cuPID System Design Document

Team Tech Support
Guelor Emanuel
Derek Felson
Margarita Otochkina

Submitted to
Dr. Christine Laurendeau
for the course COMP 3004

Due date
November 23, 2015

Table of contents

Introduction

The Carleton University Project Partner Identifier (cuPID) system is designed to make it easier to match students into good teams for group project work. The cuPID system has several features that work together to allow good team matches to be identified:

1. An administrator can create projects and edit their settings such as team size, name, and description.
2. A student can join any number of projects and create a profile for themselves, consisting of at least twelve qualifications relevant to team compatibility, such as grades, habits, skillsets, and personality. Along with each qualification, the profile contains a corresponding preference regarding a range of values the student would find acceptable in their team members. Students can later choose to edit their profile.
3. The administrator can select an option to compute the best teams for a project; the cuPID system will use the information in the profiles of students registered in that project to identify the most compatible teams. The administrator can then choose to view the results in either a summary form, which contains just a list of teams and which students are in them, or in detailed form, which contains specifics on why the teams were put together as they were.

To make administration easier when dealing with large class sizes, the system also allows guests to use the system without logging in, for the purpose of creating a new student (or administrator) account. Creating a student profile is part of the student account creation process.

This document is organized into four main sections which describe our various design choices.

The first section, subsystem decomposition, covers our initial prototype design, how it breaks down into subsystems, how our new design will break down into subsystems, and how our design evolved from the initial prototype to the current version.

The second section, design strategies, details and justifies the software architecture we are using and how the system is deployed in terms of subsystems to runtime components to physical computers. The second section also covers the design patterns we used in the current design and the strategy we chose for persistent data storage.

The third section, subsystem services, looks more deeply into the dependencies between subsystems by identifying and describing all the services offered by the different parts the system.

The fourth section, class interfaces, contains UML class diagrams for the classes responsible for offering the subsystem services described in the third section.

Subsystem decomposition

In phase 1 we completed a prototype of the cuPID system. In this part of the project (phase 2) we improve upon that design. This section of the document describes the subsystems in the phase 1 prototype, the subsystems in the phase #2 design, and how the design has changed from phase 1 to phase 2.

Phase 1 prototype decomposition

Our prototype included the student features, the admin features, the user management parts (account creation, student profile creation), the GUI and the database. It included all parts of the system except the algorithm for finding the best teams. The following table shows how we broke the phase 1 prototype down into subsystems.

Table 1: Prototype subsystems

Subsystem	Description
Student Features	The Student Features subsystem is responsible for allowing students to edit their profile, showing them a list of projects they are in and a list of projects they can join, and allowing them to join existing projects.
Admin Features	The Admin Features subsystem is responsible for showing administrators a list of existing projects, allowing them to view and edit project settings, and allowing them to create projects.
Storage	The Storage subsystem is responsible for allowing the other subsystems to save and retrieve information from persistent storage.
User Management	The User Management subsystem is responsible for handling login requests and creating student and admin accounts.

These four subsystems had many dependencies on each other, as shown in the following UML component diagram:

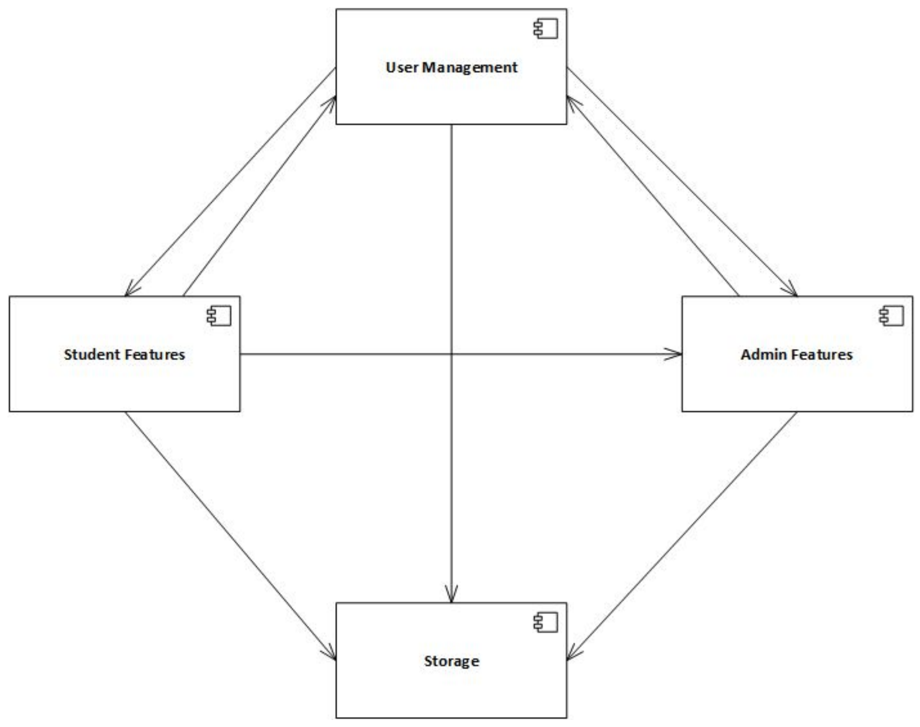


Figure 1: UML component diagram for phase 1 subsystem dependencies

The contents of those subsystems and the nature of those dependencies is shown in the following figure::

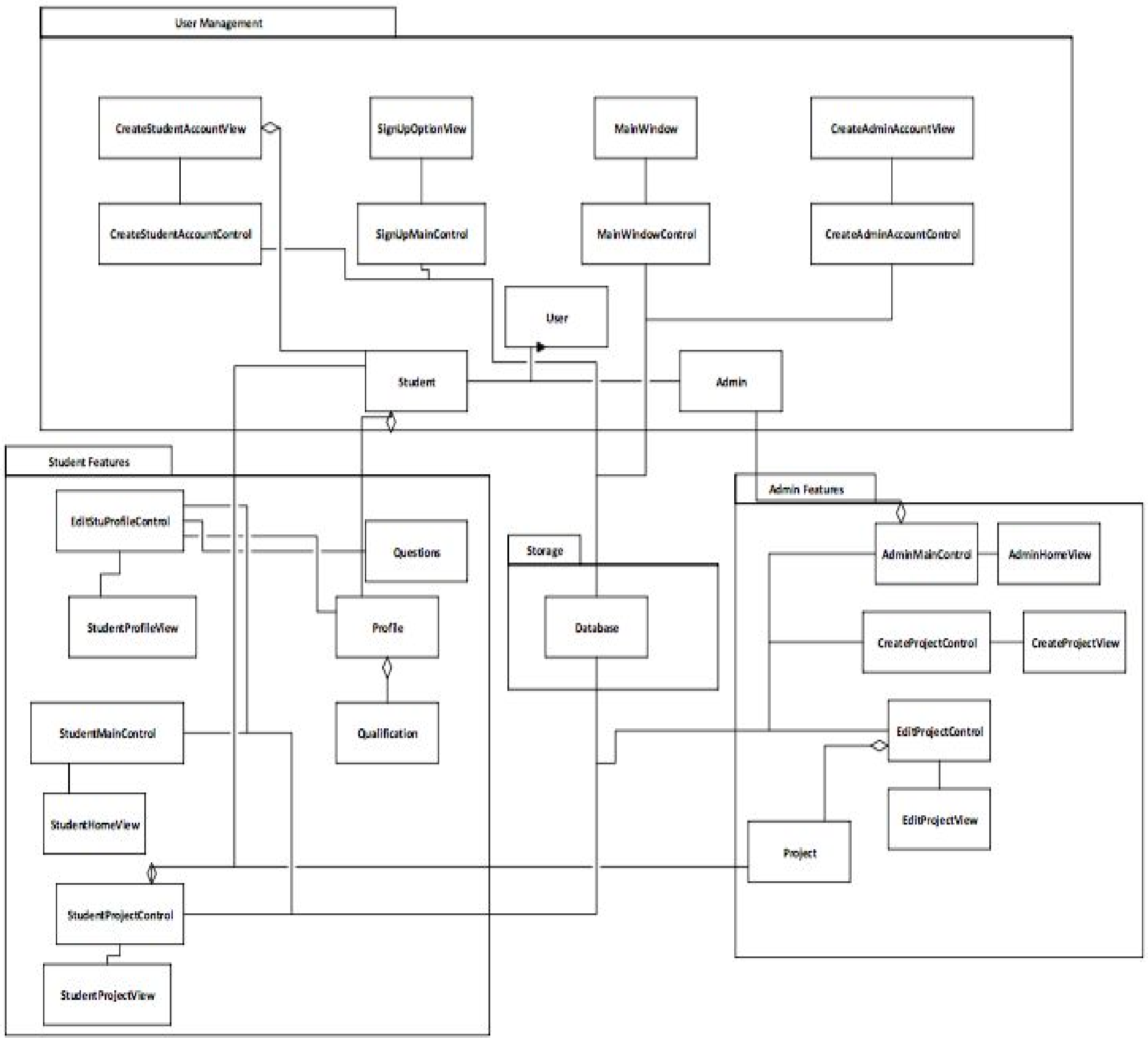


Figure 2: Phase 1 Prototype Subsystem Decomposition

The remainder of the phase 1 prototype section analyzes the subsystems and their dependencies on one another. It will also show traceability.

Student Features Subsystem

The Student Feature subsystem contains 9 classes:

- StudentMainControl

- StudentHomeView
- EditStuProfileControl
- StudentProfileView
- StudentProjectControl
- StudentProjectView
- Profile
- Qualification
- Questions

Coupling

There are 7 dependencies from the Student Features subsystem to other subsystems.

This subsystem has 2 connections to the Admin Features subsystem:

- StudentProjectControl uses Project
- StudentProjectView uses Project

This subsystem has 3 connections to the Storage subsystem:

- StudentProjectControl uses Database
- StudentMainControl uses Database
- Profile uses Database

This subsystem has 2 connections to the User Management subsystem:

- StudentMainControl uses Student
- StudentProjectControl uses Student

Cohesion

There are 11 dependencies within the Student Features subsystem.

- StudentProjectControl and StudentProjectView (bidirectional)
- StudentMainControl and StudentHomeView (bidirectional)
- EditStuProfileControl and StudentProfileView (bidirectional)
- StudentMainControl uses StudentProjectControl
- StudentMainControl uses EditStuProfileControl
- EditStuProfileControl uses Questions
- EditStuProfileControl uses Profile
- Profile uses Qualification

Traceability

The Student Features subsystem traces back to:

- F-01 Students can add themselves to an existing project
- F-02 Students can edit their own profile

- F-03 Students can view a list of existing projects
- UC-02 Edit Profile
- UC-03 Join Project
- UC-13 View Existing Projects
- UC-16 Profile Error
- UC-17 Invalid Input Error
- UC-19 Repository Error
- EO-Qualification
- EO-Profile
- EO-Project
- EO-Student
- S-04 Edit Profile
- S-05 Join Project
- S-09 View Existing Projects
- S-11 Invalid Input Error
- S-13 Profile Error
- S-14 Repository Error

The traceability for each class:

Table 2: Traceability for Student Feature Subsystem

Class	Traceability
EditStuProfileControl	F-02, UC-02, UC-16, UC-17, UC-19, S-04, S-11, S-13, S-14
StudentProfileView	F-02, UC-02, UC-16, UC-17, UC-19, S-04, S-11, S-13, S-14
StudentMainControl	F-03, UC-13, UC-19, EO-Student, S-09, S-14
StudentHomeView	F-03, UC-13, UC-19, S-09, S-14
StudentProjectControl	F-01, UC-03, UC-19, EO-Project, EO-Student, S-05, S-14
StudentProjectView	F-01, UC-03, EO-Project, UC-19, S-05, S-14
Profile	F-02, UC-02, UC-19, EO-Profile, S-14
Qualification	F-02, UC-02, EO-Qualification
Questions	F-02, UC-02

Admin Features Subsystem

The Admin Feature Subsystem contains 7 classes:

- AdminMainControl
- AdminHomeView
- CreateProjectControl

- CreateProjectView
- EditProjectControl
- EditProjectView
- Project

Coupling

There are 4 dependencies from the Admin Features subsystem to other subsystems.

The Admin Features subsystem has 2 connections to the User Management subsystem:

- AdminMainControl uses Admin
- Project uses Student

The Admin Features subsystem has 2 connections to the Storage subsystem:

- AdminMainControl uses Database
- Project uses Database

The Admin Features subsystem has 0 connections to the Student Features subsystem.

Cohesion

There are 11 dependencies within the Admin Features subsystem:

- AdminMainControl and AdminHomeView (bidirectional)
- CreateProjectControl and CreateProjectView (bidirectional)
- EditProjectControl and EditProjectView (bidirectional)
- AdminMainControl uses CreateProjectControl
- AdminMainControl uses EditProjectControl
- CreateProjectControl uses Project
- EditProjectControl uses Project
- EditProjectView uses Project

Traceability

The Admin Features subsystem traces back to:

- F-04 Administrators can edit project settings
- F-06 Administrators can view the list of existing projects
- F-07 Administrators can create new projects
- UC-04 Manage Project
- UC-09 Create Project
- UC-10 Edit Project
- UC-13 View Existing Projects
- UC-17 Invalid Input Error
- UC-19 Repository Error
- EO Admin

- EO Project
- S-07 Create Project
- S-08 Edit Project
- S-09 View Existing Projects
- S-11 Invalid Input Error
- S-14 Repository Error

The traceability for each class:

Table 3: Traceability for Admin Feature Subsystem

Class	Traceability
AdminMainControl	F-06, UC-04, UC-13, UC-19, EO-Admin, S-09, S-14
AdminHomeView	F-06, UC-04, UC-13, UC-19, S-09, S-14
CreateProjectControl	F-07, UC-04, UC-09, EO-Project, S-07, S-11, S-14
CreateProjectView	F-07, UC-04, UC-09, S-07, S-11, S-14
EditProjectControl	F-04, UC-04, UC-10, EO-Project, S-08, S-11, S-14
EditProjectView	F-04, UC-04, UC-10, EO-Project, S-08, S-11, S-14
Project	F-04, F-06, F-07, UC-04, UC-09, UC-10, UC-19, EO-Project, S-07, S-08, S-14

Storage Subsystem

The Storage Subsystem contains 1 class:

- Database

Coupling

There are 0 dependencies from the Storage subsystem to other subsystems.

Cohesion

There are 0 dependencies within the Storage subsystem.

Traceability

The Storage Features subsystem traces back to:

- NF-07 The number of all possible failures like system crash or inability to save changed data during one-day usage must be less than 2
- NF-08 The system must be able to process invalid user input avoiding system crash or corrupting data
- NF-09 If the system fails during usage it must be able to load the last saved settings

- NF-10 The system must be able to provide a stable and reliable connection between client and server
- NF-10-01 The system must be able to notify user if during usage clients gets disconnected and attempt to connect
- NF-11 The system must be able to start in less than a minute on a computer with at least 2GB RAM or 4GB RAM if it is running on virtual machine
- NF-12 The system must be able to store efficiently at least 1,000 student profiles without noticeably slowing down
- NF-14 The system must perform profile/project information update in less than 10 seconds
- NF-15 The system must be available all the time without any downtime
- NF-29 We will not be liable/responsible for any system failures or damages
- NF-30 The system must comply by the University Collection of Personal Information privacy policy
- NF-31 There won't be any specific components or third party libraries incorporated into the system in order to avoid any form of royalties fee. Any library used will provided by the operating system itself
- UC-17 Invalid Input Error
- UC-19 Repository Error
- S-11 Invalid Input Error
- S-14 Repository Error

The traceability for each class:

Table 4: Traceability for Database Subsystem

Class	Traceability
Database	NF-07, NF-08, NF-09, NF-10, NF-10-01, NF-12, NF-13, NF-14, NF-15

User Management Subsystem

The User Management Subsystem contains 11 classes:

- SignUpMainControl
- SignUpOptionView
- CreateAdminAccountControl
- CreateAdminAccountView
- CreateStudentAccountControl
- CreateStudentAccountView
- MainWindowControl
- MainWindow
- Student
- User
- Admin

Coupling

There are 8 dependencies from the User Management subsystem to other subsystems.

The User Management subsystem has 3 connections to the Student Features subsystem:

- CreateStudentAccountControl uses EditStuProfileControl
- Student uses Profile
- MainWindowControl uses StudentMainControl

The User Management subsystem has 1 connection to the Admin Features subsystem:

- MainWindowControl uses AdminMainControl

The User Management subsystem has 4 connections to the Storage subsystem:

- MainWindowControl uses Database
- CreateStudentAccountControl uses Database
- Student uses Database
- Admin uses Database

Cohesion

There are 17 dependencies within the User Management subsystem.

- SignUpMainControl and SignUpOptionView (bidirectional)
- MainWindowControl and MainWindow (bidirectional)
- CreateAdminAccountControl and CreateAdminAccountView (bidirectional)
- CreateStudentAccountControl and CreateStudentAccountView (bidirectional)
- Student extends User
- Admin extends User
- CreateAdminControl uses Admin
- SignUpMainControl uses CreateAdminAccountControl
- SignUpMainControl uses CreateStudentAccountControl
- CreateStudentAccountControl uses Student
- MainWindowControl uses Student
- MainWindowControl uses Admin
- MainWindowControl uses SignUpMainControl

Traceability

The User Management Subsystems traces back to:

- F-10 Guest users must be able to create a new account
- F-11 Guest users must be able to create Student account
- F-12 Guest users must be able to create Administrator account
- NF-01 The system navigation and interface should be intuitive so that average number of user mistakes is less than for each high level use case

- NF-02 Profile qualifications and preferences should be easy to determine so that 90% of students can create a profile in less than 10 minutes
- NF-04 90% of surveyed users should agree that the look and feel of the system including fonts, style, colour, etc.-is professional, like the UI of commercial products
- NF-06 Keyboard shortcuts must be provided where applicable so that interface can be operated without a mouse
- UC-17 Invalid Input Error
- UC-19 Repository Error
- EO Admin
- EO User
- EO Student
- S-11 Invalid Input Error
- S-14 Repository Error

The traceability for each class:

Table 5: Traceability for User subsystem

Class	Traceability
SignUpMainControl	F-10, F-11, F-12, UC-01, EO-Admin, EO-Student, EO-Guest,S-01, S-02, S-03, S-11, S-14
SignUpOptionView	F-10, F-11, F-12, UC-01, EO-Admin, EO-Student, EO-Guest,S-01, S-02, S-03, S-11, S-14
MainWindowControl	F-10, F-11, F-12, UC-01, EO-Admin, EO-Student, EO-Guest,S-01, S-02, S-03, S-11, S-14
MainWindow	F-10, F-11, F-12, UC-01, EO-Admin, EO-Student, EO-Guest,S-01, S-02, S-03, S-11, S-14
CreateAdminAccountControl	F-10, F-12, UC-01, UC-06, EO-Admin, S-01, S-02, S-11, S-14
CreateAdminAccountView	F-10, F-12, UC-01, UC-06, EO-Admin, S-01, S-02, S-11, S-14
CreateStudentAccountControl	F-10, F-11, UC-01, UC-07, EO-Student, S-01, S-03, S-11, S-14
CreateStudentAccountView	F-10, F-11, UC-01, UC-07, EO-Student, S-01, S-03, S-11, S-14
CreateAdminControl	F-10, F-12, UC-01, UC-06, EO-Admin, S-01, S-02, S-11, S-14

SignUpMainControl	F-10, F-11, F-12, UC-01, EO-Admin, EO-Student, EO-Guest,S-01, S-02, S-03, S-11, S-14

Classes not included in the system

Certain classes were accidentally submitted in D2 but were not actually used in the operation of the prototype:

- class Model
- class Qualifications
- class AccountCreationControl
- class AdminProfileView
- class ProjectView
- class Team
- class Signupview
- class signusetprofileview
- class studentsignupView
- class StudentProfileViewPartTwo
- class Guest
- class AdminHomePageView

System decomposition

This section describes the subsystems in our phase 2 design. It describes every subsystem, shows which classes are in which subsystem, and shows all dependencies between subsystems. The design has high cohesion and low coupling, though most of the explanation for that will take place in the next section, Design Evolution.

Table 6: Phase 2 subsystems

Subsystem	Description
Student Management	The Student Management subsystem is responsible for allowing students to edit their profile, showing them a list of projects they are in and a list of projects they can join, and allowing them to join existing projects.
Admin Management	The Admin Management subsystem is responsible for showing administrators a list of existing projects, allowing them to view and edit project settings, and allowing them to create projects.
User Management	The User Management subsystem is responsible for handling login requests and creating student and admin accounts.
Algorithm Management	The Algorithm Management subsystem is responsible for matching the students in a project to the best possible teams, and allowing the administrator to view summary and detailed information about why students were grouped together.
Storage	The Storage subsystem is an abstraction of the Database subsystem responsible for offering the other subsystems an easy way to interact with objects in persistent storage. It also maintains a consistent cache of objects that have been retrieved from persistent storage.
Database	The Database subsystem is responsible for implementing direct communication with the SQLite database (or, theoretically, any other type of persistent storage system).

The changes from phase 1 include the addition of the algorithm management subsystem, and the separation of the storage subsystem into Storage and Database.

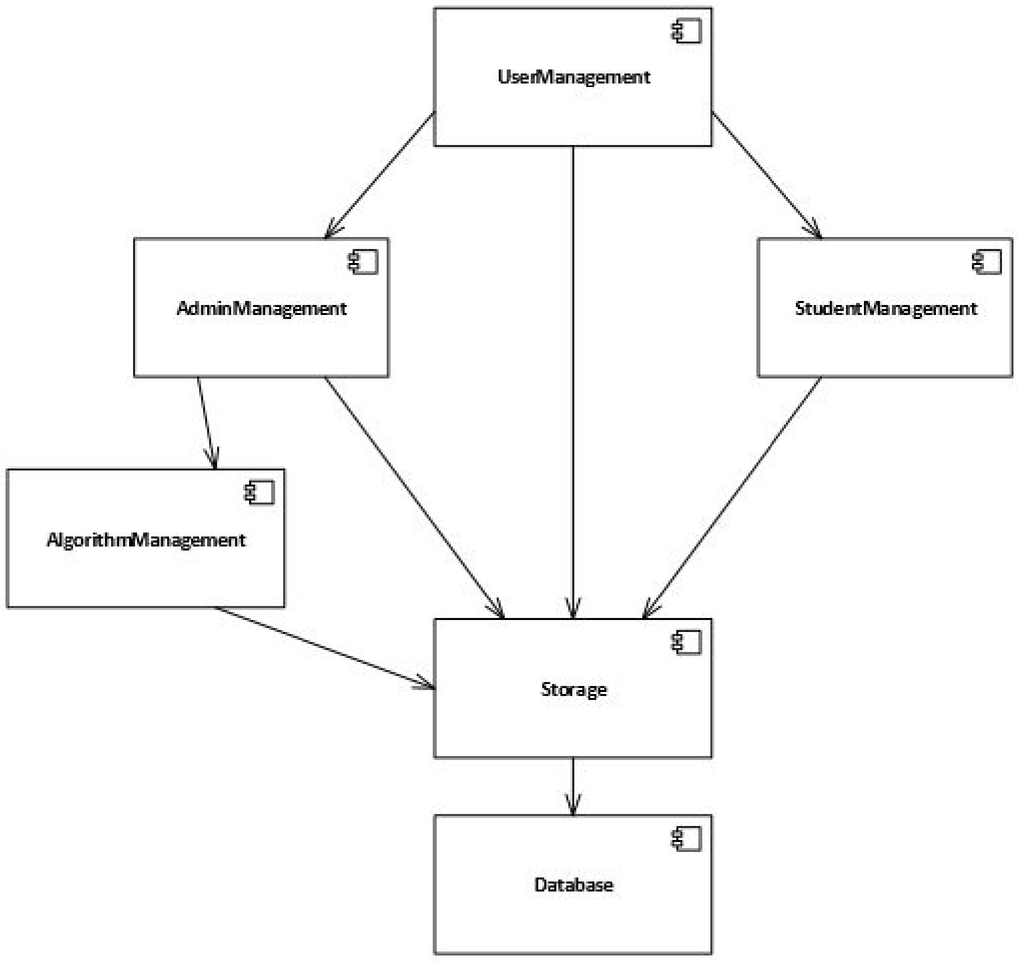


Figure 3: Phase 2 UML Component Diagram

The most important dependencies are the ones going to storage and the one from storage to database. The other dependencies (UserManagement to StudentManagement and AdminManagement, and AdminManagement to AlgorithmManagement) are much weaker; those dependencies largely exist because of the need to navigate from one part of the user interface to another. Those are actually "uses" relationships, though they are important enough to have services defined for them, so we wanted to show them here; those subsystems are not completely independent.

The following figure shows the classes in the User Management subsystem:

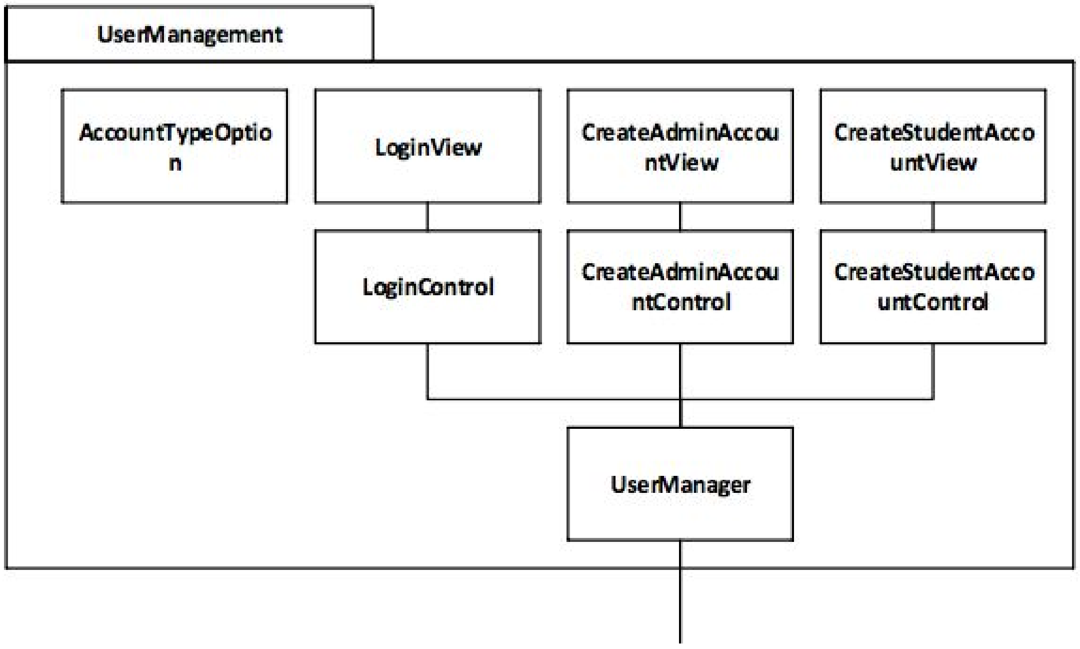


Figure 4: Phase 2 Subsystem Decomposition -- UserManagement Subsystem

Note that UserManager is a singleton and facade class. It is used by the other control objects in the subsystem, but it does not have an aggregation relationship with them. The association from UserManager going down in this diagram is actually a uses relationship with another singleton/facade class, StorageManager, and it is unidirectional from UserManager to StorageManager. We chose to represent these relationships in the diagram, despite being "uses" relationships because otherwise it is unclear how our subsystems interact.

Traceability

The UserManagement Subsystems traces back to:

- F-11 Guest users must be able to create Student account
- F-12 Guest users must be able to create Administrator account
- NF-01 The system navigation and interface should be intuitive so that average number of user mistakes is less than for each high level use case
- NF-02 Profile qualifications and preferences should be easy to determine so that 90% of students can create a profile in less than 10 minutes
- NF-04 90% of surveyed users should agree that the look and feel of the system including fonts, style, colour, etc.-is professional, like the UI of commercial products
- NF-06 Keyboard shortcuts must be provided where applicable so that interface can be operated without a mouse
- UC-17 Invalid Input Error
- UC-19 Repository Error
- EO Admin
- EO User

- S-11 Invalid Input Error
- S-14 Repository Error

The traceability for each class:

Table 7: Traceability for UserManagement subsystem

Class	Traceability
AccountTypeOption	F-10, F-11, F-12, UC-01, EO-Admin, EO-Student, S-01, S-02, S-03, S-11, S-14
LoginControl	F-10, F-11, F-12, UC-01, EO-Admin, EO-Student, S-01, S-02, S-03, S-11, S-14
LoginView	F-10, F-11, F-12, UC-01, EO-Admin, EO-Student,S-01, S-02, S-03, S-11, S-14
CreateAdminAccountControl	F-10, F-12, UC-01, UC-06, EO-Admin, S-01, S-02, S-11, S-14
CreateAdminAccountView	F-10, F-12, UC-01, UC-06, EO-Admin, S-01, S-02, S-11, S-14
CreateStudentAccountControl	F-10, F-11, UC-01, UC-07, EO-Student, S-01, S-03, S-11, S-14
CreateStudentAccountView	F-10, F-11, UC-01, UC-07, EO-Student, S-01, S-03, S-11, S-14
UserManager	EO-Student, EO-Admin,S-01, S-02,S-04

The next subsystem is AdminManagement, and the classes in it are shown in the following figure:

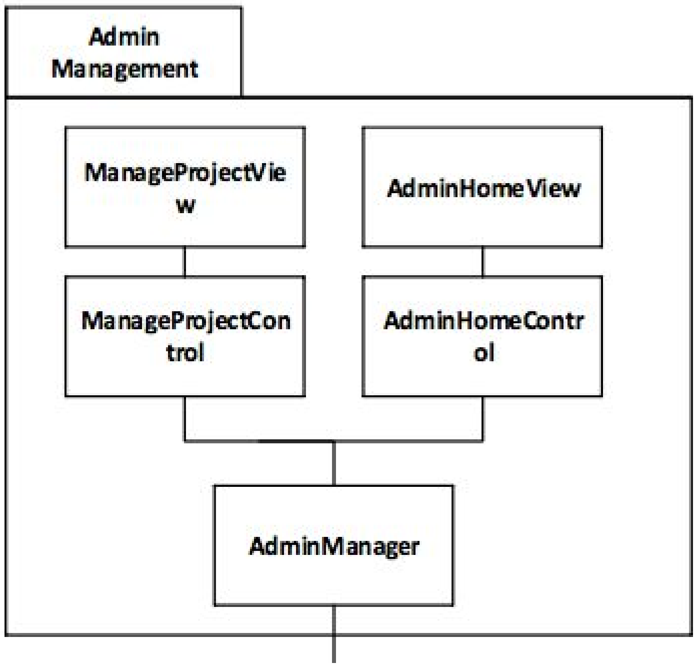


Figure 5: Phase 2 Subsystem Decomposition Diagram -- AdminManagement Subsystem

Like UserManager in the UserManagement subsystem, the AdminManager is a facade/singleton class, and the control objects have a unidirectional "uses" relationship with it. It also has unidirectional a "uses" relationship from itself to StorageManager (outside of the diagram).

Traceability

The Admin Features subsystem traces back to:

- F-04 Administrators can edit project settings
- F-06 Administrators can view the list of existing projects
- F-07 Administrators can create new projects
- UC-04 Manage Project
- UC-09 Create Project
- UC-10 Edit Project
- UC-13 View Existing Projects
- UC-17 Invalid Input Error
- UC-19 Repository Error
- EO Admin
- EO Project
- S-07 Create Project
- S-08 Edit Project

- S-09 View Existing Projects
- S-11 Invalid Input Error
- S-14 Repository Error

The traceability for each class:

Table 8: Traceability for AdminManagement subsystem

Class	Traceability
AdminHomeControl	F-06, UC-04, UC-13, UC-19, EO-Admin, S-09, S-14
AdminHomeView	F-06, UC-04, UC-13, UC-19, S-09, S-14
ManageProjectView	F-04, UC-04, UC-09, UC-10, EO-Project, S-08, S-11, S-14
ManageProjectControl	F-07, UC-04, UC-09, EO-Project, S-07, S-11, S-14
AdminManager	F-07, UC-04, UC-09, EO-Project, S-07, S-11, S-14

The next subsystem is Student Management, shown in the following diagram:

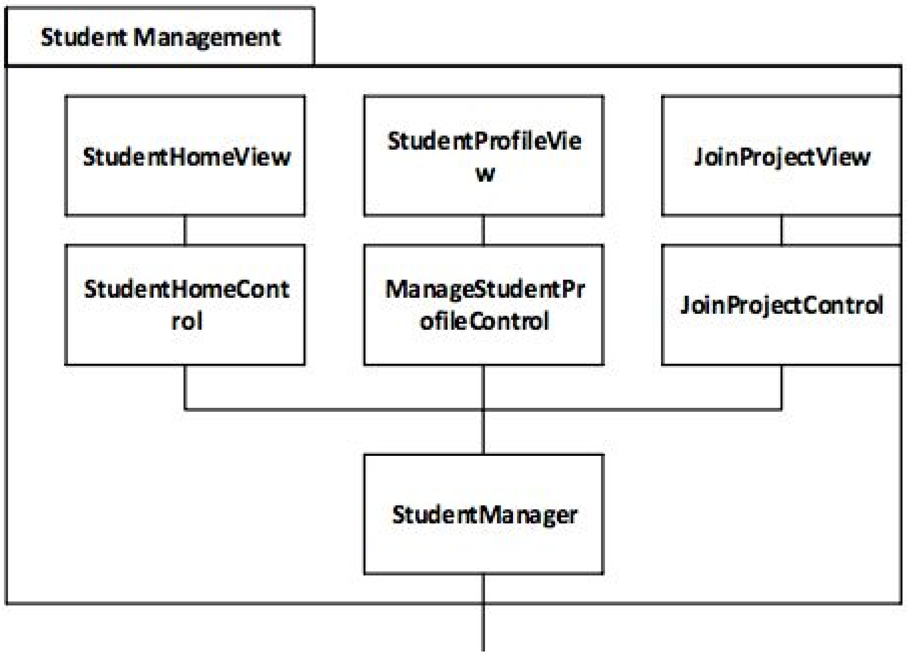


Figure 6: Phase 2 Subsystem Decomposition Diagram -- StudentManagement Subsystem

Like UserManager in the UserManagement subsystem, the StudentManager is a facade/singleton class, and the control objects have a unidirectional “uses” relationship with it. It also has unidirectional a “uses” relationship from itself to StorageManager (outside of the diagram).

Traceability

The Student Features subsystem traces back to:

- F-01 Students can add themselves to an existing project
- F-02 Students can edit their own profile
- F-03 Students can view a list of existing projects
- UC-02 Edit Profile
- UC-03 Join Project
- UC-13 View Existing Projects
- UC-16 Profile Error
- UC-17 Invalid Input Error
- UC-19 Repository Error
- EO-Qualification
- EO-Profile
- EO-Project
- EO-Student
- S-04 Edit Profile
- S-05 Join Project
- S-09 View Existing Projects
- S-11 Invalid Input Error
- S-13 Profile Error
- S-14 Repository Error

The traceability for each class:

Table 9: Traceability for StudentManagement subsystem

Class	Traceability
ManageStudentProfileControl	F-02, UC-02, UC-16, UC-17, UC-19, S-04, S-11, S-13
StudentProfileView	F-02, UC-02, UC-16, UC-17, UC-19, S-04, S-11, S-13
StudentHomeControl	F-03, UC-13, UC-19, S-09
StudentHomeView	F-03, UC-13, UC-19, S-09
JoinProjectControl	F-01, UC-03, UC-19, S-05
JoinProjectView	F-01, UC-03, UC-19, S-05
Profile	F-02, UC-02, UC-19, EO-Profile

StudentManager	F-02, UC-02, EO-Qualification, EO-Student,EO-Profile,S-14

The next subsystem is AlgorithmManager, shown in the following figure:

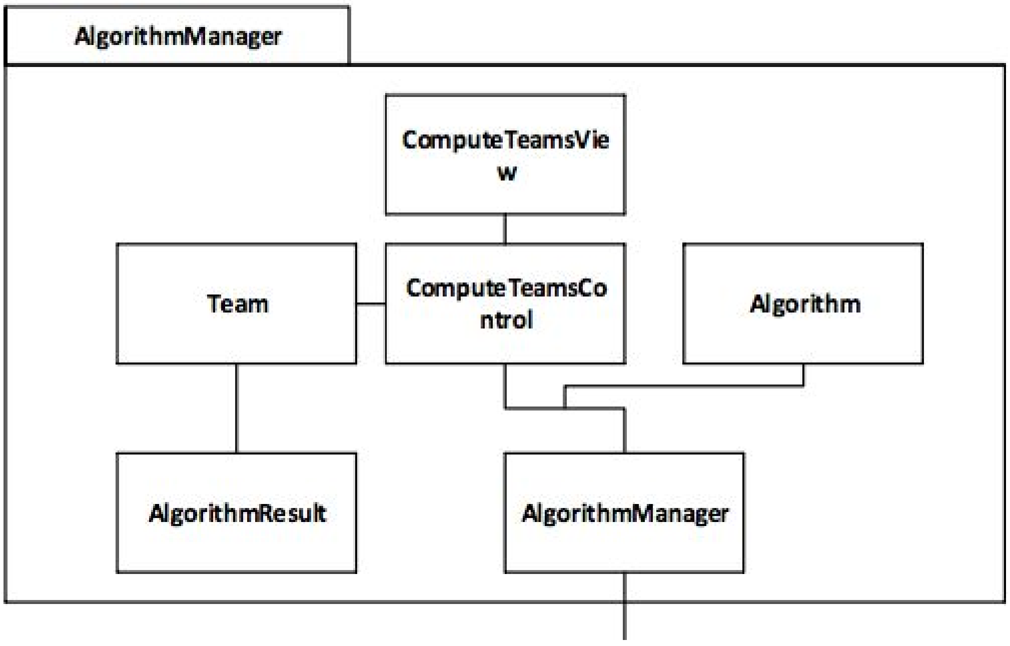


Figure 7: Phase 2 Subsystem Decomposition Diagram -- AlgorithmManager Subsystem

Like UserManager in the UserManagement subsystem, the AlgorithmManager is a facade/singleton class, and the control objects have a unidirectional “uses” relationship with it. It also has unidirectional a “uses” relationship from itself to StorageManager (outside of the diagram).

Traceability

The AlgorithmManager Features subsystem traces back to:

- NF-07 The number of all possible failures like system crash or inability to save changed data during
- NF-09 If the system fails during usage it must be able to load the last saved settings

- NF-30 The system must comply by the University Collection of Personal Information privacy policy

- S-06 ComputeBestTeams
- F-08 Administrator must be able to view Summary Result for the Match result
- F-09 Administrator must be able to view Detailed Result for the Match Result
- NF-13 The program must be able to run the project partner matching algorithm for 100 profiles in less than a minute
- UC-05 Administrator can run the Project Partner Matching algorithm and view the result
- UC-11 Administrator user views Summary result after running Project Partner Matching algorithm
- UC-12 Administrator user views Detailed result after running Project Partner Matching algorithm
- UC-12-1 Administrator user views matched result after runinng Project Partner Matching algorithm
- EO-Profile
- EO-Student
- EO-Project

The traceability for each class:

Table 10: Traceability for AlgorithmManager subsystem

Class	Traceability
ComputeTeamView	F-08, F-09, UC-05, UC-11, UC-12, UC-12-1,S-06, S-12,S-13,
ComputeTeamControl	EO-Profile, EO-Student, EO-Project, F-08, F-09, UC-05, UC-11, UC-12, UC-12-1,S-06, S-12,S-13,NF-07,NF-09, NF30

The next subsystem is Storage, shown in the following figure:

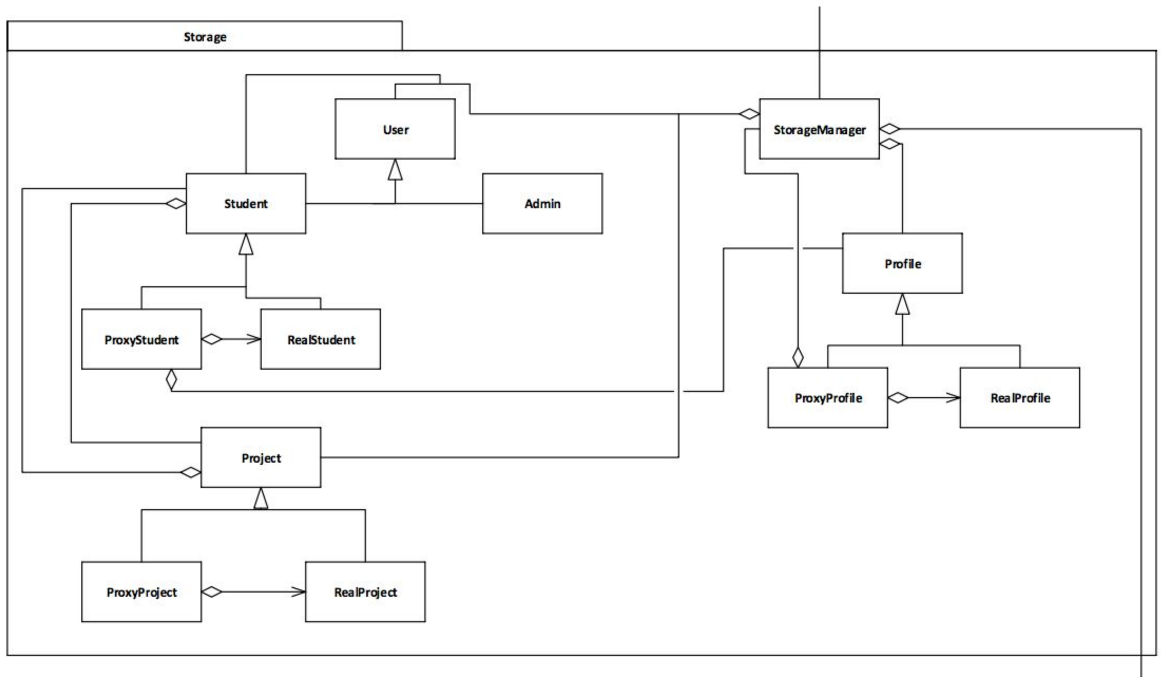


Figure 8: Phase 2 Subsystem Decomposition Diagram -- Storage Subsystem

The StorageManager contains a reference to the Database interface (implemented by SqliteDatabase) in the Database subsystem. The relationship coming in from the top of the diagram is a "uses" relationship from UserManager, StudentManager, AdminManager, and AlgorithmManager to StorageManager.

Traceability

The Storage Features subsystem traces back to:

- NF-07 The number of all possible failures like system crash or inability to save changed data during one-day usage must be less than 2
- NF-08 The system must be able to process invalid user input avoiding system crash or corrupting data
- NF-09 If the system fails during usage it must be able to load the last saved settings
- NF-10 The system must be able to provide a stable and reliable connection between client and server
- NF-10-01 The system must be able to notify user if during usage clients gets disconnected and attempt to connect
- NF-11 The system must be able to start in less than a minute on a computer with at least 2GB RAM or 4GB RAM if it is running on virtual machine
- NF-12 The system must be able to store efficiently at least 1,000 student profiles without noticeably slowing down
- EO-Project

- EO-Student
- EO-Student
- EO-Admin
- NF-14 The system must perform profile/project information update in less than 10 seconds
- NF-15 The system must be available all the time without any downtime
- NF-29 We will not be liable/responsible for any system failures or damages
- NF-30 The system must comply by the University Collection of Personal Information privacy policy
- NF-31 There won't be any specific components or third party libraries incorporated into the system in order to avoid any form of royalties fee. Any library used will provided by the operating system itself
- UC-17 Invalid Input Error
- UC-19 Repository Error
- S-11 Invalid Input Error
- S-14 Repository Error

The traceability for each class:

Table 11: Traceability for Storage subsystem

Class	Traceability
StorageManager	NF-07, NF-08, NF-09, NF-10, NF-10-01, NF-12, NF-13, NF-14, NF-15, S-12, S-13, S-15,EO-Student, EO-Admin,EO-Profile, EO-Project

The following diagram shows the Database subsystem:

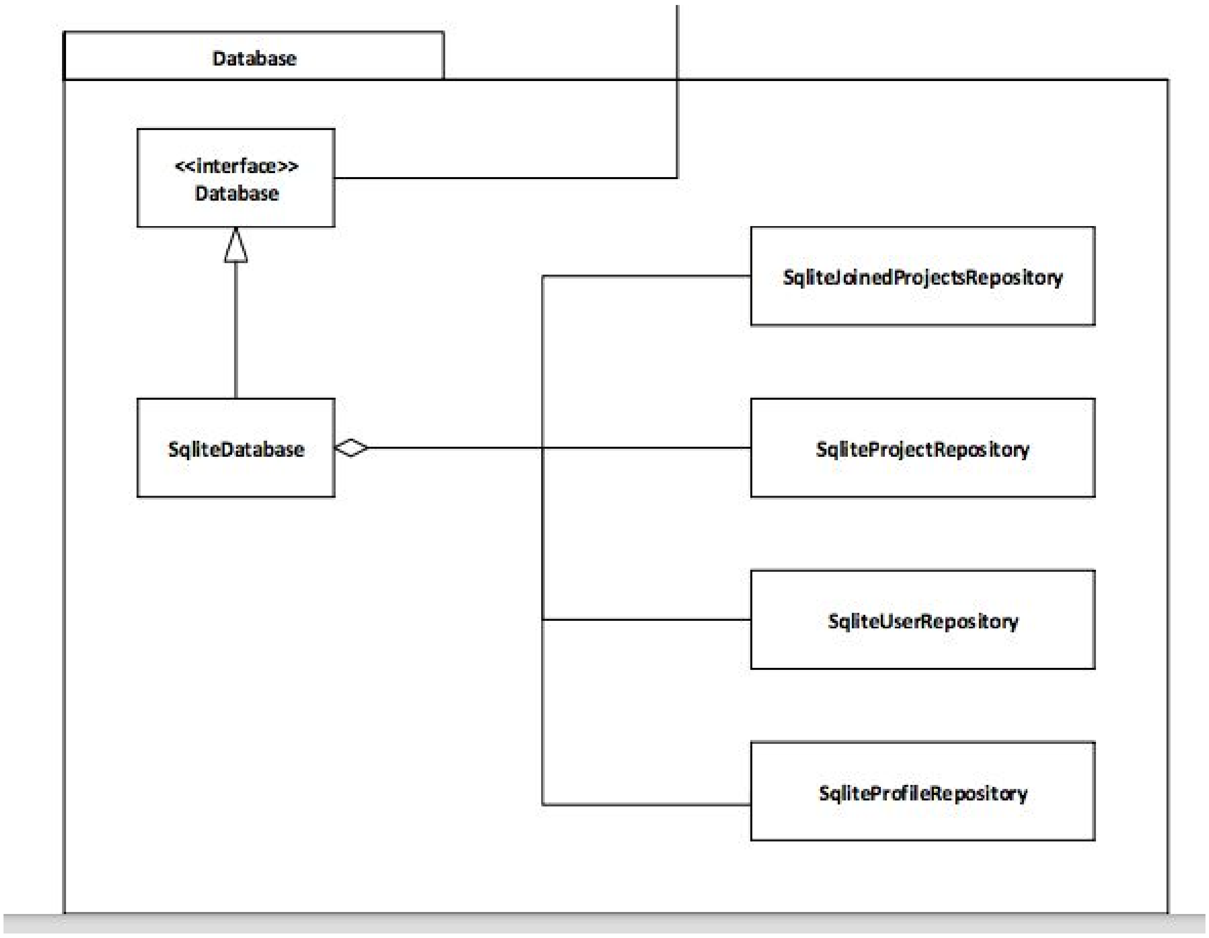


Figure 8: Phase 2 Subsystem Decomposition Diagram -- Database Subsystem

Traceability

The Database Features subsystem traces back to:

- NF-07 The number of all possible failures like system crash or inability to save changed data during one-day usage must be less than 2
- NF-08 The system must be able to process invalid user input avoiding system crash or corrupting data
- NF-09 If the system fails during usage it must be able to load the last saved settings
- NF-10 The system must be able to provide a stable and reliable connection between client and server
- NF-10-01 The system must be able to notify user if during usage clients gets disconnected and attempt to connect
- NF-11 The system must be able to start in less than a minute on a computer with at least 2GB RAM or 4GB RAM if it is running on virtual machine
- NF-12 The system must be able to store efficiently at least 1,000 student profiles without noticeably slowing down
- NF-14 The system must perform profile/project information update in less than 10 seconds

- NF-15 The system must be available all the time without any downtime
- NF-29 We will not be liable/responsible for any system failures or damages
- NF-30 The system must comply by the University Collection of Personal Information privacy policy
- NF-31 There won't be any specific components or third party libraries incorporated into the system in order to avoid any form of royalties fee. Any library used will provided by the operating system itself
- UC-17 Invalid Input Error
- UC-19 Repository Error
- S-11 Invalid Input Error
- S-14 Repository Error

The traceability for each class:

Table 12: Traceability for Database subsystem

Class	Traceability
SqliteDatabasem, SQliteJoinedProjectRepository, SQliteProjectRepository, SQliteUserRepository, SQliteProfileRepository,	NF-07, NF-08, NF-09, NF-10, NF-10-01, NF-12, NF-13, NF-14, NF-15

Design evolution

The phase 2 cuPID system design offers many improvements over the prototype design. This section examines the changes between the prototype and the full system design and offers explanations and justifications for them.

Improvements

Moving SQL code into one place to reduce coupling

A consequence of the phase 1 design is that there are many relationships between subsystems, especially with the Storage subsystem. But the real weakness of the phase 1 design is that the entity objects (Project, Profile, Student, etc.) interacted directly with the SQLite database, and those entity objects were used just about everywhere. So this meant that all over the code, things were using objects that were tightly coupled to our database implementation.

Furthermore, it didn’t make much logical sense for the entity objects to communicate directly with the database. The Project object, for example, should be able to tell you which students are in it, but it should rely on some other class to get that information from the database. What’s worse, we relied directly on the concrete entity object classes, rather than abstractions of their interfaces. If we had wanted to replace the actual database with a unit testing system that returned test data, we would have had to change every entity object reference.

Additionally, we were inconsistent about whether control objects or entity objects were responsible for communicating with the database. This resulted in SQL code being just about everywhere except the view classes, making the control classes unnecessarily complex and introducing even more unnecessary coupling.

Using facade objects to reduce coupling

We reduced coupling between subsystems by introducing facade objects to sit at the subsystem boundaries and handle communication between them. This means that instead of all the control classes within the user, student, admin, and algorithm subsystems interacting with the Storage subsystem, the control classes each interact with a facade object within their own subsystem, which in turn interacts only with the Storage Manager class.

Clearly defining the role of entity objects

Another weakness of the prototype system is the unclear role of entity objects: are they structs for passing data around, are they objects for interacting with the database, should they consistently model the actual state of the system, or should each control object use them however it likes.

In the phase 2 design, entity objects live in the Storage subsystem. They are owned by the StorageManager, which is a facade-like singleton that maintains a consistent cache of all the objects that have been retrieved from the database. This allows us to eliminate unnecessary calls to the database. It also means that entity objects have a clear purpose: any changes to the data will be reflected in changes to objects in the Storage subsystem before being passed onto the database for saving. When the other subsystems ask for information about the data, the Storage subsystem can give them a copy of the current cached object.

Reducing database queries with the proxy pattern

Sometimes it makes sense to ask a Student object for its attached profile, for example, but we don’t always need the profile information. We use a proxy pattern to allow the other subsystems to use Student objects, to have the ability to ask them for their profile, but the database queries won’t be made unless the code tries to access the profile properties.

Database independence

Whereas the phase 1 design was tightly coupled with the SQLite implementation, in our phase 2 design the type of database could be changed (perhaps from SQLite to flat file storage) without even changing the Storage class, let alone every other subsystem. We achieved this decoupling by defining a Database interface class that StorageManager relies on, and then offering the SQLite implementation as one solution.

Simplifying the coupling by consolidating entity objects into Storage subsystem

When looking at the phase 1 subsystem decomposition, it may be unclear why we chose to keep some entity objects in subsystems other than Storage (e.g. keeping Project in the Admin subsystem). We did this in phase 1 because it increased cohesion and reduced coupling. The admin feature classes tended to use the Project class more often than any classes elsewhere did. An admin can create and edit project, view a list of projects, and so on. Whereas the only thing project was used for elsewhere was the user getting the list of projects he’s joined.

However that design had problems. First, it made it difficult to add subsystems. When it came time to create the Algorithm subsystem, for example, it needed to access Student, Project, and Profile, which meant reaching into the User Management, Student Features, and Admin Features subsystems. In phase 2 we moved the entity classes into the Storage subsystem, which is why now the Algorithm Management only depends on the Storage subsystem.

It also simplified the coupling between User Management, Student Management, and Admin Management. They might *use* something in one of their peer subsystems (e.g. User Management using the profile creation service of Student Management), but there won’t be strong dependencies like the one between Student and Project (a student has a list of projects, but Student was in User Management and Project was in Admin Management).

At one point in phase 1 we even considered pretending there wasn’t a relationship between Student and Project, just so we could say there wasn’t coupling between the Student Management and Admin Management subsystems. Our phase 2 design is much better, as it allows us to represent real relationships between entity objects without bad design.

Weaknesses

Tight coupling between the user interfaces and the control objects that implement the application logic means that we would have a hard time creating an alternative view (e.g. from a web interface) for the same cuPID features.

There is a lot of repetition between the StorageManager and the Database interface, and between UserManager, etc, and StorageManager. When we chose to have facades at every layer facilitating communication, that means that we will have a lot of code that simply handles function calls by passing them to another object. This is inefficient during run-time, and it takes extra time when coding as well. It also means that if we change one interface we will have to change many interfaces.

Note that although the StorageManager maintains a cache of entity objects, it is not responsible for enforcing all the application logic rules. That would turn this into a three-tier design, rather than the repository architecture we have now. We will have to be careful not to make StorageManager too “smart” during implementation. We should also try to not make the UserManager and other facade classes too “smart”, since true facades do not implement a great deal of application logic. We want the controller classes to manage that.

Class changes

One part of the evolution from the prototype design to the full system design involved renaming some classes, adding some classes, and removing some classes. This section details what changed and why.

Renamed classes

Some of the class names in the prototype were confusing. Sometimes there was inconsistency between the name of the control and the name of the view, and it wasn’t clear that they were related. And some classes ended up being merged and had to be renamed to reflect their broader purpose. The following table shows which classes were renamed and why.

Table 13: Classes renamed during design evolution

Name in prototype	New name	Reason
StudentMainControl	StudentHomeControl	Sounds better and makes it consistent with name of StudentHomeView
AdminMainControl	AdminHomeControl	Sounds better and makes it consistent with name of AdminHomeView
SignupOptionView	AccountTypeOption	Makes the purpose of the class more clear
MainWindowContol	LoginControl	Makes the purpose of the class more clear
MainWindow	LoginView	Makes the purpose of the class more clear
EditStuProfileControl	ManageStudentProfileControl	The abbreviation was unnecessary, and the same control is used for creating and editing profiles.

EditProjectControl	ManageProjectControl	We combine the functionality of editing, joining, and creating projects to reduce code duplication.
EditProjectView	ManageProjectView	We combine the functionality of editing, joining, and creating projects to reduce code duplication.

Removed classes

Some classes in the prototype were redundant, badly designed, or served no clear purpose. The following table shows which classes were removed and why.

Table 14: Classes removed during design evolution

Removed class	Reason
SignupMainControl	This control object did not represent an actual use case; it only allowed the user to select whether they wanted to create an admin account or a student account. We can simplify the design by moving the code into LoginControl. LoginControl displays the AccountTypeOption and then launches the control for the right account creation use case.
Questions	This was a badly designed class that only served to read information from a static file and provide it to the control object for editing profiles. We can simplify the design by moving that functionality into the control object.
CreateProjectControl	Functionality moved to ManageProjectControl, which handles creating, editing, and joining projects.
CreateProjectView	Functionality moved to ManageProjectView, which handles creating, editing, and joining projects.

The major improvement here was combining create and edit project functionality into a single view/control class pair. We realized that the view for each of those use cases was identical except for the window title, button text, and editability of some fields (in the case of join project). The control class can take care of telling the view which mode to operate in, and that lets us remove 2 classes, which simplifies our design.

Added classes

Some classes were added during the design evolution. Many were added because the prototype only covered some of the classes the full design would need. Others were added to improve the design and reduce coupling. The following table shows which classes were added and why.

Table 15: Classes added during design evolution

Added class	Purpose
Team	Contains information about a grouping of students into a team and why they were matched together.
MatchResults	A complete assignment of every student in a project into teams. Also manages functionality relating to displaying detailed or summary results.
AlgorithmManager	A facade class to simplify computing the best teams for a project.
SqliteProjectRepository SqliteUserRepository SqliteProfileRepository SqliteJoinedProjectsRepository	These classes in the Database subsystem are responsible for implementing access to the tables in our SQLite database.
SqliteDatabase	A facade class that simplifies access to all the ways we have of storing different objects.
Algorithm	An interface or abstract class for taking a list of students in a project and returning the match results. Used in a strategy pattern to allow the user to select different ways of computing the best teams.
ComputeBestTeamsControl	Control class for the compute best teams use case.
ComputeBestTeamsView	View for the compute best teams use case.

Additionally, the Database class was repurposed as a general interface that offers access to persistent storage regardless of how it is implemented. There were also some added classes involving the proxy pattern, but those are shown in the subsystem diagrams and explained in the design patterns section.

Reducing dependencies

This section details many of the places where we changed the phase 2 design to reduce coupling.

Student Features subsystem

StudentMainControl (renamed to StudentHomeControl)

The purpose of StudentHomeControl is to allow the student to see a list of projects they are in and a list of projects they are not in, and select projects to join. There is also an option to edit the student's profile. In the prototype it had 2 dependencies on other subsystems. The following table shows what changed:

Table 16: Dependency evolution for StudentMainControl

Dependency	Purpose	Design Evolution
Database (in Storage)	To get list of joined and unjoined projects	Gets lists from StudentManager class instead, which is within the same subsystem.

EditStuProfileControl (renamed to ManageStudentProfileControl)

The purpose of ManageStudentProfileControl is to support creating and editing student profiles. In the prototype it had no direct dependencies on other subsystems, though it used the Profile class to communicate with the database, which turned out to be bad design. The following table shows how the dependencies of this class will change in the full system design:

Table 17: Dependency evolution for EditStuProfileControl

Dependency	Purpose	Design Evolution
Profile	To create profiles To edit profiles To get profile data	Uses StudentManager for create profile, edit profile, and get profile data. This turns a dependency on another subsystem into cohesion within the subsystem.

StudentProjectControl (merged with ManageProjectControl)

The purpose of StudentProjectControl was to allow the student to view the details of a project and elect to join it. The following table describes its dependencies and how they will change:

Table 18: Dependency evolution for StudentProjectControl

Dependency	Purpose	Design Evolution
Database (in Storage)	To retrieve the project settings	Will call on StudentManager for this instead, which reduces coupling with the database. We would like as few classes as possible directly communicating with the database.
Project	To register the student with the project	Will call on StudentManager for this instead, which reduces coupling with the database. We would like as few classes as possible directly

Profile

The purpose of the Profile class is to hold a working copy of all the data in a student profile and allow its client classes to manipulate that data. In the prototype the Profile class also offered an interface for creating, editing, and reading profile information from the database. The following table describes its dependencies and how they will change:

Table 19: Dependency evolution for Profile

Dependency	Purpose	Design Evolution
Database	To create profiles To edit profiles To read profiles	Profiles will no longer go directly to the database. ProxyProfile will go to StorageManager if it needs more information. The ones using the Profile can also call methods on StorageManager for profile creation, editing, and retrieval.

Admin Features subsystem

AdminMainControl

In phase 1, this class issued SQL queries against the database to get a list of all projects. In phase 2 it will use the AdminManager class for that.

Project

The Project class used to directly access the database. Now the ProxyProject uses the StorageManager if it needs more information (e.g. to get the list of joined students). Otherwise it is the responsibility of the calling class to issue project-related commands (create, edit, retrieve, join) to StorageManager.

Admin

The Admin class used to contain SQL code to add an admin to the database. Now classes that used Admin to do that access the same functionality through StorageManager.

Student

The Student class used to contain SQL code to add a student to the database. Now classes that used Student to do that access the same functionality through StorageManager.

MainWindowControl (renamed to LoginControl)

In phase 1 this class connects directly to the database to initialize it, close it, and lookup users. Initializing can happen lazily in the Database class once the first database call is made, so we can skip that. Closing may not be necessary, since it’s closed anyways once the program quits. And user lookup could be accessed through the UserManager facade. The User Management subsystem is no longer responsible for handling database implementation details.

CreateStudentAccountControl

In phase 1 this class accessed Database to query for the max user id. But that’s unnecessary since the user id field autoincrements, so you don’t need to know it when creating a new user. We can easily cut this dependency.

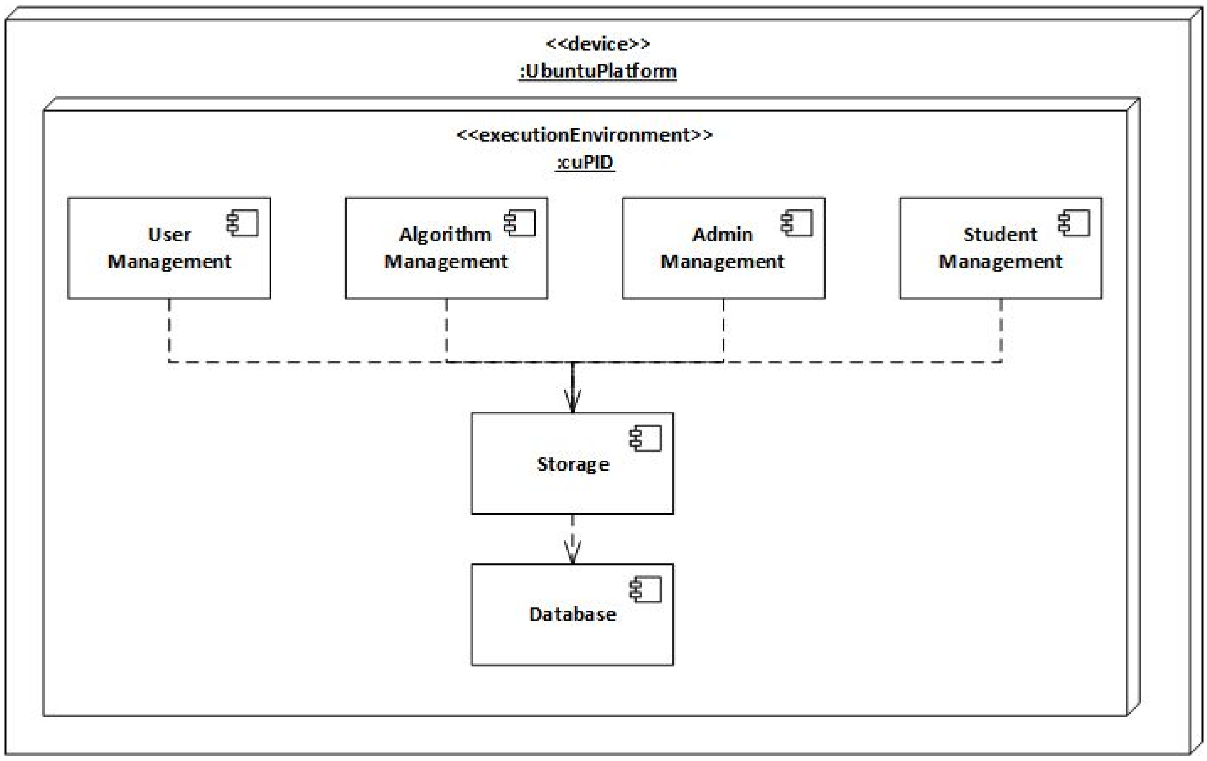
Design strategies

This section covers our phase 2 architecture, how the subsystems are mapped to hardware and runtime components, what strategy we use for persistent data storage, and which design patterns we use.

Hardware/software mapping

We chose a repository style architecture for the cuPID system. The following figure shows the cuPID subsystems and how they are mapped to hardware and runtime components:

Figure 10: UML deployment diagram for cuPID



The cuPID system is only intended to run on a single computer (running Ubuntu, or a Ubuntu virtual machine), and there should only be one instance of it running at a time. This makes the deployment diagram very simple. All the subsystems are contained within the single cuPID binary, and they all exist in the same process when you run it.

The UML deployment diagram also shows why a repository architecture is a good match for the system. We have four subsystems at the top responsible for user interface and application logic, and they each rely on the Storage subsystem, which in turn depends on the Database subsystem. Conceptually we can combine the Storage and Database subsystems into a single Repository component, which everything else depends on.

The Repository architecture is exactly that: your program is organized around a single complex data structure or repository which all other subsystems use. We have high cohesion in each of the subsystems, and there is very simple coupling between them. None of the four subsystems on top have very strong dependencies on each other (just uses relationships).

Our system doesn't really need to be viewed in terms of layers and partitions. If you had to look at it that way, you would say there are two layers: one for application, containing the four management subsystems on top, and one for repository, containing Storage and Database. It would be a closed architecture, though that term isn't very meaningful when there are only two layers.

The downside is that there is always coupling between the repository parts and the non-repository parts. In the repository architecture, the repository can become a bottleneck, and changes to it may require changes to many of the other subsystems. For this reason we decoupled the Storage subsystem from the Database. We believe that the interface offered by Storage is less likely to change than the interface offered by the Database; for example, we could change the database backend from SQLite to something else without affecting Storage. We could also change the details of some SQL queries in the Database layer and the Storage interface would stay the same.

We also considered the Three Tier and MVC architectures, but we decided against using them.

The Three Tier architecture would be just like what we have now, except divided into three layers instead of two. The Storage and Database subsystems would stay in the Storage layer, but the other subsystems would each be split into separate interface and logic subsystems, which would be separated into the interface and logic layers. However, the use case for three tier seems to decouple the interface, application logic, and storage so much that you could reuse the storage layer in different applications, and you could create different interfaces for the same application logic.

If we wanted to support different interfaces for cuPID, such as an API to access the database for a completely different application, or a web interface to access the cuPID application features from something other than the Qt GUI, a Three Tier design would be great. However, in the current implementation we have no plans to support those things, and so the Three Tier design would be overkill.

A Three Tier architecture would also mean the many dependencies between the user interfaces and the control objects that implement the logic for them would count as coupling instead of cohesion in our subsystem decomposition. We would be separating tightly coupled classes that serve a common purpose into different subsystems.

We also considered the MVC architecture, which is a special case of the Repository architecture where the model classes maintain the domain knowledge, the views display information to the user, and the control classes handle the control flow for the different cases of interacting with the user. The model classes would keep the view classes updated using the observer pattern.

The MVC architecture, however, would be better suited to something like a multiplayer game where many user interfaces have to be kept synchronized with changes to the model. In our case we only have a very simple user interface, and there should only be one instance of the system running at a time. There are only a few times when we have to update the user interface based on changes to the model, and it would overkill to use the observer pattern just for those few cases, let alone build our entire architecture around them.

Persistent data management

We store our persistent data using SQLite in 4 tables. The following table shows what they are and what they trace back to:

Table 20: Traceability for database tables

#	Table name	Traceability
DBT-1	user	F-10, F-11, F-12, UC-01, UC-06, UC-07, UC-14, UC-15, EO-User, EO-Admin, EO-Student, S-01, S-02, S-03
DBT-2	project	F-01, F-03, F-04, F-05, F-06, F-07, UC-03, UC-04, UC-05, UC-09, UC-10, UC-13, EO-Project, S-05, S-06, S-07, S-08, S-09
DBT-3	profile	F-02, F-05, F-11, NF-02, UC-01, UC-02, UC-05, UC-07, UC-16, EO-Profile, S-01, S-03, S-04, S-06, S-13
DBT-3	project_student_registered	F-01, UC-03, EO-Student, EO-Project, S-05

This section has three subsections: the first describes why we chose SQLite, and the second describes how the data is organized, what objects are in persistent storage, and how we avoid data duplication. The third briefly describes other types of information that we chose not to store in the database.

Why we chose SQLite

We had two options for our choice of persistent storage: flat files or a relational database.

For flat file storage we could have chosen to store objects as JSON, XML, or some other format. One advantage of choosing flat files is that we could organize the data in a way that easily maps to objects in code, as long as we can define some way to serialize and deserialize them. By choosing relational databases we are left with some less intuitive ways of organizing data, and we may need slightly more detailed code to convert objects to/from SQL queries, particularly where table joins are involved.

On the other hand, flat files have many disadvantages. We would have to write our own code for reading and writing from the files in whatever file format we have. We would have to write custom code for searching, updating, and adding data, which is more likely to introduce errors into the system than relying on standard SQL library code.

There are also the issues of concurrency, atomic read/writes, crash recovery, and data integrity. One of the major reasons we chose SQLite was because it solves these problems for us.

As for why we chose SQLite in particular, as opposed to MySQL or Postgres, we knew that SQLite was on the virtual machine we needed to deploy to, and SQLite doesn’t require any configuration beyond making sure the program knows where to find the database file. Both MySQL and Postgres would require configuring a server, which opposes our design goals of simple deployment.

Flat files are good for large, binary blobs of data (like images), temporary data, and sparse data, but none of those apply here. We don’t use any images, we keep our temporary data in memory, and our data isn’t sparse.

The cuPID system is meant to be used by one person at a time on a single system, so concurrency is not much of a concern. However it is good to have the ability to support it in the future. Furthermore, our use of Qt GUIs may hide some multithreading that could theoretically result in concurrent database access, and using SQLite helps protect us from unforeseen bugs in that regard.

The cuPID system is not meant to amass gigabytes of data, but the database could still grow enough that SQLite starts to be more efficient than flat file storage. We expect maybe a thousand students could be entered into the system per semester, and it could be used for several years. SQLite is the safer choice, given that we don’t know for sure how much data there will be.

The most complicated queries we will have to make are listing all the students in a project or all the projects a student is registered in. Relational databases can handle such queries easily, which is another reason we chose SQLite. It wouldn’t have been much of a problem to write flat file code for that, but it still would have been more custom code and more places where we could introduce errors into the program.

Object-oriented databases were not considered because none of the development team had experience with them and we were unsure if the deployment environment would have one installed.

Data organization

Persistent data for the cuPID project is stored in 4 tables: user, project, profile, and project_student_registered. The last one is a join table that just shows which students are in which projects. This section describes each table, why we designed it that way, and how we avoid duplicate data.

Organization of “user” table

The “user” table is responsible for persistent storage of Admin and Student objects. The User class is a generalization of those and would exist purely for abstraction; such an object would not actually be persisted. The relationship between students and profiles is stored in the profile table.

The following table shows how the “user” table is designed:

Table 21: SQLite layout for “user” table

Attribute	**Data type**	**Constraints**
id	integer	primary key, autoincrement
username	char (max size 20)	unique, not null
display_name	char (max size 30)	not null
student_id	integer	unique

id: A unique identifier for each user account. Not visible to the user. We had the option of using the unique username as the primary key, but decided to add an integer id field instead. This actually ends up saving space in the long run, since the primary key will have to appear over and over again in the student-project association table.

username: every admin or student has a username that they log in with. Names shouldn’t be unreasonably long. 20 characters is enough. Carleton’s MC1 system only allows 20 characters. If their name is longer than 20 characters, we can truncate it at 20, just as Carleton’s MC1 system does. Although this can be confusing, it also saves people from typing really long user names.

Usernames must be unique within the system.

display_name: The name (of the student or admin) that the system displays. This is not always the same as the username, which is what they login with. The username has length limits and is always lowercase. So a username may be “donaldtrump” and a display name may be “Donald Trump”.

The other parts of the system may enforce a rule by which usernames can be determined from display names, which may make this an example of redundant data, but it makes queries much simpler. It also leaves the option open if we later want to allow students, perhaps with very long names, to choose a significantly shorter name (less than 20 characters), rather than taking their full name and truncating.

We do not allow users to leave the display name null since to maintain a professional presentation the full name of the logged in user should be nicely displayed to them, and we can’t just use their username for that.

Unlike with usernames, display names do not have to be unique within the system. It is entirely possible for two distinct students to have the same first and last names. If that happens, they may have usernames such as johnsmith and johnsmith2, but their display names would both be John Smith.

Note that if the administrator is viewing a list of the students in a project, and two of them have the same display name, there will be no confusion. In all cases where the system lists multiple students, it will also show the student IDs alongside the names.

student_id: Carleton student IDs are 9 digits long, so a regular integer will suffice.

If this is an admin account, the student_id field will be NULL. We could also have created a separate table for administrators, who would not have a student_id field but would otherwise be the same, but then checking if the user entered a valid username in the login screen would require an extra join. Using a single table makes for simpler queries, despite the complication of allowing null values.

We chose something like a horizontal mapping of the user object hierarchy to database tables. We prefer to do it this way because the admin does not have any extra fields that the student does not. This minimizes the usual cost of a horizontal mapping, which is that you need different tables for each object in the hierarchy, and the superclass attributes in the tables are repeated. However, it places a limitation on our design: we will need to do extra modifications if we do choose to add special attributes to the admin class.

Organization of “project” table

The “project” table is responsible only for persistent storage of Project objects. The relationship between students and projects is tracked in a separate table.

The following table shows how the “project” table is designed:

Table 22: SQLite layout for “project” table

Attribute	**Data type**	**Constraints**
id	integer	primary key, autoincrement

name	char (max size 70)	unique, not null
min_team_size	integer (1 byte)	default 1
max_team_size	integer (1 byte)	not null
description	varchar (max size 1024)	not null

id: A unique identifier for each project. Not visible to the user. We had the option of using the unique project name as the primary key, but decided to add an integer id field instead. This actually ends up saving space in the long run, since the project's primary key will have to appear over and over again in the student-project association table.

name: a 50-character string. Most values will be fairly short, possibly just the course code and a very short title after that. 70 characters allows us to have somewhat long names such as "COMP 3004A Fall 2015 Term Project -- cuPID" (42 characters), but supporting names much longer than that is unnecessary and would complicate the GUI.

Project names must be unique within the system. It is likely that over years of use the system will have, for example, several COMP 3004 or COMP 4109 projects, and it would be confusing if they were all names "COMP 3004" or "COMP 4109". The user interface should provide some sort of hint about what a good project name looks like.

min_team_size: A 1-byte signed integer. Since we aren't expecting any 100-person teams, this is enough. The default value is 1, to allow the other parts of the system to possibly simplify the user interface by not asking for a minimum team size. This keeps our options open. If the min team size must be the same as the max team size, the rest of the system is responsible for enforcing that logic.

max_team_size: A 1-byte signed integer. Since we aren't expecting any 100-person teams, this is enough. The administrator creating the project must specify a maximum team size, so this field cannot be NULL.

description: A varchar with a max size of 1024. That lets the project description contain up to a few paragraphs of text while saving space if the description is shorter. A project must have a description, so this field cannot be NULL.

Organization of "profile" table

The "profile" table is responsible for persistent storage of Profile objects. It is also responsible for knowing which profile belongs to which student.

While we could have combined the profile table with a student table, that would have made the storage requirements for student and admin complicated enough to necessitate separate tables. A separate profile table lets us keep a single user table, which simplifies the login query because it only has to check the entered username in one place.

Additionally, the profile table is large (56 attributes) and a lot of the time when we are querying student objects we don't care about their profile. So a separate table for profiles allows us to handle most student queries with a simple SELECT * without passing unnecessary data. Similarly it makes it easy to select only the profile when we need it. On the other hand, SELECT * may be a bad practice in general, since it means queries may break if we add columns.

Another reason we have a separate profile table is because the rest of our system makes a distinction between a Student object and a Profile object, so it makes some sense to have the database make the same distinction.

The following table shows how the "profile" table is designed:

Table 23: SQLite layout for "profile" table

Attribute	**Data type**	**Constraints**
user_id	integer	primary key, foreign key references user(id)
q1	integer	not null
q1min	integer	not null
q1max	integer	not null
...	...	...
q28	integer	not null
q28min	integer	not null
q28max	integer	not null

user_id: The id of the user this profile is for. *Note that this refers to the internal "id" attribute used for that row in the database, not the user-visible "student_id" attribute.* It also acts as a primary key, which places a limitation on our design: one profile per student. In the current implementation that's fine; we only want one profile per student. But if we want to later support one student having many profiles, we will have to change this table to have a separate id field as the primary key.

There is nothing in the database stopping a profile from being created for an administrator, which is unfortunate. It will be the responsibility of the code to prevent that from happening. It should be easy to enforce.

q1: The student's own answer to the first qualification question.

q1min: The lowest value for the first qualification question the student would prefer to see in his or her teammates.

q1max: The highest value for the first qualification question the student would prefer to see in his or her teammates.

We decided to include a separate min and max value for every question because our rules work based on the idea of a *preferred range*, as opposed to the algorithms of some other teams that operate based on closeness to a single *preferred value*.

Note that while everything in the table says it is an integer, question 4 asks for cumulative GPA which is actually a double. We will store q4, q4min, and q4max as doubles, even though that may complicate queries slightly.

We do not enforce the range of accepted values at the database level; that is the responsibility of the code.

Organization of “project_student_registered” table

This table is responsible for tracking which students are registered in which projects.

One of the challenges of mapping objects to relational databases is that many-to-many relationships, such as the one we have between students and projects, are somewhat unintuitive. This table uses a standard technique for many-to-many relationships: every time you want a connection between a student and a project you add a row containing the project id and the user id. This minimizes data duplication since the only part of the objects that appears multiple times is their ids, and the ids are just simple integers.

Furthermore, even if the project name or student name changes, we won’t have to track down those changes in multiple places; they are only stored in the project and user tables. The ids of projects and users are internal to the system and never change.

The following table shows how the “project_student_registered” table is designed:

Table 24: SQLite layout for “project_student_registered” table

Attribute	**Data type**	**Constraints**
project_id	integer	primary key, foreign key references project(id)
user_id	integer	primary key, foreign key references user(id)

project_id: The id of the project the student it in.

user_id: The id of the user that is in the project. Note that this is the internal “id” attribute of the user table, not the user-visible “student_id” attribute.

Unfortunately this design will allow admins to be registered into projects, which we don’t want. We could have joined on the student_id attribute instead, but generally we prefer to only do joins on primary keys. Perhaps that intuition is misguided in this case, but if so our team did not have the relational database experience to tell. In any case, it is easy to enforce in code the rule that only students can join projects.

Other information

The prototype stored two additional types of information:

- Images
- Question data

If our final version of cuPID uses images, they will be stored as flat files using Qt’s resources system. The question data will be in a simple text file, which will again be stored using Qt’s resources system.

The Qt resource system wraps up read-only data into the build somehow, which saves us the trouble of finding a way to deploy it. Unfortunately we cannot store the sqlite database file in the Qt resource system, since it has to be modified during the execution of the program. Instead we have the database file stored in the build directory, either through instructions in the readme or by a special build rule in Qt Creator.

When a student fills out their profile, they are presented with a series of 28 questions. Each question has a range of values it will accept, and the range changes per question. Our prototype stored the text prompts and range of accepted values in a file called “questions.txt”.

The advantage of using a flat file here is that we can easily edit the prompt for a question without opening up a database editor or recompiling the code. Furthermore the range of accepted values may be used in more than one place in the code, so it minimizes duplication to have them stored in only one place.

Finally, we chose to store the questions in a flat file rather than SQLite because the number of questions is fixed. Unlike the record of all users, projects, and profiles, the questions file should only be changed if the cuPID design or requirements change, and if it does the questions file can be edited by hand. If we expected the questions file to need to be edited by the program or to change often during the operation of the system we would have put the information in the database.

Design patterns

Our cuPID system design uses 3 established design patterns: proxy, facade, singleton. This section will discuss and justify the use of each of these patterns in detail. We also considered using the observer pattern but decided against it. The last part of this section will explain why.

Proxy

The proxy pattern is meant to be used when you have an object that is be expensive to create or initialize, and you want to defer that cost as much as possible. It's even better if sometimes you can avoid paying the full cost to initialize the object, for example when its most expensive functionality is never used. That is why we use the proxy pattern in our cuPID system design.

We use the proxy pattern in 3 places within the Storage subsystem. They fit the "virtual proxy" use case described in Gamma *et al* to create expensive objects on demand. The proxy object handles as much as it can without making expensive database queries, and will make those queries only if it absolutely has to.

The client code won't know if it is using the proxy object or the real object.

- With Project
  - A project can have many students. Project::getStudents should return all the students registered in the project. However, most of the time when we use a Project object we do not care about all the students in it. We use the proxy pattern where ProxyProject::getStudents is a stub method that queries StorageManager for the data, then creates a RealProject object to hold it. ProjectProxy can handle requests for the project id, name, description, and min and max team size without going to StorageManager.
- With Student
  - A student can be in many projects. Student::getProjects should return those projects. ProxyStudent::getProjects is a stub method that will query StorageManager for the data, then create a RealStudent object to hold it. Most requests on Student objects only ask for the user id, student id, username, or display name, and ProxyStudent can handle those without going to StorageManager.
- With Profile
  - Profiles are large (56 attributes), and sometimes when we are working with a Profile object all we really care about is its id. For example, the Student object may hold a Profile object but never actually use any of the data in it. In that case, all we need to do is have the Profile's id *in case* the data is needed. We use the proxy pattern here so that every ProxyProfile function besides getId is a stub method that defers to the RealProfile if it exists, and otherwise queries StorageManager for the profile data and creates a RealProfile object.

Note that once a Real[Something] object has been created, *all* the related methods in the Proxy[Something] object will defer to the real object, as per the established design pattern described in Gamma *et al*.

We considered using the proxy pattern in other places, namely for the Admin and User classes, but they have no expensive operations that could be deferred.

Without the proxy pattern, we would have had to choose between making our implementation of Student, Profile, and Project make unnecessary database calls, or omitting the relationships between those entity objects entirely and relying on the control objects to maintain those connections. Proxy leads to a more elegant design where real entity relationships are present without sacrificing efficiency.

Facade

The purpose of the facade pattern is to provide one public interface that simplifies access to several interfaces within a subsystem. It makes the subsystem easier to use while still allowing client code to access the classes behind the facade directly, if they need more fine-grained control. We use it for:

- The SqliteDatabase class
- A few other classes that serve the same purpose as facades

One suggested use of the facade pattern is when you have a large number of dependencies between client code and the classes in a subsystem that implement an abstraction. Such was the case in one of our earlier designs, where we had one class with SQL code that knew how to create a project, one class with SQL code that knew how to create a user, and so on, and there was no single, unified point of access. Even if we abstracted out the type of repository (SQL or flat file) in each class, there would have still been a lot of coupling.

We decided to create a generic interface in the Database subsystem, called simply “Database”, and we defined a facade class “SqliteDatabase” that implements it. The SqliteDatabase class passes on requests to each of the different implementation classes responsible for talking to the different parts of the database: SqliteProjectRepositoroy, SqliteUserRepository, SqliteProfileRepository, SqliteJoinedProjectsRepository. There is an aggregation relationship between each of those and the facade.

This has a few benefits for our design. First, we minimize coupling while avoiding a “blob” class that stores the SQL code for every possible database query. Second, though more as a side-effect than because of the facade, we can change the type of repository to flat file or something else without changing any other subsystems. All we would have to do is create a different implementation of the Database interface.

Another suggested use for the facade pattern in Gamma *et al* is to decrease coupling when layering subsystems. When there are many dependencies between two subsystems, you can define a facade in each and make all inter-subsystem communication go through the facades. We have several facade-like objects that behave in this way, however we are unsure if they actually qualify as facades, since they *may* implement some small amount of logic that goes beyond just translating calls from one interface to another, perhaps for maintaining consistency of cached objects or to provide notification when something in Storage changes.

We use facade-like objects to simplify communication between the application layer and the repository layer:

- the User Management subsystem has a User Manager class
- the Admin Management subsystem has an Admin Manager class
- the Student Management subsystem has a Student Manager class
- the Algorithm Manager subsystem has an Algorithm Manager class
- in the repository layer, the Storage subsystem has a Storage Manager class

Singleton

The singleton pattern is used when you need to ensure that there is only ever one instance of a class, and you want to provide a global point of access to it. It is often used with the facade pattern because you rarely want multiple instances of a facade. Our cuPID design uses the singleton pattern for:

- the Storage Manager class
- the Database interface (SqliteDatabase implementation)

The Storage Manager is a facade-like object that provides a single, well-known, global point of access to Storage-related features. Because the Storage Manager is also responsible for maintaining a consistent cache of entity objects, it is important that there is only ever one Storage Manager. If we allowed different subsystems to make their own Storage Manager instance, the caching would be ineffective at best, and at worst the caches would become out of date with no way of detecting when it happens.

“Database” is an an interface implemented by SqliteDatabase, which also uses the facade pattern. We use a singleton pattern for the Database for two reasons: first, we want only one database connection and only one instance of the database class in the program. Second, we want the single instance to be extensible by subclassing, so that the SqliteDatabase could theoretically be replaced by a flat file system without changing the Storage subsystem.

If we had simply made the Database functions static, polymorphism would not work on them, so we would be unable to create different implementations of it. If we had made the Database a static data member of the Storage Manager, that still would not prevent multiple instances of the database from being instantiated, and it would force Storage Manager to depend on the concrete SqliteDatabase implementation.

There are ways described in Gamma *et al* to create a singleton registry that allows you to swap out the implementation of singletons. It it slightly more complicated than the traditional singleton pattern, but it would allow us to achieve our design goals of keeping a single database instance and maintaining separation between the Database and Storage subsystems.

The weakness of using Singleton for the Database is that access to it becomes global, and we really only want the Storage Manager accessing it.

We also used the singleton pattern for a few other facade or facade-like classes:

- User Manager
- Admin Manager
- Student Manager
- Algorithm Manager

However there is only a slight benefit to having only one instance of those classes, and they do not need global access. But they have to coordinate between different classes, there should be only one instance of them, and it’s not clear what should be responsible for creating that single instance or how it should be passed around if it isn’t a singleton, so we decided to use a singleton.

Observer (not used)

We rejected the idea of using an observer pattern. It looked like there might have been a place to use it, since the Student Home Window has to update its list of which projects the student is in when the student joins a project (which happens in the Join Project Control). There is a similar situation in the Admin Home Window where the list of projects needs to change when the admin adds a project (in Manage Project Control). So there is some slight justification to set up subscribe/notify functions.

We decided against it because the observer pattern isn’t just about subscribe/notify; it’s about abstracting subscribe and notify in case you don’t know how many objects you will need to notify or you don’t want to make assumptions about what kinds of objects you will need to notify. In our case, however, we know exactly what objects need to be notified. Furthermore, the objects are within the same subsystem in each case, so having tight coupling between them is not much of a concern.

We keep our design simpler here by avoiding an unnecessary design pattern.

Subsystem services

When designing the phase 2 cuPID system, we used the idea of subsystem services to gain a better understanding of the dependencies between subsystems. This section details the services offered by each subsystem.

Four of our subsystems provide services to other subsystems.

- Storage
- Database
- AlgorithmManagement
- StudentManagement

Most of the services are provided by Storage subsystem. It manages passing data from the Management subsystems to the Database subsystem. The Database subsystem provides a connection to the persistent data repository. It retrieves, saves, and updates data. The AlgorithmManagement subsystem provides the AdminManagement subsystem with access to the Team Match feature. The StudentManagement subsystem provides the UserManagement subsystem with the Profile Creation feature.

SSL-01: List of the services provided

SS#	**Subsystem: Service**
SS-01	Storage: ObjectVerification
SS-02	Storage: ProjectList
SS-03	Storage: JoinedProjectList
SS-04	Storage: ProjectJoin
SS-05	Storage: ObjectDataRetrieval
SS-06	Storage: UserAccountCreation
SS-07	Storage: ProfileCreation
SS-08	Storage: ProfileEditing
SS-09	Storage: ProjectCreation
SS-10	Storage: ProjectEditing
SS-11	Database: ObjectSave
SS-12	Database: ObjectUpdate
SS-13	Database: ObjectRetrieval
SS-14	AlgorithmManagement: TeamMatch
SS-15	StudentManagement: ProfileCreation

Services provided by Storage subsystem

Service #	SS-01
Service Name	**ObjectVerification**
Offering Subsystem	Storage
Offering Class	StorageManager
Participating Subsystems	UserManagement StudentManagement AdminManagement Database
Operation Description	Storage repository gets a request from one of Management subsystems with object information and asks Database subsystem to check if that object exists in the repository. Database subsystem returns the result to Management subsystem through Storage subsystem.
Traceability	NF-08, UC-19, S-14

Service #	SS-02
Service Name	**ProjectList**
Offering Subsystem	Storage
Offering Class	StorageManager
Participating Subsystems	AdminManagement Database
Operation Description	Storage subsystem gets a request to provide a list of projects and redirects it to Database subsystem. Database subsystem obtain a list from the repository and returns it through Storage subsystem.
Traceability	F-06, UC-04, UC-13, UC-19, S-09, S-14

Service #	SS-03
Service Name	**JoinedProjectList**
Offering Subsystem	Storage

Offering Class	StorageManager
Participating Subsystems	Database StudentManagement
Operation Description	StudentManagement request a list of joined projects from Storage repository. Storage repository asks Database subsystem, which gets the list of joined projects from the repository, and returns the list to Management subsystem.
Traceability	F-03, UC-13, UC-19, S-09, S-14

Service #	SS-04
Service Name	**ProjectJoin**
Offering Subsystem	Storage
Offering Class	StorageManager
Participating Subsystems	StudentManagement Database
Operation Description	StudentManagement subsystem passes student's and project's IDs to Storage subsystem. Storage subsystem redirects the request to Database subsystem, which adds the information to the repository.
Traceability	F-01, UC-03, UC-19, S-05, S-14

Service #	SS-05
Service Name	**ObjectDataRetrieval**
Offering Subsystem	Storage
Offering Class	StorageManager
Participating Subsystems	Database UserManagement StudentManagement AlgorithmManagement AdminManagement

Operation Description	Storage subsystem receives a request from one of Management subsystems and if requested object cached in it, it returns a copy of the object, otherwise it asks Database storage for the object, saves it and returns a copy to Management subsystem.
Traceability	UC-19, S-14

Service #	SS-07
Service Name	**UserAccountCreation**
Offering Subsystem	Storage
Offering Class	StorageManager
Participating Subsystems	UserManagement Database
Operation Description	UserManagement subsystem sends a request to create a new student user with required information to Storage subsystem. Storage subsystem passes the information to Database subsystem, which adds the new user to the repository.
Traceability	F-10, F-11, F-12, NF-14, UC-01, UC-06, UC-07, UC-19, S-01, S-02, S-03, S-14

Service #	SS-08
Service Name	**ProfileCreation**
Offering Subsystem	Storage
Offering Class	StorageManager
Participating Subsystems	StudentManagement Database
Operation Description	StudentManagement subsystem requests Storage subsystem to create new profile and passes student’s ID and profile information. Storage subsystem redirects the request to

	Database subsystem, which adds the profile to the repository.
Traceability	NF-02, NF-08, NF-14, NF-32, NF-12, UC-08, UC-19, S-14

Service #	SS-11
Service Name	**ProfileEditing**
Offering Subsystem	Storage
Offering Class	StorageManagement
Participating Subsystems	StudentManagement Database
Operation Description	StudentManagement subsystem requests Storage subsystem to create a profile and passes profile’s ID and information. Storage subsystem redirects the request to Database subsystem, which updates the profile.
Traceability	NF-02, NF-14, UC-02, UC-10, UC-19, S-04, S-14

Service #	SS-09
Service Name	**ProjectCreation**
Offering Subsystem	Storage
Offering Class	StorageManagement
Participating Subsystems	AdminManagement Database
Operation Description	AdminManagement subsystem passes project information to Storage subsystem. Storage subsystem requests Database to add new project to the repository and passes the information to it. Database subsystem adds the project to the repository and returns operation result.
Traceability	F-07, NF-14, UC-09, UC-19, S-07, S-14

Service #	SS-10

Service Name	**ProjectEditing**
Offering Subsystem	Storage
Offering Class	StorageManager
Participating Subsystems	AdminManagement Database
Operation Description	AdminManagement subsystem passes project information to Storage subsystem. Storage subsystem requests Database to add new project to the repository and passes the information to it. Database subsystem adds the project to the repository and returns operation result.
Traceability	F-04, Nf-14, UC-19, S-08, S-14

Services provided by Database subsystem

Service #	SS-11
Service Name	**ObjectSave**
Offering Subsystem	Database
Offering Class	SqliteDatabase
Participating Subsystems	Storage
Operation Description	Storage subsystem passes an information about new project that is needed to be saved to Database subsystem. Database subsystem sends a command to the storage.
Traceability	UC-19, S-14

Service #	SS-12
Service Name	**ObjectUpdate**
Offering Subsystem	Database
Offering Class	SqliteDatabase
Participating Subsystems	Storage
Operation Description	Storage subsystem passes object’s ID and update information to Database. Database subsystems sends a command to the storage.

Traceability	NF-14, UC-19, S-14

Service #	SS-13
Service Name	**ObjectRetrieval**
Offering Subsystem	Database
Offering Class	SqliteDatabase
Participating Subsystems	Storage
Operation Description	Storage subsystem passes object's ID to Database subsystem. Database subsystem retrieves object's information from the storage and sends it back to Storage subsystem.
Traceability	UC-19, S-14

Services provided by AdminManagement subsystem

Service #	SS-14
Service Name	**TeamMatch**
Offering Subsystem	AlgorithmManager
Offering Class	AlgorithmManager
Participating Subsystems	AdminManager
Operation Description	AdminManager subsystem requests AlgorithmManager subsystem to compute a set of teams for selected project.
Traceability	F-05, F-08, F-09, NF-13, UC-05, S-06

Services provided by StudentManagement subsystem

Service #	SS-15
Service Name	**ProfileCreation**
Offering Subsystem	StudentManagement
Offering Class	ManageProfileControl

Participating Subsystems	UserManagement
Operation Description	UserManagement requests StudentManagement to create new profile and passes student's ID to it.
Traceability	Nf-02

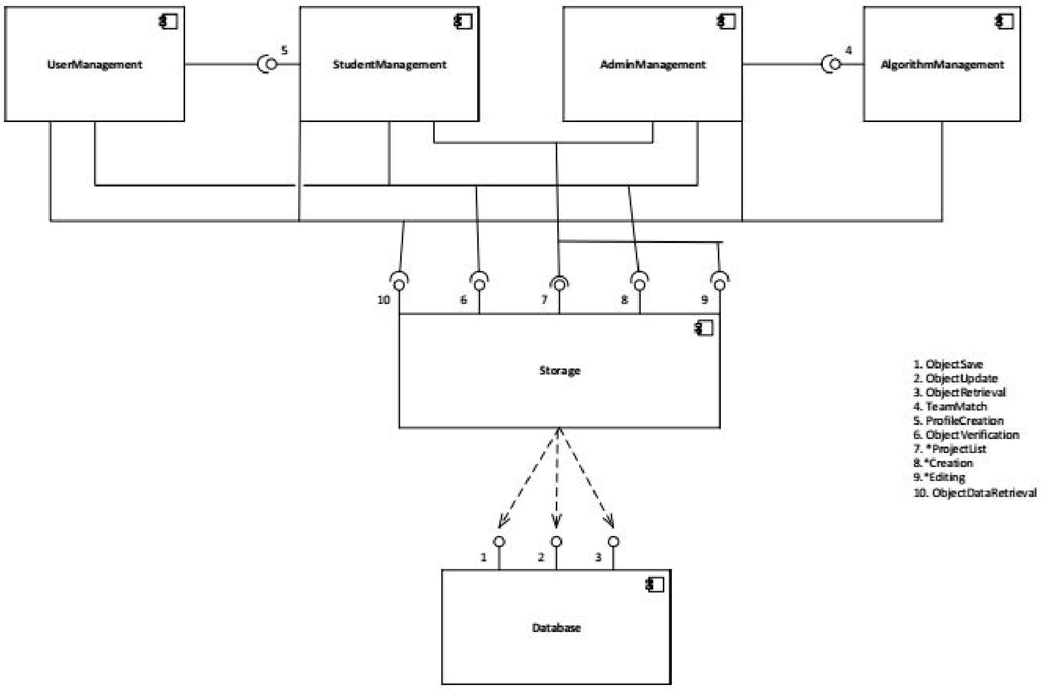


Figure 11: UML Component Diagram

Services on the diagram:

1. ObjectSave -- SS-11
2. ObjectUpdate -- SS-12
3. ObjectRetrieval -- SS-13
4. TeamMatch -- SS-14
5. ProfileCreation -- SS-15
6. ObjectVerification -- SS-01
7. *ProjectList: ProjectList -- SS-02, JoinedProjectList -- SS-03
8. *Creation: ProfileCreation -- SS-07, ProjectCreation -- SS-09
9. *Editing: ProfileEditing -- SS-08, ProjectEditing -- SS-10

10. ObjectDataRetrieval:

Class interfaces

This section contains UML class diagram showing the interfaces of every class that offers a service.

The following diagram shows the interfaces offered by the Database subsystem to the Storage subsystem:

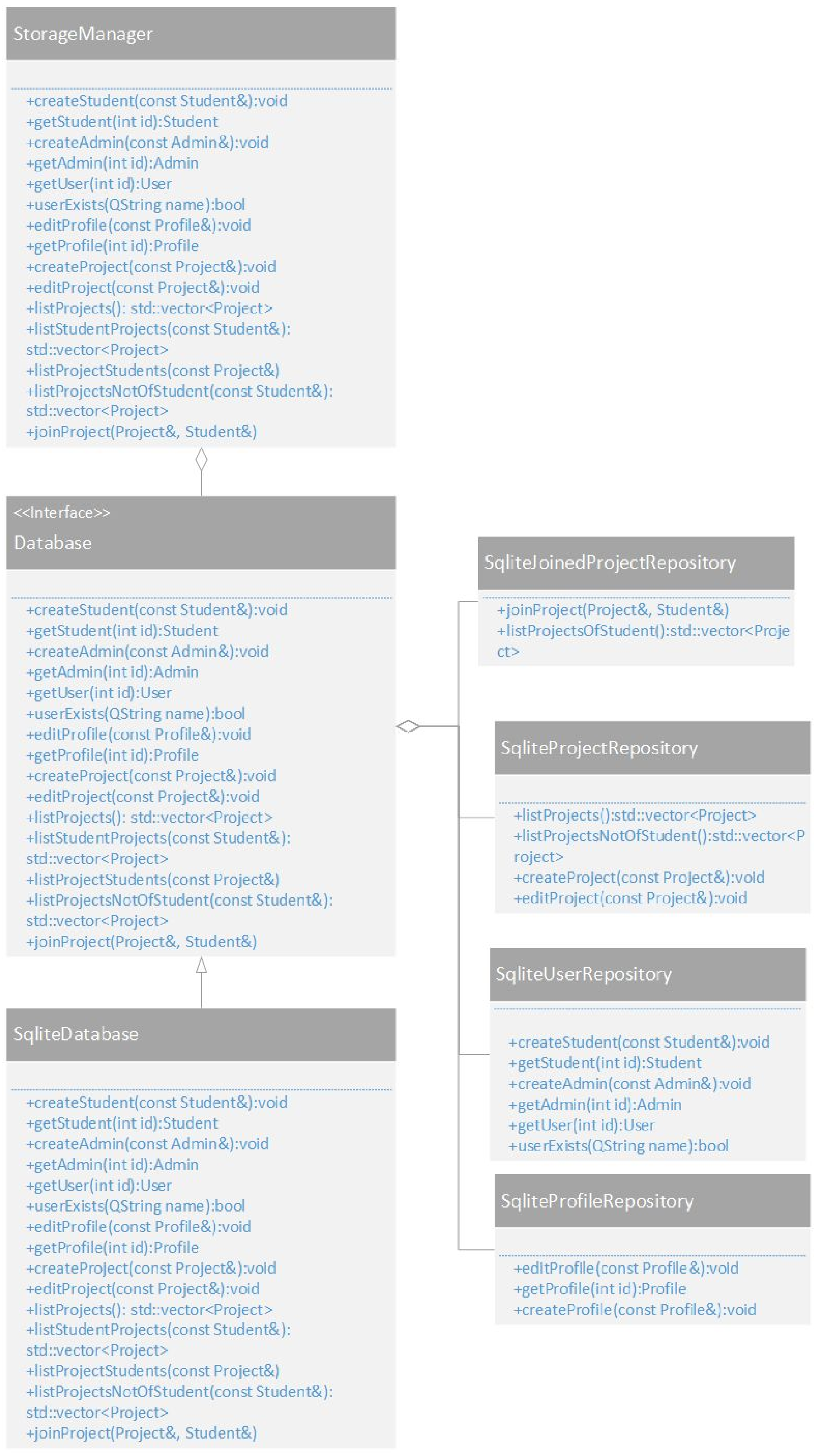

StorageManager
+createStudent(const Student&):void
+getStudent(int id):Student
+createAdmin(const Admin&):void
+getAdmin(int id):Admin
+getUser(int id):User
+userExists(QString name):bool
+editProfile(const Profile&):void
+getProfile(int id):Profile
+createProject(const Project&):void
+editProject(const Project&):void
+listProjects(): std::vector<Project>
+listStudentProjects(const Student&):
std::vector<Project>
+listProjectStudents(const Project&)
+listProjectsNotOfStudent(const Student&):
std::vector<Project>
+joinProject(Project&, Student&)
<<Interface>>
Database
+createStudent(const Student&):void
+getStudent(int id):Student
+createAdmin(const Admin&):void
+getAdmin(int id):Admin
+getUser(int id):User
+userExists(QString name):bool
+editProfile(const Profile&):void
+getProfile(int id):Profile
+createProject(const Project&):void
+editProject(const Project&):void
+listProjects(): std::vector<Project>
+listStudentProjects(const Student&):
std::vector<Project>
+listProjectStudents(const Project&)
+listProjectsNotOfStudent(const Student&):
std::vector<Project>
+joinProject(Project&, Student&)
SqliteJoinedProjectRepository
+joinProject(Project&, Student&)
+listProjectsOfStudent():std::vector<Proje
ct>
SqliteProjectRepository
+listProjects():std::vector<Project>
+listProjectsNotOfStudent():std::vector<P
roject>
+createProject(const Project&):void
+editProject(const Project&):void
SqliteUserRepository
+createStudent(const Student&):void
+getStudent(int id):Student
+createAdmin(const Admin&):void
+getAdmin(int id):Admin
+getUser(int id):User
+userExists(QString name):bool
SqliteProfileRepository
+editProfile(const Profile&):void
+getProfile(int id):Profile
+createProfile(const Profile&):void
SqliteDatabase
+createStudent(const Student&):void
+getStudent(int id):Student
+createAdmin(const Admin&):void
+getAdmin(int id):Admin
+getUser(int id):User
+userExists(QString name):bool
+editProfile(const Profile&):void
+getProfile(int id):Profile
+createProject(const Project&):void
+editProject(const Project&):void
+listProjects(): std::vector<Project>
+listStudentProjects(const Student&):
std::vector<Project>
+listProjectStudents(const Project&)
+listProjectsNotOfStudent(const Student&):
std::vector<Project>
+joinProject(Project&, Student&)

Figure 12: UML Class Diagram for services offered by Database subsystem

In Figure 12, it is the Database class (interface) offering services to the StorageManager (in Storage subsystem).

The following diagram shows the interfaces offered by the Storage subsystem to the other subsystems:

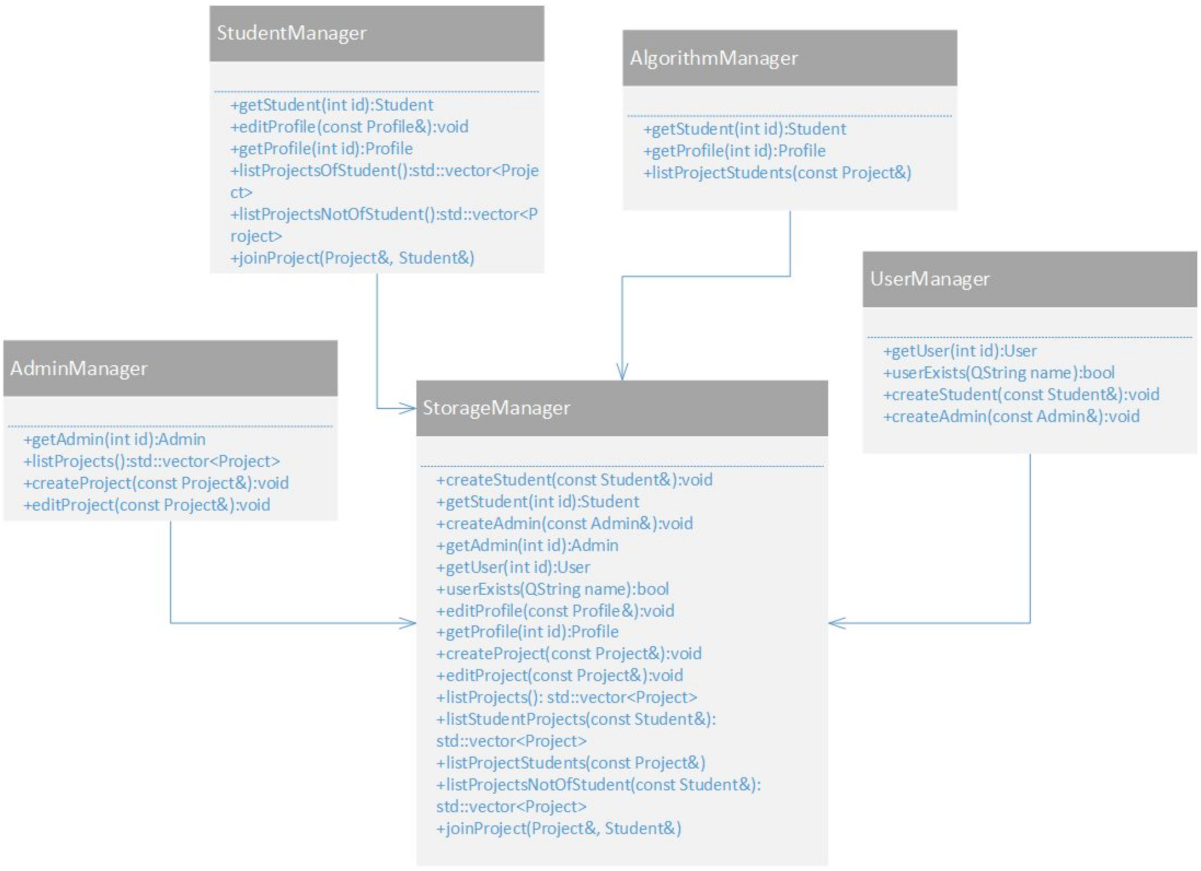

StudentManager
+getStudent(int id):Student
+editProfile(const Profile&):void
+getProfile(int id):Profile
+listProjectsOfStudent():std::vector<Project>
+listProjectsNotOfStudent():std::vector<Project>
+joinProject(Project&, Student&)
AlgorithmManager
+getStudent(int id):Student
+getProfile(int id):Profile
+listProjectStudents(const Project&)
UserManager
+getUser(int id):User
+userExists(QString name):bool
+createStudent(const Student&):void
+createAdmin(const Admin&):void
AdminManager
+getAdmin(int id):Admin
+listProjects():std::vector<Project>
+createProject(const Project&):void
+editProject(const Project&):void
StorageManager
+createStudent(const Student&):void
+getStudent(int id):Student
+createAdmin(const Admin&):void
+getAdmin(int id):Admin
+getUser(int id):User
+userExists(QString name):bool
+editProfile(const Profile&):void
+getProfile(int id):Profile
+createProject(const Project&):void
+editProject(const Project&):void
+listProjects(): std::vector<Project>
+listStudentProjects(const Student&): std::vector<Project>
+listProjectStudents(const Project&)
+listProjectsNotOfStudent(const Student&): std::vector<Project>
+joinProject(Project&, Student&)

Figure 13: UML Class Diagram for services offered by Storage Subsystem

In Figure 13, it is the Storage Manager (in Storage subsystem) offering services to other subsystems.

Those two diagrams cover almost all of the services.