups the students and the results are displayed through **PPIDResultUI** which is launched from the **PPIDRunControl** class. This service implements **UC-11** The classes involved in this service are depicted in the class diagram shown in **(pg.24 *appendix* E4)**

Retrieving Project Lists (Student service):

The retrieval of the project lists in the cupid system is initiated from the **StudentLoggedInUI** class which launches the **StudentViewProjectControl** class. **StudentViewProjectControl** class retrieves the projects from the main Storage via **StudentStorageFacade** and displays them on the screen using the **StudentVIewProjectUI**. This service implements **UC-13, UC-10, F-08**.The classes involved in this service are depicted in the class diagram shown in **(pg.25 *appendix* E5)**

Retriving projects (Admin Service)

Once the admin is logged in **AdminLoggedinUI** enables the admin to view projects through **AdminProjectViewControl** which launches the **AdminProjectViewUI**. **AdminProjectViewUI** then instantiates an instance of **AdminStorageFacade** which retrieve the Project list from the main storage. This service implements **UC-17, F-01-04, UC-10**.The classes involved in this service are depicted in the class diagram shown in **(pg.26 *appendix* E6)**

Retrieving student profile (Student Service)

Once the student is logged in, they are directed to the **StudentLoggedinUI**. If a student chooses to Edit their questionnaire and profile, they can do so by clicking on Edit profile butto n. The button launches an instance of **StudentEditQuesitonsControl** which launches the **StudentEditQuestionsUI**. The **StudentEditQuestionsControl** retrieves the profile of the student and populates the **StudentEditQuestionUI**. This service implements **UC -18**.The classes involved in this service are depicted in the class diagram shown in **(pg.27 *appendix* E7)**

Student Registering in Project Service

A Student views the list of the project through **StudentViewProjectUI** which launches the **StudentRegistrationControl**. The constructor for **StudentRegistrationControl** takes id of the project which student wants to register into and passes it on to the **StudentStorageFacade** which passes it on to the main storage to be processed. This service implements **UC -06, UC-14, UC-15, F-05** .The classes involved in this service are depicted in the class diagram shown in **(pg.28 *appendix* E8)**

Student Storage Edit Question System:

Once logged in the student has the option of editing the previously filled questionnaire for his/her profile. **StudentEditQuestionsUI** is where the student views the questions and edits them. The **StudentEditProfileStorageControl**, takes the edited questions and sends to the **StudentStorageFacade**, which later deals with the Storage. Here the edited questions are stored for the student. Once that is done, the **StudentEditProfileStorageControl** generates a success message via the **StudentProfileEditSuccessUI**, if it was not successful or an error message if it wasn't successful via **StudentProfileEditErrorUI**. This service implements **UC-07**, **F-06**.The classes involved in this service are depicted in the class diagram shown in **(pg.29 *appendix* E9)**

Edit Project (admin service)

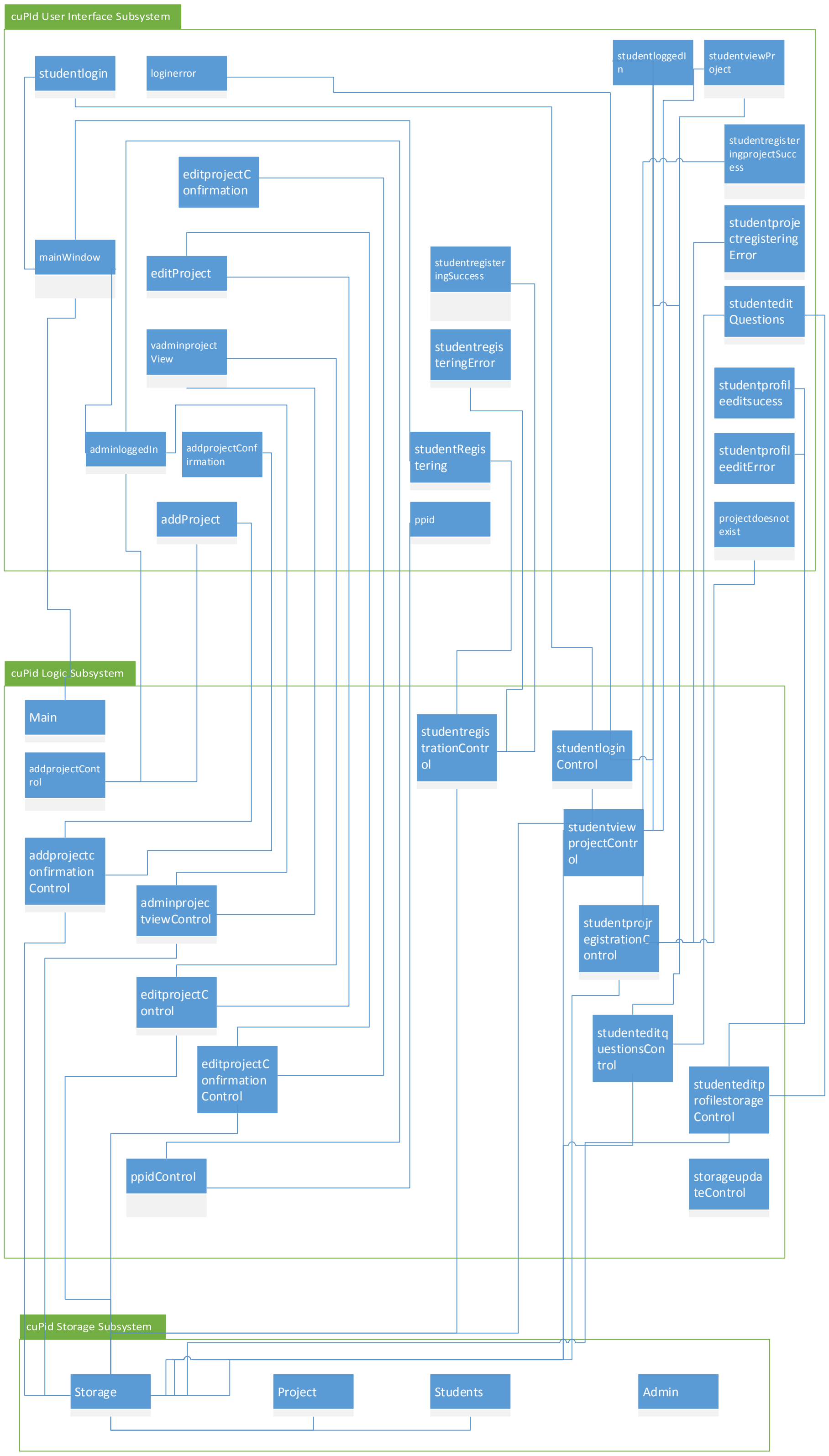
The administrator of the cuPID System has ability to modify the project's name and description. The admin can choose to edit a particular project by selecting it from the list. The **AdminProjectViewUI** then invokes **EditProjectControl** class which takes in a string as input (id of the project that needs to be edited). The **EditProjectControl** then launches the **EditProjectUI** which takes the admin's input and passes in all the modified values to the **AdminStorageFacade** which then passes it on the storage. This service implements **UC -02, UC -10, F-01-02**, **F-01-03** . The classes involved in this service are depicted in the class diagram shown in **(pg.30 *appendix* E10)**

Adding new project admin service:

Once logged in, the admin clicks add project button on the **AdminloggedinUI**, which goes to the **AdminProjectControl**. This generates the **AddProjectUI**, where the page allows the admin to add name and description for the project. This information is forwarded to the **AddProjectConfirmationControl**. The control class forwards the information to the **AdmonStroageFacade** and the project information is stored in the Storage. The control generates the **AddProjectConfirmationUI** for success or **AddProjectConfirmationError** for error. This service implements **UC-01, F-01-01**. The classes involved in this service are depicted in the class diagram shown in **(pg.31 *appendix* E11)**

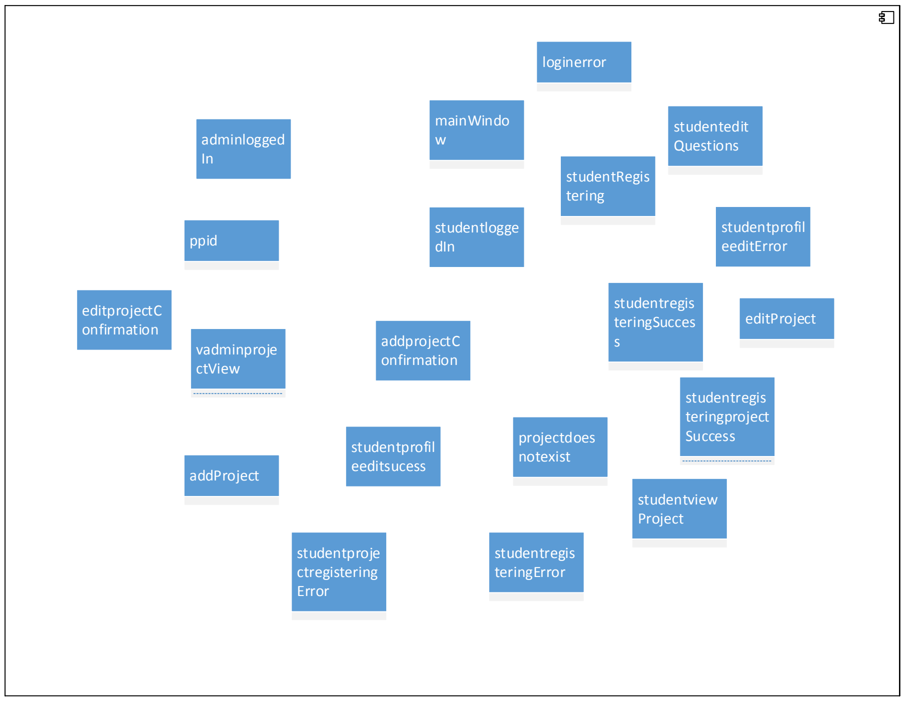
Appendix

A-1 Phase 1 Decomposition Class and Package Diagram

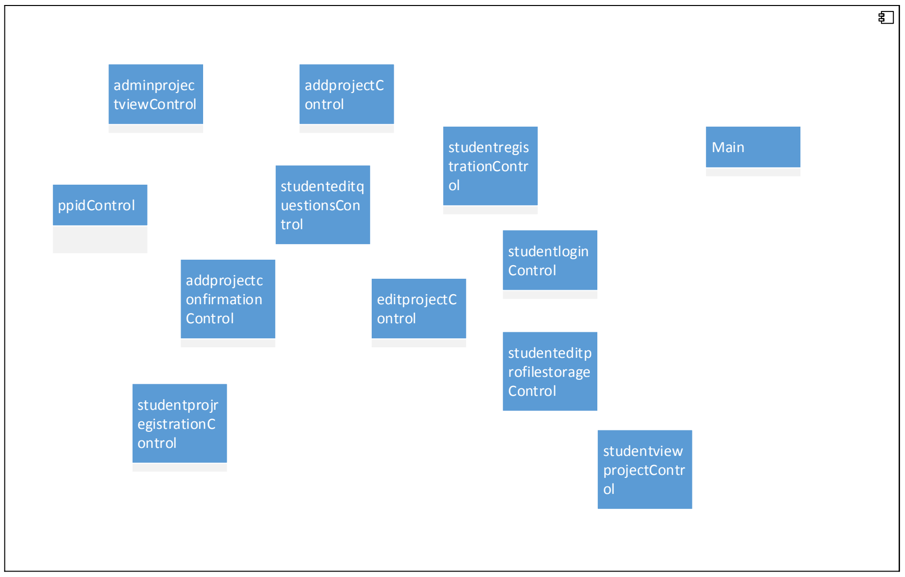


A-2 UML Component Diagram

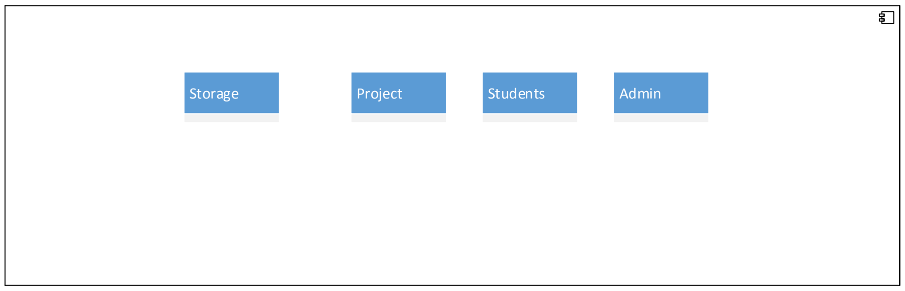
cuPID User Interface



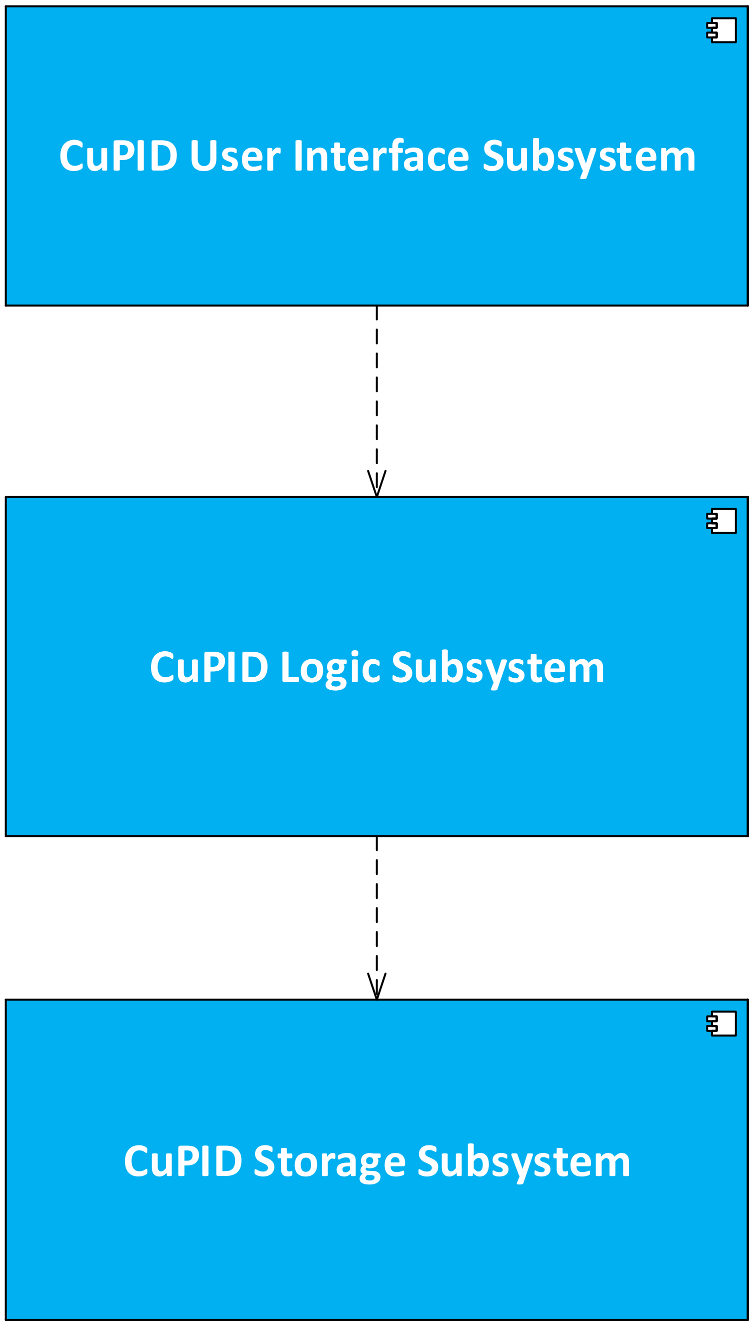
cuPId Logic Subsystem


adminprojectviewControl
addprojectControl
studentregistrationControl
Main
ppidControl
studenteditquestionsControl
studentloginControl
addprojectconfirmationControl
editprojectControl
studenteditprofilestorageControl
studentprojregistrationControl
studentviewprojectControl

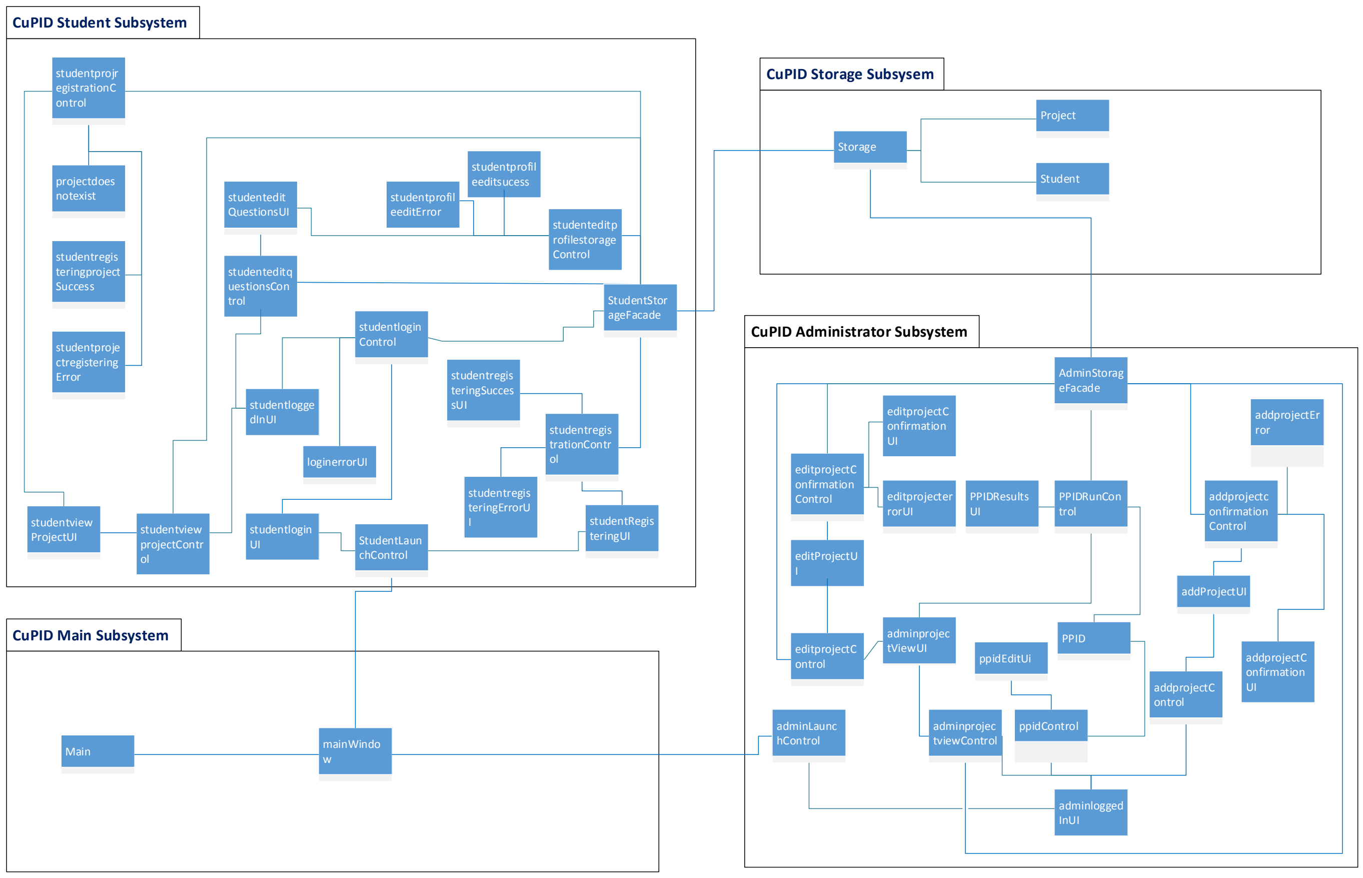
Storage Subsystem



A-3 UML Component Diagram

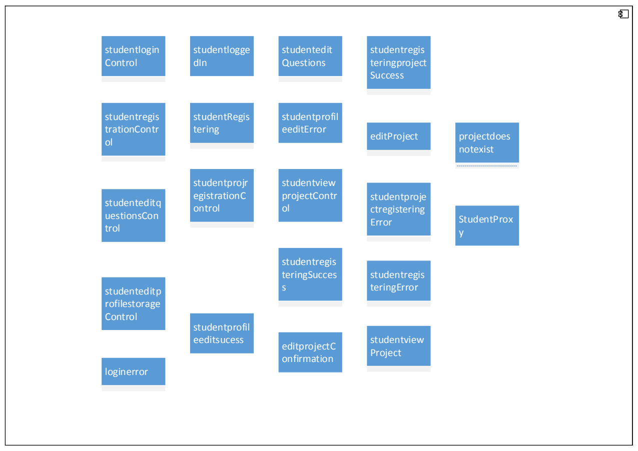


B-1 System Decomposition UML Class and Package diagrams

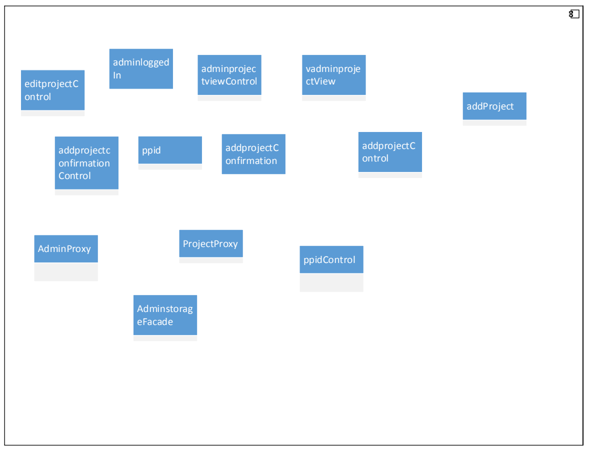


B-2 UML Component Diagram

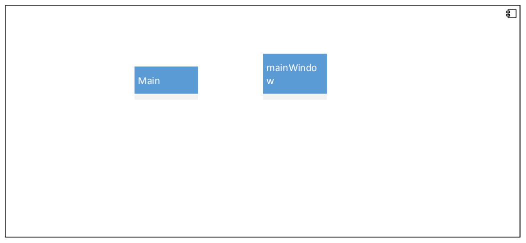
CuPID Student Subsystem



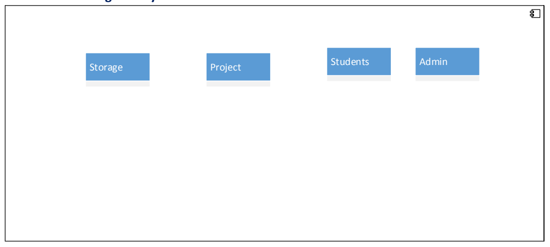
CuPID Admin Subsystem



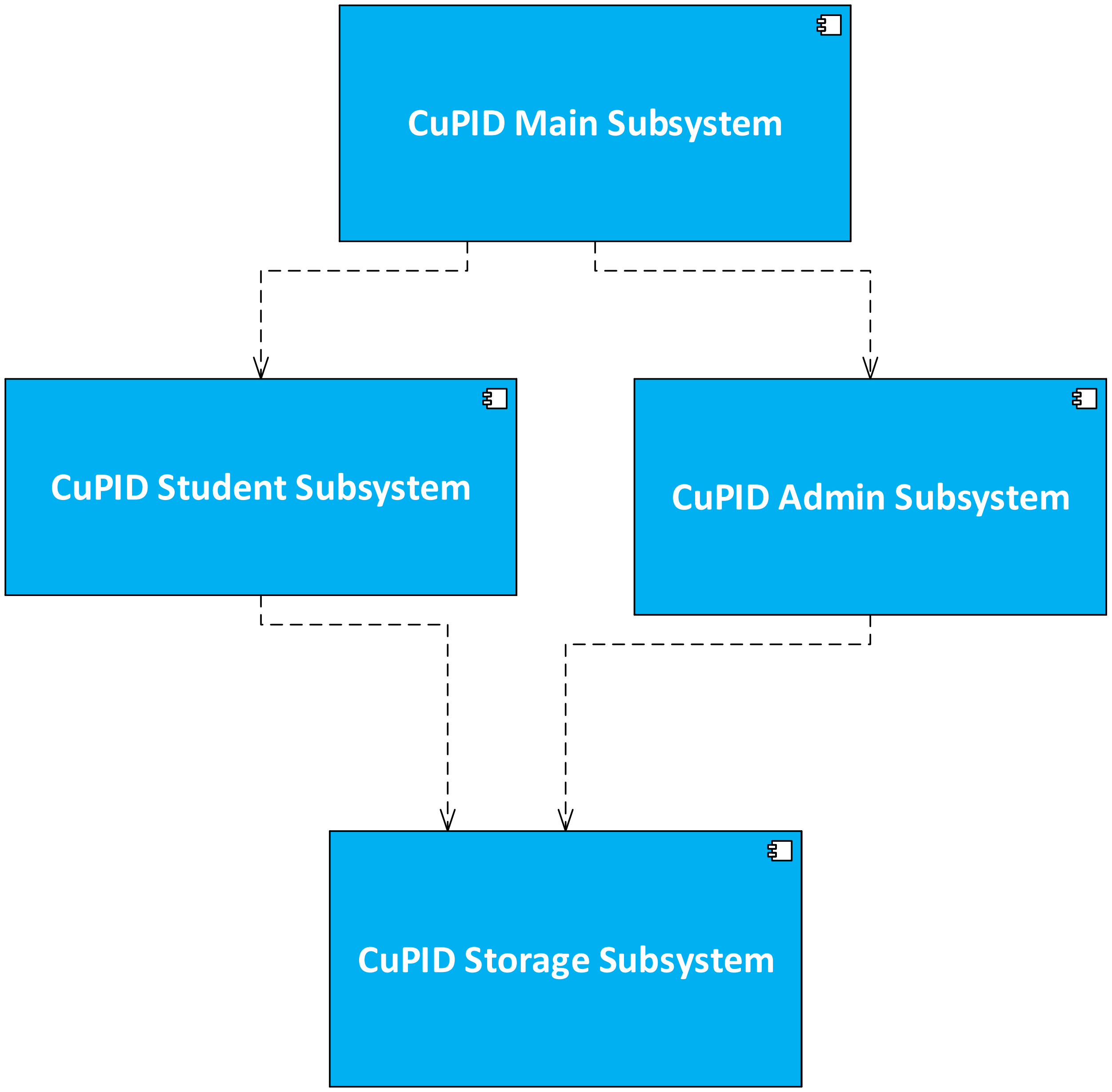
CuPID Main Subsystem



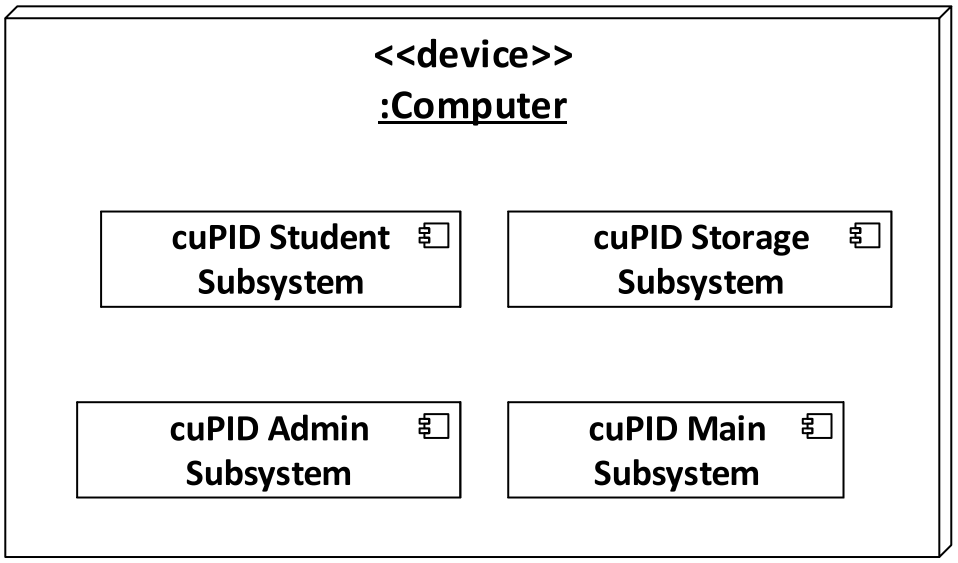
CuPID Storage Subsysem



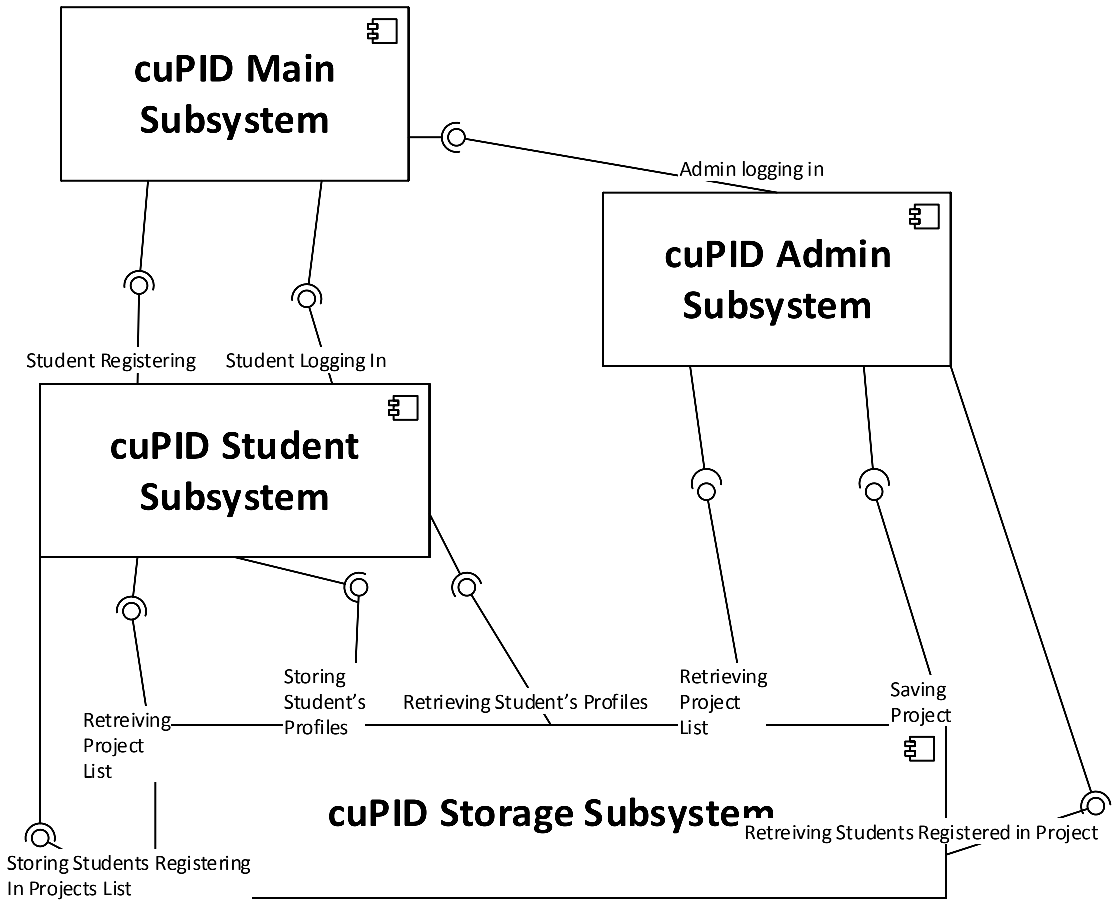
B-3 UML Component Diagram



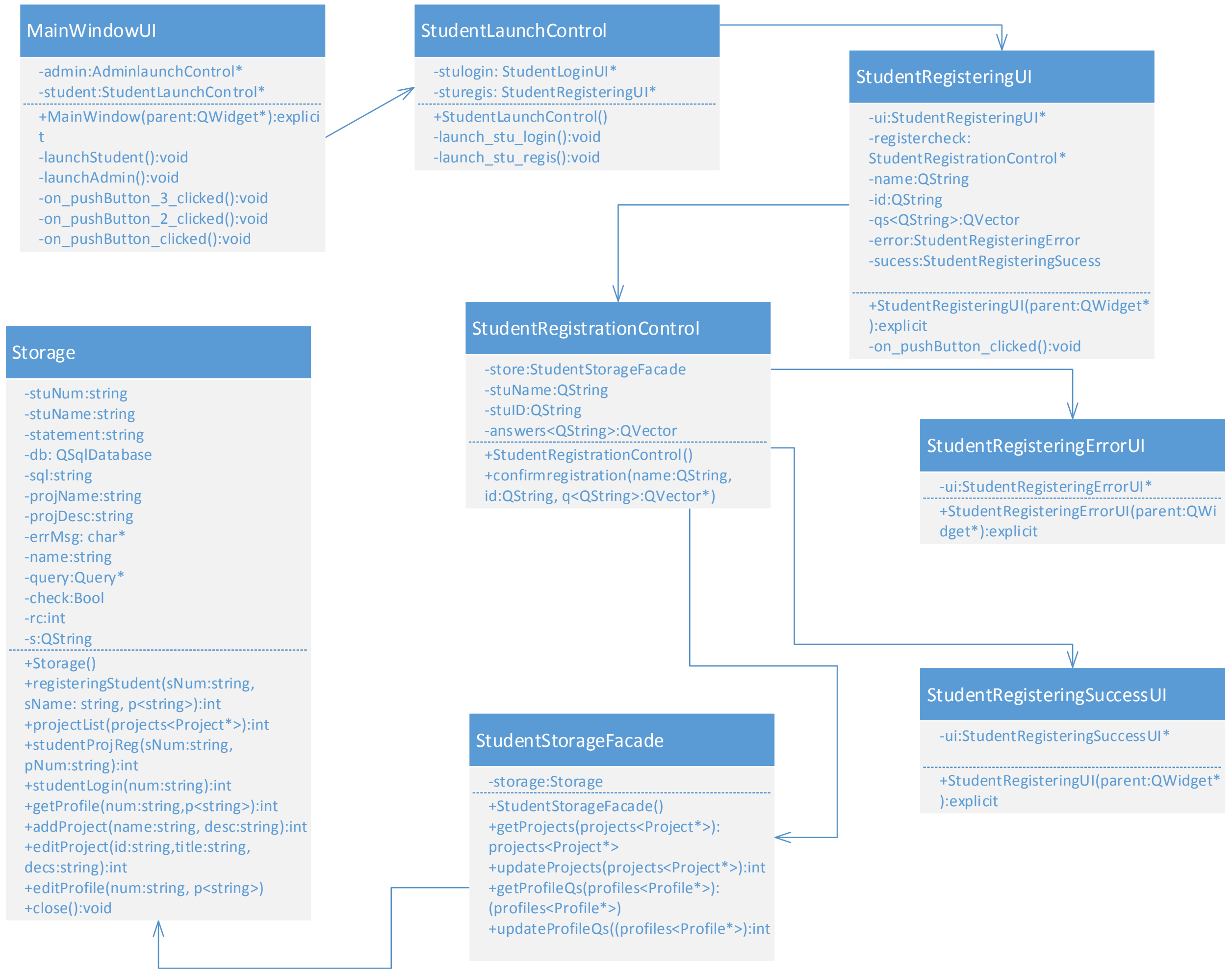
C-1 UML Deployment Diagram



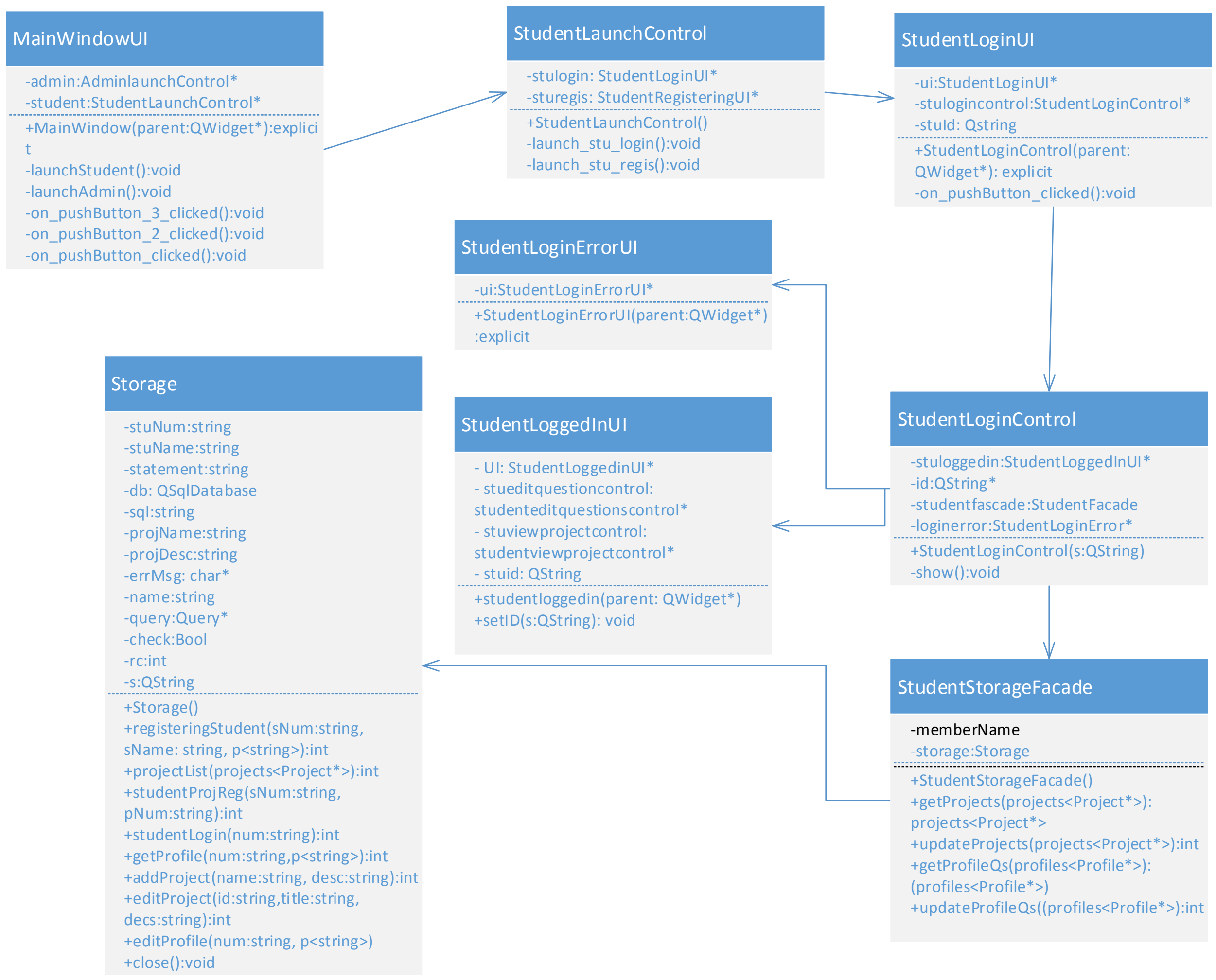
D-1 UML Ball-And-Socket UML Component Diagram



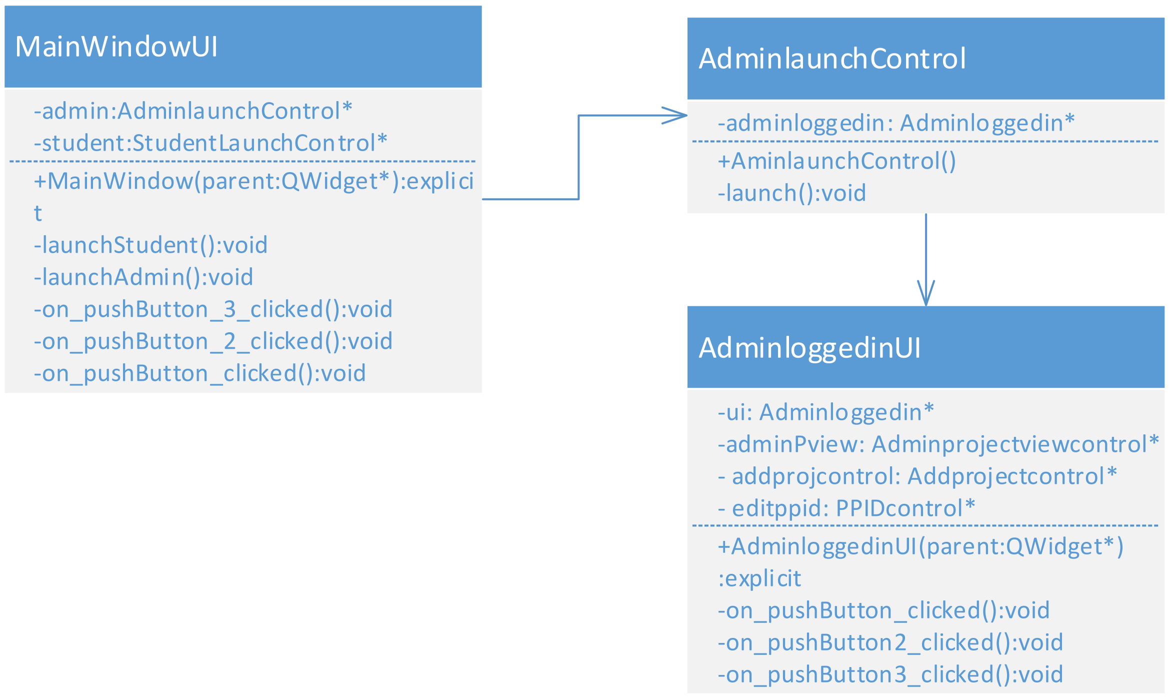
E-1 Student Registering Service UML Diagram


MainWindowUI
-admin:AdminlaunchControl*
-student:StudentLaunchControl*
+MainWindow(parent:QWidget*):explicit
-launchStudent():void
-launchAdmin():void
-on_pushButton_3_clicked():void
-on_pushButton_2_clicked():void
-on_pushButton_clicked():void
StudentLaunchControl
-stulogin: StudentLoginUI*
-sturegis: StudentRegisteringUI*
+StudentLaunchControl()
-launch_stu_login():void
-launch_stu_regis():void
StudentRegisteringUI
-ui:StudentRegisteringUI*
-registercheck: StudentRegistrationControl*
-name:QString
-id:QString
-qs<QString>:QVector
-error:StudentRegisteringError
-sucess:StudentRegisteringSucess
+StudentRegisteringUI(parent:QWidget*):explicit
-on_pushButton_clicked():void
StudentRegistrationControl
-store:StudentStorageFacade
-stuName:QString
-stuID:QString
-answers<QString>:QVector
+StudentRegistrationControl()
+confirmregistration(name:QString, id:QString, q<QString>:QVector*)
StudentRegisteringErrorUI
-ui:StudentRegisteringErrorUI*
+StudentRegisteringErrorUI(parent:QWidget*):explicit
StudentRegisteringSuccessUI
-ui:StudentRegisteringSuccessUI*
+StudentRegisteringUI(parent:QWidget*):explicit
Storage
-stuNum:string
-stuName:string
-statement:string
-db: QSqlDatabase
-sql:string
-projName:string
-projDesc:string
-errMsg: char*
-name:string
-query:Query*
-check:Bool
-rc:int
-s:QString
+Storage()
+registeringStudent(sNum:string, sName: string, p<string>):int
+projectList(projects<Project*>):int
+studentProjReg(sNum:string, pNum:string):int
+studentLogin(num:string):int
+getProfile(num:string,p<string>):int
+addProject(name:string, desc:string):int
+editProject(id:string,title:string, decs:string):int
+editProfile(num:string, p<string>)
+close():void
StudentStorageFacade
-storage:Storage
+StudentStorageFacade()
+getProjects(projects<Project*>): projects<Project*>
+updateProjects(projects<Project*>):int
+getProfileQs(profiles<Profile*>): (profiles<Profile*>)
+updateProfileQs((profiles<Profile*>):int

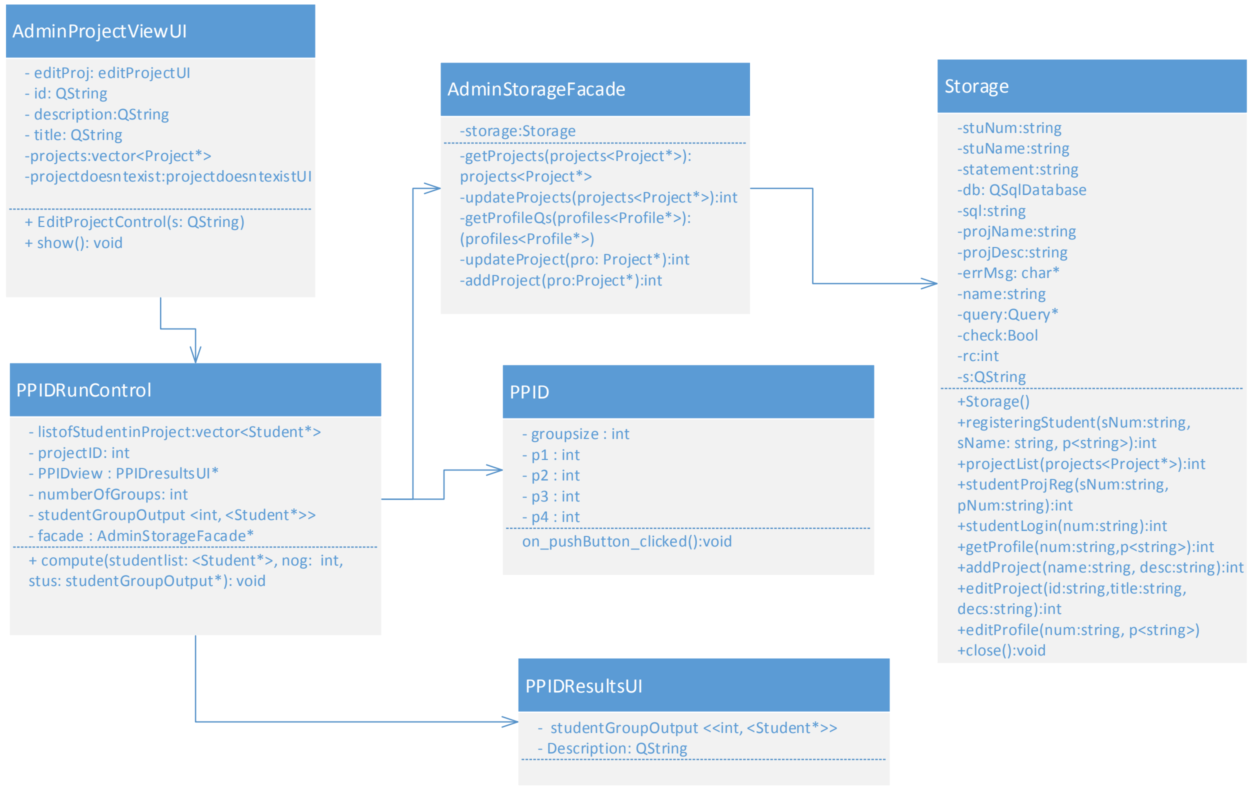
E-2 Student Login Service UML Diagram


MainWindowUI
-admin:AdminlaunchControl*
-student:StudentLaunchControl*
+MainWindow(parent:QWidget*):explicit
-launchStudent():void
-launchAdmin():void
-on_pushButton_3_clicked():void
-on_pushButton_2_clicked():void
-on_pushButton_clicked():void
StudentLaunchControl
-stulogin: StudentLoginUI*
-sturegis: StudentRegisteringUI*
+StudentLaunchControl()
-launch_stu_login():void
-launch_stu_regis():void
StudentLoginUI
-ui:StudentLoginUI*
-stulogincontrol:StudentLoginControl*
-stuId: Qstring
+StudentLoginControl(parent: QWidget*): explicit
-on_pushButton_clicked():void
StudentLoginErrorUI
-ui:StudentLoginErrorUI*
+StudentLoginErrorUI(parent:QWidget*) :explicit
Storage
-stuNum:string
-stuName:string
-statement:string
-db: QSqlDatabase
-sql:string
-projName:string
-projDesc:string
-errMsg: char*
-name:string
-query:Query*
-check:Bool
-rc:int
-s:QString
+Storage()
+registeringStudent(sNum:string, sName: string, p<string>):int
+projectList(projects<Project*>):int
+studentProjReg(sNum:string, pNum:string):int
+studentLogin(num:string):int
+getProfile(num:string,p<string>):int
+addProject(name:string, desc:string):int
+editProject(id:string,title:string, decs:string):int
+editProfile(num:string, p<string>)
+close():void
StudentLoggedInUI
- UI: StudentLoggedinUI*
- stueditquestioncontrol: studenteditquestionscontrol*
- stuviewprojectcontrol: studentviewprojectcontrol*
- stuid: QString
+studentloggedin(parent: QWidget*)
+setID(s:QString): void
StudentLoginControl
-stuloggedin:StudentLoggedInUI*
-id:QString*
-studentfascade:StudentFacade
-loginerror:StudentLoginError*
+StudentLoginControl(s:QString)
-show():void
StudentStorageFacade
-memberName
-storage:Storage
+StudentStorageFacade()
+getProjects(projects<Project*>): projects<Project*>
+updateProjects(projects<Project*>):int
+getProfileQs(profiles<Profile*>): (profiles<Profile*>)
+updateProfileQs((profiles<Profile*>):int

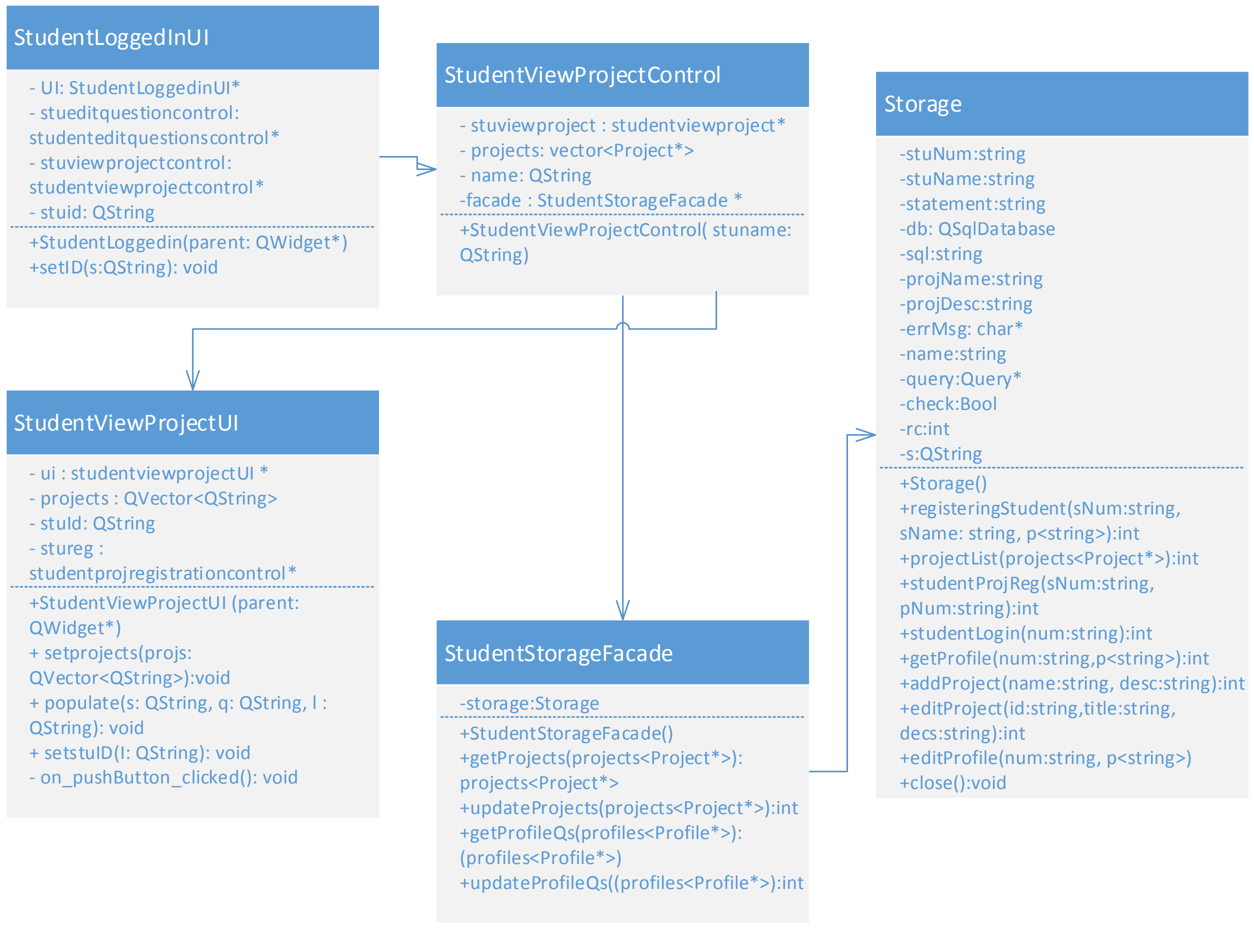
E-3 Admin Login Service UML Diagram


MainWindowUI
-admin:AdminlaunchControl*
-student:StudentLaunchControl*
+MainWindow(parent:QWidget*):explicit
-launchStudent():void
-launchAdmin():void
-on_pushButton_3_clicked():void
-on_pushButton_2_clicked():void
-on_pushButton_clicked():void
AdminlaunchControl
-adminloggedin: Adminloggedin*
+AminlaunchControl()
-launch():void
AdminloggedinUI
-ui: Adminloggedin*
-adminPview: Adminprojectviewcontrol*
- addprojcontrol: Addprojectcontrol*
- editppid: PPIDcontrol*
+AdminloggedinUI(parent:QWidget*) :explicit
-on_pushButton_clicked():void
-on_pushButton2_clicked():void
-on_pushButton3_clicked():void

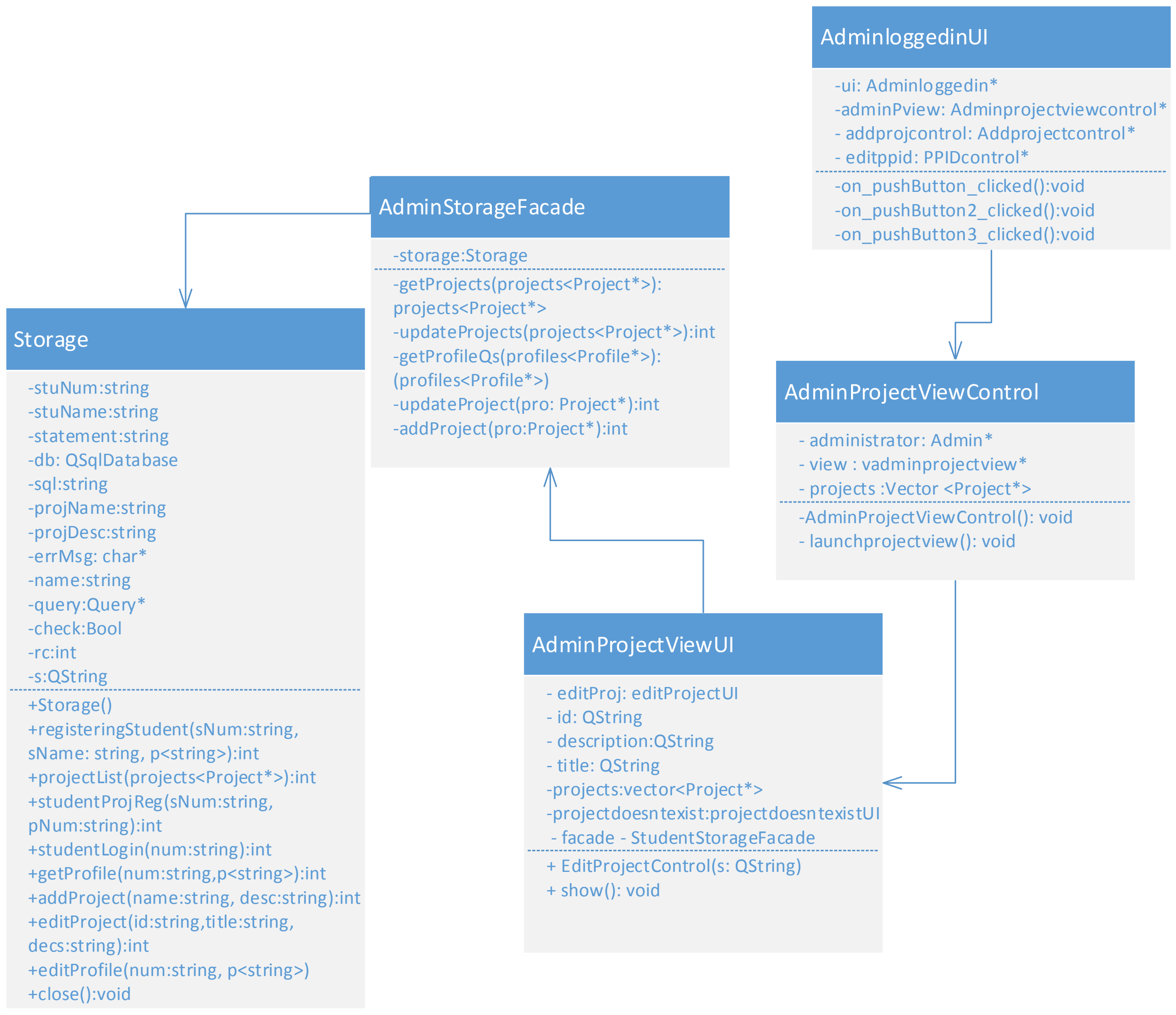
E-4 Retrieving Students in Projects UML Diagram


AdminProjectViewUI
- editProj: editProjectUI
- id: QString
- description:QString
- title: QString
-projects:vector<Project*>
-projectdoesntexist:projectdoesntexistUI
+ EditProjectControl(s: QString)
+ show(): void
AdminStorageFacade
-storage:Storage
-getProjects(projects<Project*>): projects<Project*>
-updateProjects(projects<Project*>):int
-getProfileQs(profiles<Profile*>): (profiles<Profile*>)
-updateProject(pro: Project*):int
-addProject(pro:Project*):int
Storage
-stuNum:string
-stuName:string
-statement:string
-db: QSqlDatabase
-sql:string
-projName:string
-projDesc:string
-errMsg: char*
-name:string
-query:Query*
-check:Bool
-rc:int
-s:QString
+Storage()
+registeringStudent(sNum:string, sName: string, p<string>):int
+projectList(projects<Project*>):int
+studentProjReg(sNum:string, pNum:string):int
+studentLogin(num:string):int
+getProfile(num:string,p<string>):int
+addProject(name:string, desc:string):int
+editProject(id:string,title:string, decs:string):int
+editProfile(num:string, p<string>)
+close():void
PPIDRunControl
- listofStudentinProject:vector<Student*>
- projectID: int
- PPIDview : PPIDresultsUI*
- numberOfGroups: int
- studentGroupOutput <int, <Student*>>
- facade : AdminStorageFacade*
+ compute(studentlist: <Student*>, nog: int, stus: studentGroupOutput*): void
PPID
- groupsize : int
- p1 : int
- p2 : int
- p3 : int
- p4 : int
on_pushButton_clicked():void
PPIDResultsUI
- studentGroupOutput <<int, <Student*>>
- Description: QString

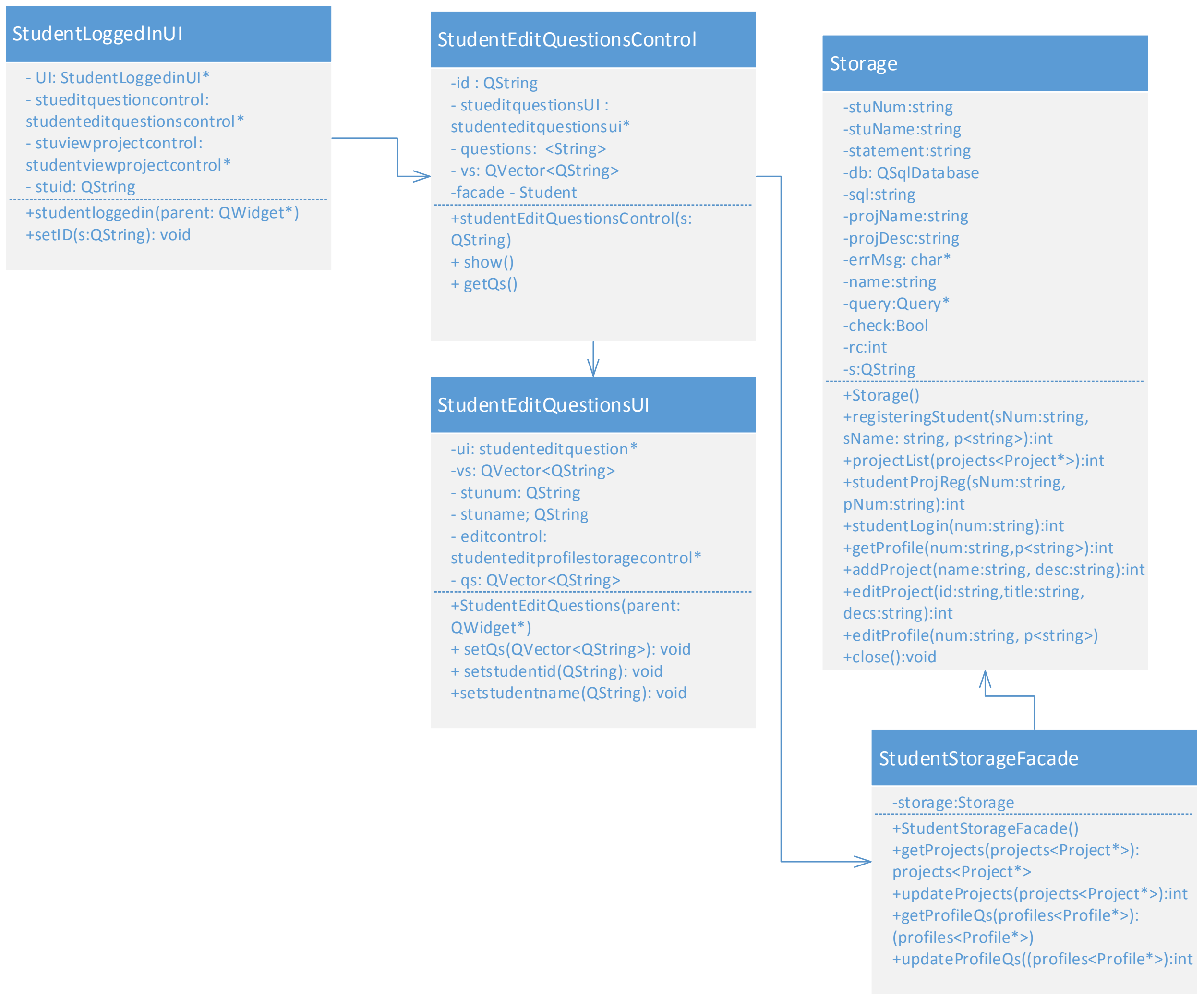
E-5 Retrieving Projects Lists Student Service UML Diagram


StudentLoggedInUI
- UI: StudentLoggedinUI*
- stueditquestioncontrol: studenteditquestionscontrol*
- stuviewprojectcontrol: studentviewprojectcontrol*
- stuid: QString
+StudentLoggedin(parent: QWidget*)
+setID(s:QString): void
StudentViewProjectControl
- stuviewproject : studentviewproject*
- projects: vector<Project*>
- name: QString
-facade : StudentStorageFacade *
+StudentViewProjectControl(stuname: QString)
Storage
-stuNum:string
-stuName:string
-statement:string
-db: QSqlDatabase
-sql:string
-projName:string
-projDesc:string
-errMsg: char*
-name:string
-query:Query*
-check:Bool
-rc:int
-s:QString
+Storage()
+registeringStudent(sNum:string, sName: string, p<string>):int
+projectList(projects<Project*>):int
+studentProjReg(sNum:string, pNum:string):int
+studentLogin(num:string):int
+getProfile(num:string,p<string>):int
+addProject(name:string, desc:string):int
+editProject(id:string,title:string, decs:string):int
+editProfile(num:string, p<string>)
+close():void
StudentViewProjectUI
- ui : studentviewprojectUI *
- projects : QVector<QString>
- stuId: QString
- stureg : studentprojregistrationcontrol*
+StudentViewProjectUI (parent: QWidget*)
+ setprojects(projs: QVector<QString>):void
+ populate(s: QString, q: QString, l : QString): void
+ setstuID(I: QString): void
- on_pushButton_clicked(): void
StudentStorageFacade
-storage:Storage
+StudentStorageFacade()
+getProjects(projects<Project*>): projects<Project*>
+updateProjects(projects<Project*>):int
+getProfileQs(profiles<Profile*>): (profiles<Profile*>)
+updateProfileQs((profiles<Profile*>):int

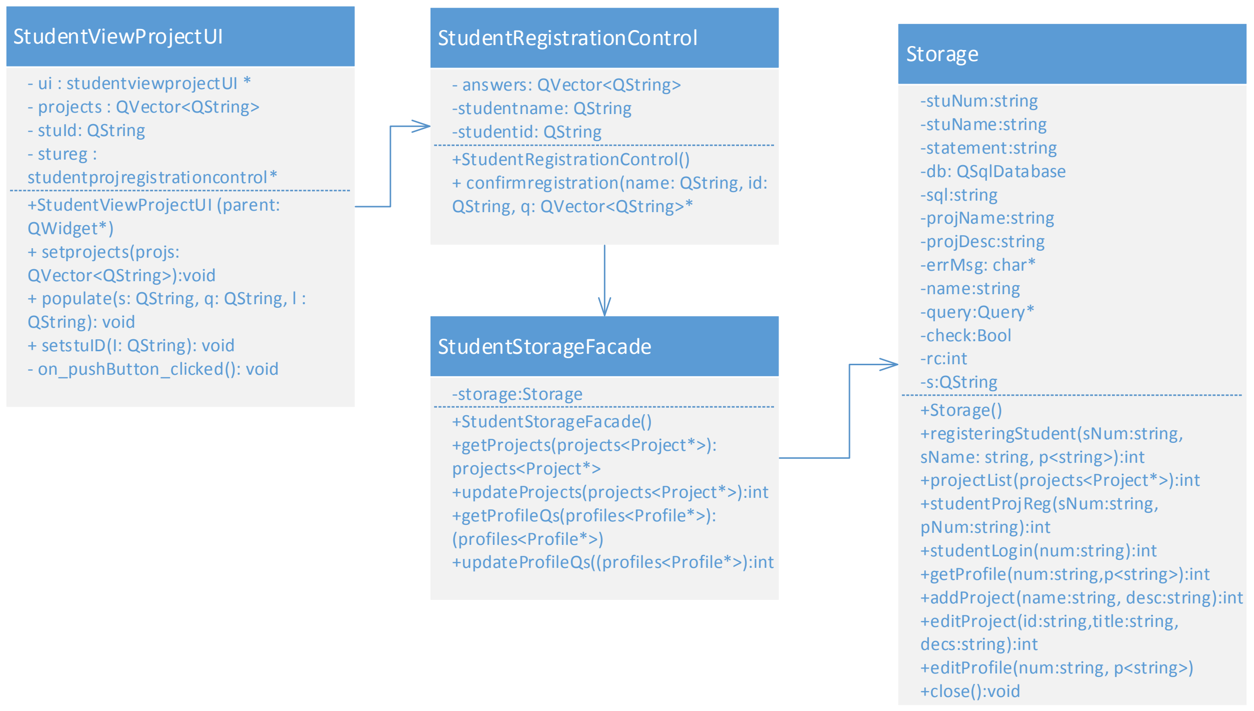
E-6 Retrieving Projects Lists Service UML Diagram


AdminloggedinUI
-ui: Adminloggedin*
-adminPview: Adminprojectviewcontrol*
- addprojcontrol: Addprojectcontrol*
- editppid: PPIDcontrol*
-on_pushButton_clicked():void
-on_pushButton2_clicked():void
-on_pushButton3_clicked():void
AdminStorageFacade
-storage:Storage
-getProjects(projects<Project*>): projects<Project*>
-updateProjects(projects<Project*>):int
-getProfileQs(profiles<Profile*>): (profiles<Profile*>)
-updateProject(pro: Project*):int
-addProject(pro:Project*):int
Storage
-stuNum:string
-stuName:string
-statement:string
-db: QSqlDatabase
-sql:string
-projName:string
-projDesc:string
-errMsg: char*
-name:string
-query:Query*
-check:Bool
-rc:int
-s:QString
+Storage()
+registeringStudent(sNum:string, sName: string, p<string>):int
+projectList(projects<Project*>):int
+studentProjReg(sNum:string, pNum:string):int
+studentLogin(num:string):int
+getProfile(num:string,p<string>):int
+addProject(name:string, desc:string):int
+editProject(id:string,title:string, decs:string):int
+editProfile(num:string, p<string>)
+close():void
AdminProjectViewControl
- administrator: Admin*
- view : vadminprojectview*
- projects :Vector <Project*>
-AdminProjectViewControl(): void
- launchprojectview(): void
AdminProjectViewUI
- editProj: editProjectUI
- id: QString
- description:QString
- title: QString
-projects:vector<Project*>
-projectdoesntexist:projectdoesntexistUI
- facade - StudentStorageFacade
+ EditProjectControl(s: QString)
+ show(): void

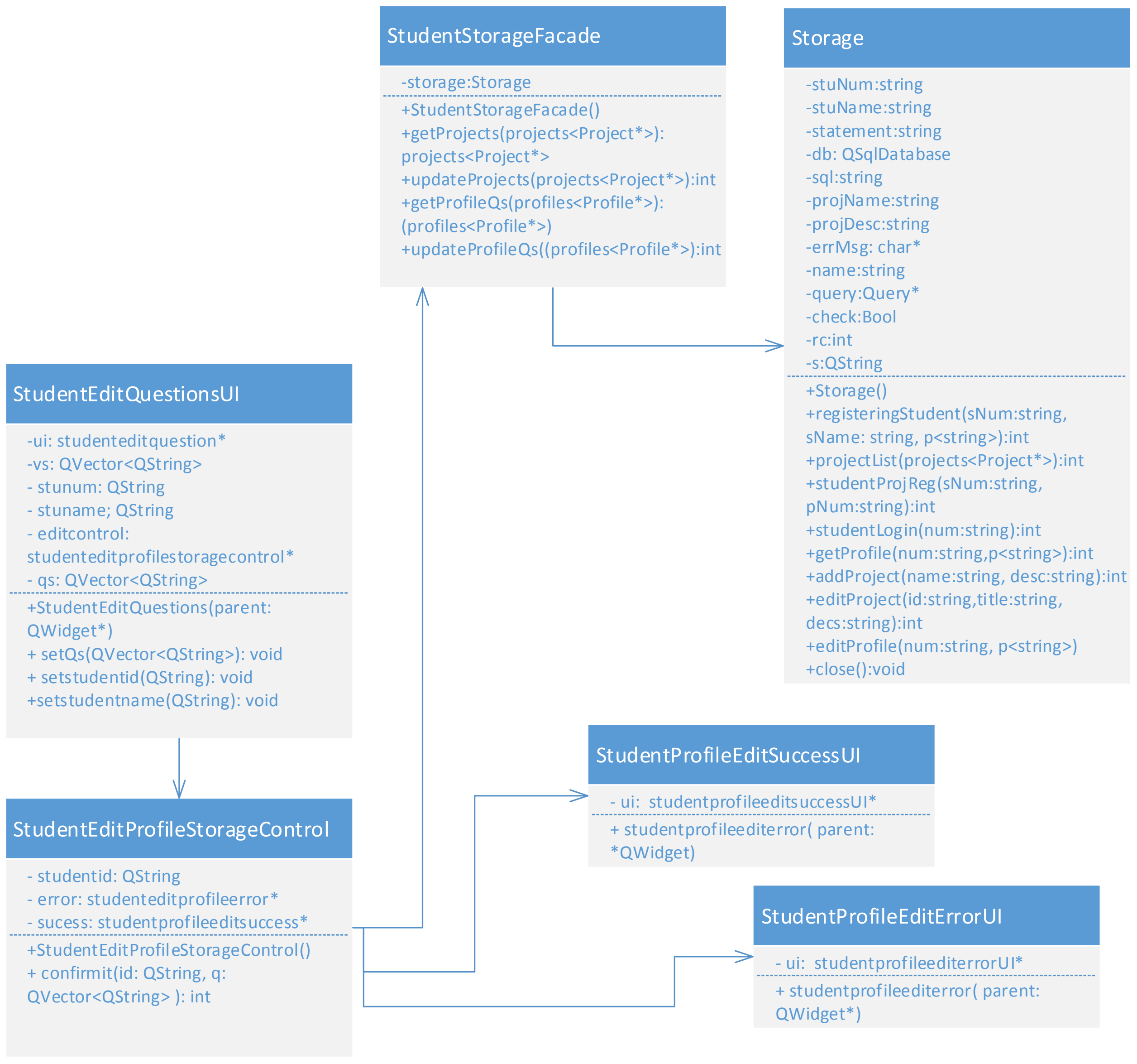
E-7 Retrieving Student Profile Service UML Diagram


StudentLoggedInUI
- UI: StudentLoggedinUI*
- stueditquestioncontrol: studenteditquestionscontrol*
- stuviewprojectcontrol: studentviewprojectcontrol*
- stuid: QString
+studentloggedin(parent: QWidget*)
+setID(s:QString): void
StudentEditQuestionsControl
-id : QString
- stueditquestionsUI : studenteditquestionsui*
- questions: <String>
- vs: QVector<QString>
-facade - Student
+studentEditQuestionsControl(s: QString)
+ show()
+ getQs()
StudentEditQuestionsUI
-ui: studenteditquestion*
-vs: QVector<QString>
- stunum: QString
- stuname; QString
- editcontrol: studenteditprofilestoragecontrol*
- qs: QVector<QString>
+StudentEditQuestions(parent: QWidget*)
+ setQs(QVector<QString>): void
+ setstudentid(QString): void
+setstudentname(QString): void
Storage
-stuNum:string
-stuName:string
-statement:string
-db: QSqlDatabase
-sql:string
-projName:string
-projDesc:string
-errMsg: char*
-name:string
-query:Query*
-check:Bool
-rc:int
-s:QString
+Storage()
+registeringStudent(sNum:string, sName: string, p<string>):int
+projectList(projects<Project*>):int
+studentProjReg(sNum:string, pNum:string):int
+studentLogin(num:string):int
+getProfile(num:string,p<string>):int
+addProject(name:string, desc:string):int
+editProject(id:string,title:string, decs:string):int
+editProfile(num:string, p<string>)
+close():void
StudentStorageFacade
-storage:Storage
+StudentStorageFacade()
+getProjects(projects<Project*>): projects<Project*>
+updateProjects(projects<Project*>):int
+getProfileQs(profiles<Profile*>): (profiles<Profile*>)
+updateProfileQs((profiles<Profile*>):int

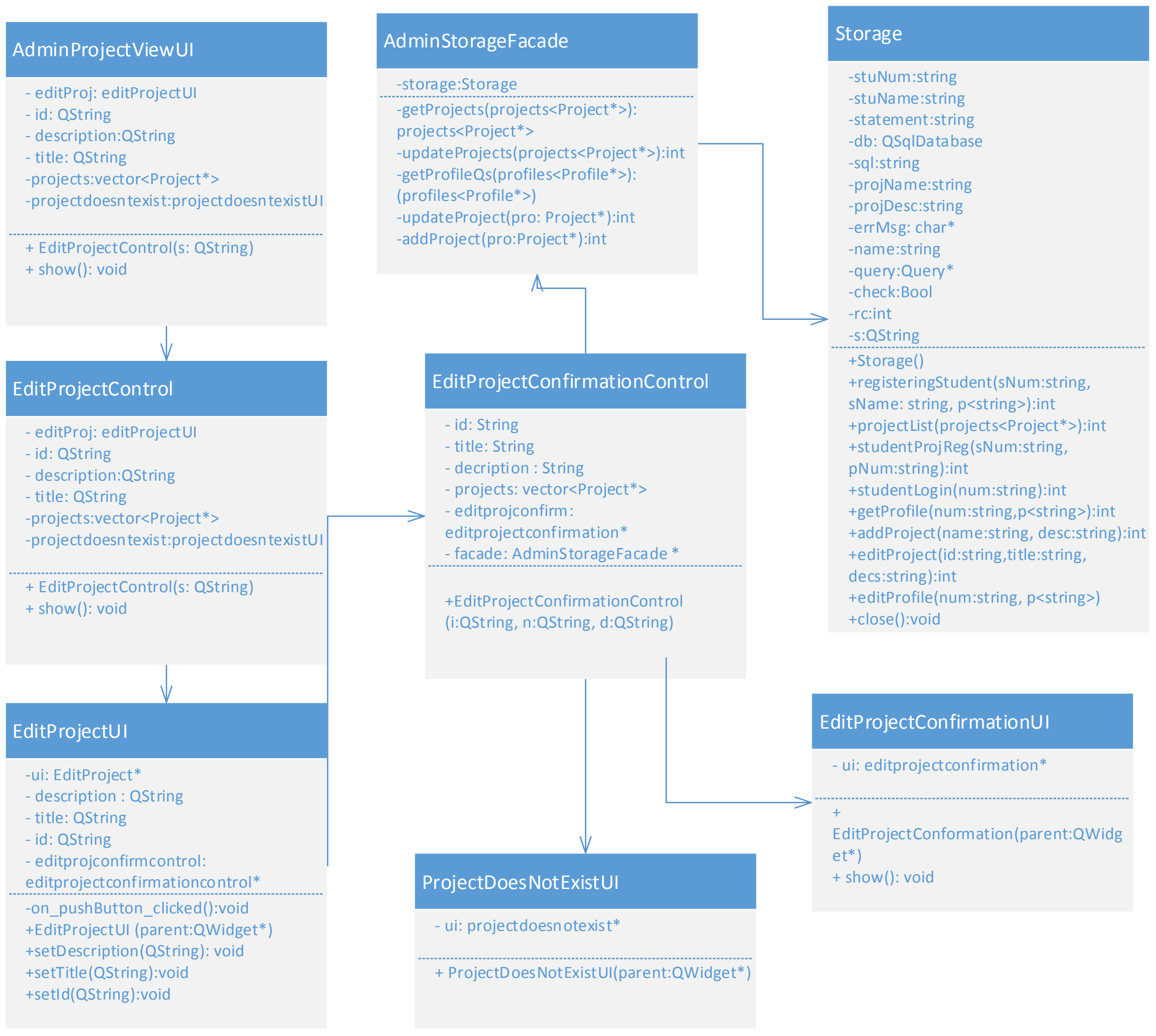
E-8 Student Registering in Project Service UML Diagram


StudentViewProjectUI
- ui : studentviewprojectUI *
- projects : QVector<QString>
- stuId: QString
- stureg : studentprojregistrationcontrol*
+StudentViewProjectUI (parent: QWidget*)
+ setprojects(projs: QVector<QString>):void
+ populate(s: QString, q: QString, l : QString): void
+ setstuID(I: QString): void
- on_pushButton_clicked(): void
StudentRegistrationControl
- answers: QVector<QString>
-studentname: QString
-studentid: QString
+StudentRegistrationControl()
+ confirmregistration(name: QString, id: QString, q: QVector<QString>*
StudentStorageFacade
-storage:Storage
+StudentStorageFacade()
+getProjects(projects<Project*>): projects<Project*>
+updateProjects(projects<Project*>):int
+getProfileQs(profiles<Profile*>): (profiles<Profile*>)
+updateProfileQs((profiles<Profile*>):int
Storage
-stuNum:string
-stuName:string
-statement:string
-db: QSqlDatabase
-sql:string
-projName:string
-projDesc:string
-errMsg: char*
-name:string
-query:Query*
-check:Bool
-rc:int
-s:QString
+Storage()
+registeringStudent(sNum:string, sName: string, p<string>):int
+projectList(projects<Project*>):int
+studentProjReg(sNum:string, pNum:string):int
+studentLogin(num:string):int
+getProfile(num:string,p<string>):int
+addProject(name:string, desc:string):int
+editProject(id:string,title:string, decs:string):int
+editProfile(num:string, p<string>)
+close():void

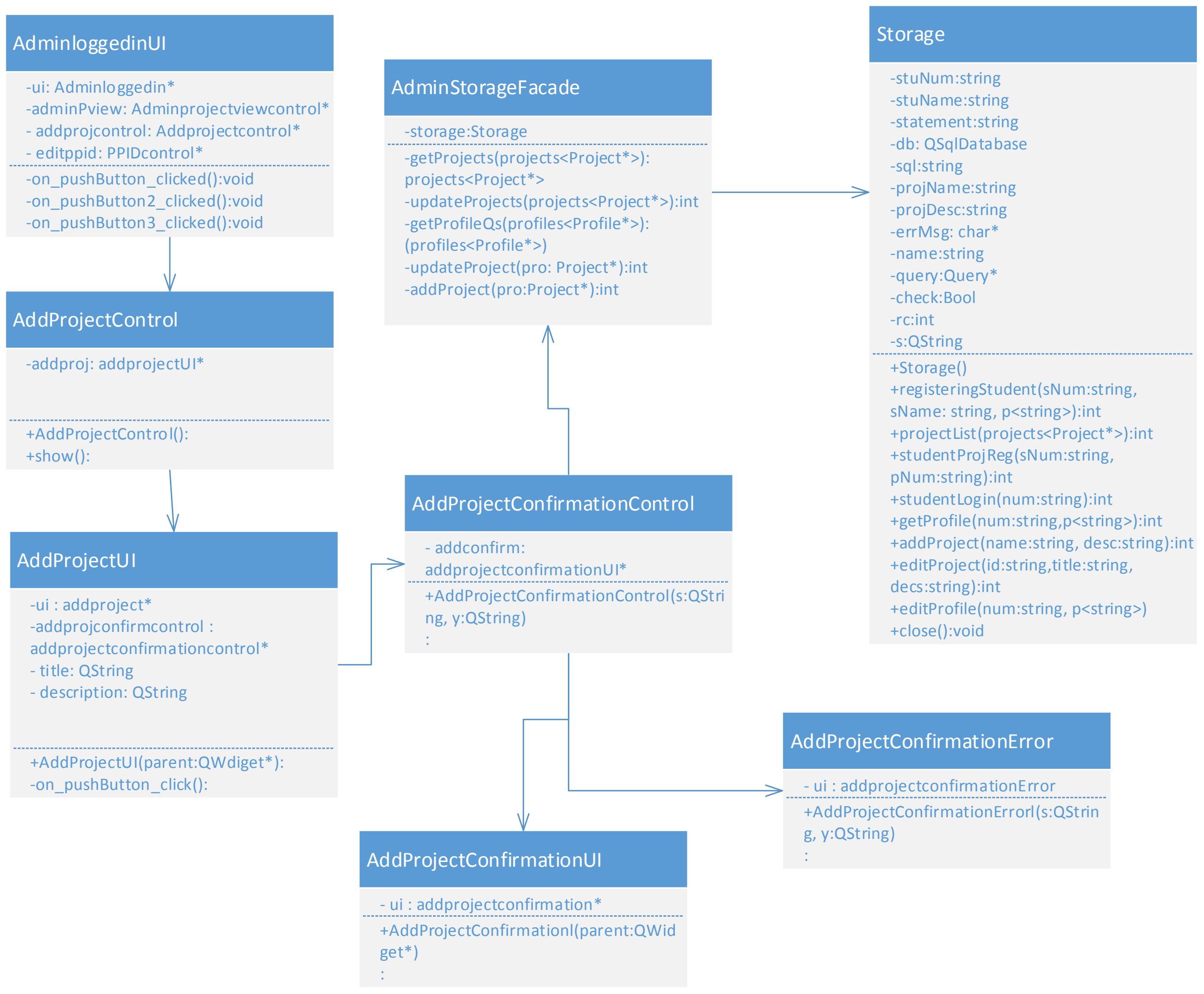
E-9 Student Storage Edit Questions Service UML Diagram


StudentStorageFacade
-storage:Storage
+StudentStorageFacade()
+getProjects(projects<Project*>): projects<Project*>
+updateProjects(projects<Project*>):int
+getProfileQs(profiles<Profile*>): (profiles<Profile*>)
+updateProfileQs((profiles<Profile*>):int
Storage
-stuNum:string
-stuName:string
-statement:string
-db: QSqlDatabase
-sql:string
-projName:string
-projDesc:string
-errMsg: char*
-name:string
-query:Query*
-check:Bool
-rc:int
-s:QString
+Storage()
+registeringStudent(sNum:string, sName: string, p<string>):int
+projectList(projects<Project*>):int
+studentProjReg(sNum:string, pNum:string):int
+studentLogin(num:string):int
+getProfile(num:string,p<string>):int
+addProject(name:string, desc:string):int
+editProject(id:string,title:string, decs:string):int
+editProfile(num:string, p<string>)
+close():void
StudentEditQuestionsUI
-ui: studenteditquestion*
-vs: QVector<QString>
- stunum: QString
- stuname; QString
- editcontrol: studenteditprofilestoragecontrol*
- qs: QVector<QString>
+StudentEditQuestions(parent: QWidget*)
+ setQs(QVector<QString>): void
+ setstudentid(QString): void
+setstudentname(QString): void
StudentEditProfileStorageControl
- studentid: QString
- error: studenteditprofileerror*
- sucess: studentprofileeditsuccess*
+StudentEditProfileStorageControl()
+ confirmit(id: QString, q: QVector<QString>): int
StudentProfileEditSuccessUI
- ui: studentprofileeditsuccessUI*
+ studentprofileediterror(parent: *QWidget)
StudentProfileEditErrorUI
- ui: studentprofileediterrorUI*
+ studentprofileediterror(parent: QWidget*)

E-10 Admin Edit Project Service UML Diagram


AdminProjectViewUI
- editProj: editProjectUI
- id: QString
- description:QString
- title: QString
-projects:vector<Project*>
-projectdoesntexist:projectdoesntexistUI
+ EditProjectControl(s: QString)
+ show(): void
AdminStorageFacade
-storage:Storage
-getProjects(projects<Project*>): projects<Project*>
-updateProjects(projects<Project*>):int
-getProfileQs(profiles<Profile*>): (profiles<Profile*>)
-updateProject(pro: Project*):int
-addProject(pro:Project*):int
Storage
-stuNum:string
-stuName:string
-statement:string
-db: QSqlDatabase
-sql:string
-projName:string
-projDesc:string
-errMsg: char*
-name:string
-query:Query*
-check:Bool
-rc:int
-s:QString
+Storage()
+registeringStudent(sNum:string, sName: string, p<string>):int
+projectList(projects<Project*>):int
+studentProjReg(sNum:string, pNum:string):int
+studentLogin(num:string):int
+getProfile(num:string,p<string>):int
+addProject(name:string, desc:string):int
+editProject(id:string,title:string, decs:string):int
+editProfile(num:string, p<string>)
+close():void
EditProjectControl
- editProj: editProjectUI
- id: QString
- description:QString
- title: QString
-projects:vector<Project*>
-projectdoesntexist:projectdoesntexistUI
+ EditProjectControl(s: QString)
+ show(): void
EditProjectConfirmationControl
- id: String
- title: String
- decription : String
- projects: vector<Project*>
- editprojconfirm: editprojectconfirmation*
- facade: AdminStorageFacade *
+EditProjectConfirmationControl (i:QString, n:QString, d:QString)
EditProjectUI
-ui: EditProject*
- description : QString
- title: QString
- id: QString
- editprojconfirmcontrol: editprojectconfirmationcontrol*
-on_pushButton_clicked():void
+EditProjectUI (parent:QWidget*)
+setDescription(QString): void
+setTitle(QString):void
+setId(QString):void
EditProjectConfirmationUI
- ui: editprojectconfirmation*
+ EditProjectConformation(parent:QWidget*)
+ show(): void
ProjectDoesNotExistUI
- ui: projectdoesnotexist*
+ ProjectDoesNotExistUI(parent:QWidget*)

E-11 Adding New Project Admin Service UML Diagram


AdminloggedinUI
-ui: Adminloggedin*
-adminPview: Adminprojectviewcontrol*
- addprojcontrol: Addprojectcontrol*
- editppid: PPIDcontrol*
-on_pushButton_clicked():void
-on_pushButton2_clicked():void
-on_pushButton3_clicked():void
AddProjectControl
-addproj: addprojectUI*
+AddProjectControl():
+show():
AddProjectUI
-ui : addproject*
-addprojconfirmcontrol : addprojectconfirmationcontrol*
- title: QString
- description: QString
+AddProjectUI(parent:QWdiget*):
-on_pushButton_click():
AdminStorageFacade
-storage:Storage
-getProjects(projects<Project*>): projects<Project*>
-updateProjects(projects<Project*>):int
-getProfileQs(profiles<Profile*>): (profiles<Profile*>)
-updateProject(pro: Project*):int
-addProject(pro:Project*):int
AddProjectConfirmationControl
- addconfirm: addprojectconfirmationUI*
+AddProjectConfirmationControl(s:QString, y:QString)
:
AddProjectConfirmationUI
- ui : addprojectconfirmation*
+AddProjectConfirmationl(parent:QWidget*)
:
Storage
-stuNum:string
-stuName:string
-statement:string
-db: QSqlDatabase
-sql:string
-projName:string
-projDesc:string
-errMsg: char*
-name:string
-query:Query*
-check:Bool
-rc:int
-s:QString
+Storage()
+registeringStudent(sNum:string, sName: string, p<string>):int
+projectList(projects<Project*>):int
+studentProjReg(sNum:string, pNum:string):int
+studentLogin(num:string):int
+getProfile(num:string,p<string>):int
+addProject(name:string, desc:string):int
+editProject(id:string,title:string, decs:string):int
+editProfile(num:string, p<string>)
+close():void
AddProjectConfirmationError
- ui : addprojectconfirmationError
+AddProjectConfirmationErrorl(s:QString, y:QString)
: