CARLETON UNIVERSITY

(Carleton University Project Partner Identifier)

System Design Document

Team Insomnia

Dabeluchi Ndubisi
Charlie Li
Pierre Seguin

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

Table of Contents

Table of Figures

Section 1: Introduction

The Carleton University Project Partner Identifier (cuPID) is a single host, data driven application used by AdministratorUsers (professors) to generate teams of StudentUsers who are most compatible for a single project that he/she created. This document is intended for the development team to build more than just a mental model of the application, but to challenge design decision process and to bring out the most prominent design solution. The baseline of this document will build up from the cuPID prototype that was recently demonstrated to the client and will be thoroughly cross referencing some attributes and behaviours from that submitted system. The sections of this document will describe the evolution of the prototype and system as the development team is tasked to analyze and decompose the system into its atomic subsystems. In conjunction to further elaborating on the design goals set by the project, different design strategies will be audited in this document to weigh in on it's pros and cons.

This document is composed of four main components and a definitive overview is provided for the reader's perusal in this introductory section:

The first component of this document (*Section 2*) is the *subsystem decomposition*. In this section, a detailed description of the logical subsystem implemented as a prototype for the client is provided. The representations of the subsystems are illustrated with UML class diagrams and packages in order to provide the reader with a substantial mental model of the prototype system in question. Also in this section, the decomposition of the *proper* cuPID system is provided. By proper, we mean the best admissible designed architecture and subsystem relationships that the cuPID system should be composed of. Finally, in this section, a comprehensive design evolution is provided in order to highlight the discrepancies observed between the prototype system provided to the client and how the *proper* system should be represented. The essence of this is to scrutinize the development process with keen eyes in order to observe flaws and faults that may arise due to the prototype system representation.

The second section (*Section 3*) of this document is set to provide descriptions of the design strategies employed in the development life cycle of the cuPID system. In this section, design concepts such as architectural styles, design patterns and persistent data management are discussed with respect to the context of the cuPID system. The essence of this section is to provide a mapping context of how the cuPID subsystems interact with both external components (hardware) and themselves (internal software components).

The next section (*Section 4*) of the documents highlights the services provided by each subsystem that the cuPID system contains. Detailed descriptions of the operations within each service, including classes to which each operation belongs are highlighted in this document. The representation of the services are illustrated with the UML ball-and-socket notation for component diagrams.

The next section (*Section 5*) of the document provides a detailed description of each class interface that is involved in the provision of operations for a particular service. For the sake of conductive organization, these class interfaces are grouped by the services that they provide. For easy readability and organization, traceability of concepts have been provided where applicable.

The last section is an Appendix section which contains abbreviations, tables, figures and other necessary information necessary to aid the easy assimilation of the information presented in this document.

Finally, it is worth noting that all design decision provided in this document were thoroughly scrutinized and debated in order to maintain a coherence with the **Requirement Analysis Document** provided to the client.

Section 2: Subsystem Decomposition

In this section of the document, the reader is first provided with a decomposition of the prototype cuPID system provided to the client. This is done to establish a conceptual model of the subsystems in the prototype so that a baseline can be obtained as the how the cuPID system should be comprised. Also in this section as a means of the contrast, the reader is then provided with the decomposition of the *ideal* cuPID system. This decomposition outlines the proper design architecture and subsystem dependencies of the cuPID system as a whole with all functionalities provided as opposed to the prototype decomposition. Finally, a comparison of the two decompositions is provided to the user in order to highlight the changes and reasons for such design decisions.

2.1 Phase #1 Prototype Decomposition

In this phase, we will break down the cuPID prototype to identify the logical subsystems that were present in the prototype build followed by their UML class diagrams representing associations and relationships among the classes of the subsystem. During the process of developing this build, special care was taken to group common functionality into ‘packages’ of their relation. This section will depict the general application decomposition in conjunction to the detailed breakdown of each package that was created for cuPID.

The submitted prototype of the cuPID system embodies four subsystems: *StorageSubsystem* (*NFR-05, NFR-06*), *ProfileManagementSubsystem* (*FR-01*), *ProjectManagementSubsystem* (*FR-02, FR-03, NFR-23*), and *UserAuthenticationSubsystem* (*NFR-21*). The idea behind this approach was to group together common components of the cuPID system together such all objects or classes that participate in providing a particular functionality were grouped together. This way allowed us to form a better conceptual model of what kind of services were to be provided by a particular subsystem. Next, a description of each subsystem and their role in the cuPID system is provided for the reader’s benefit.

The first subsystem to be discussed is the *StorageSubsystem.* The *StorageSubsystem* embodies all the entities and logic behind storing and retrieving persistent data used to populate entity objects required for other subsystems to function. These entities include: Project (*EO-04*), Configuration (*EO-05*), ProjectPartnerProfile (*EO-08*), Qualification (*EO-09*). The classes that make up the *StorageSubsystem* are enlisted as follows: Storage, DatabaseManager, UserRepository, ProjectRepository and cuPIDSession. The Storage class was designed to act as a *Facade* class that provided a list of Storage functionalities to other classes that needed it. Also, during the course of the design, a decision was made for the Storage class to be implemented with the *Singleton* design pattern in mind. Hence, only one instance of this class is available throughout the cuPID system. Since only one user is allowed to be logged into the cuPID system at any given time, this constraints helps neutralize any issues that might have arised due to deadlocks on this class. The DatabaseManager class was designed to handle the operations required for accessing the external database system. The UserRepository class was designed to handle all operations that were specific to users of the cuPID system, and the ProjectRepository class was designed to handle all operations that were specific to interactions with projects in the cuPID system. Finally, the cuPID session class was designed as a means of caching such that expensive requests such as retrieval of the user's current profile or the current project that a user was currently interacting with did not have to be executed countless times. Hence this serves as an efficiency booster for the *StorageSubsystem*.
Figure 1 shows the illustration of the Storage subsystem as a UML class diagram.

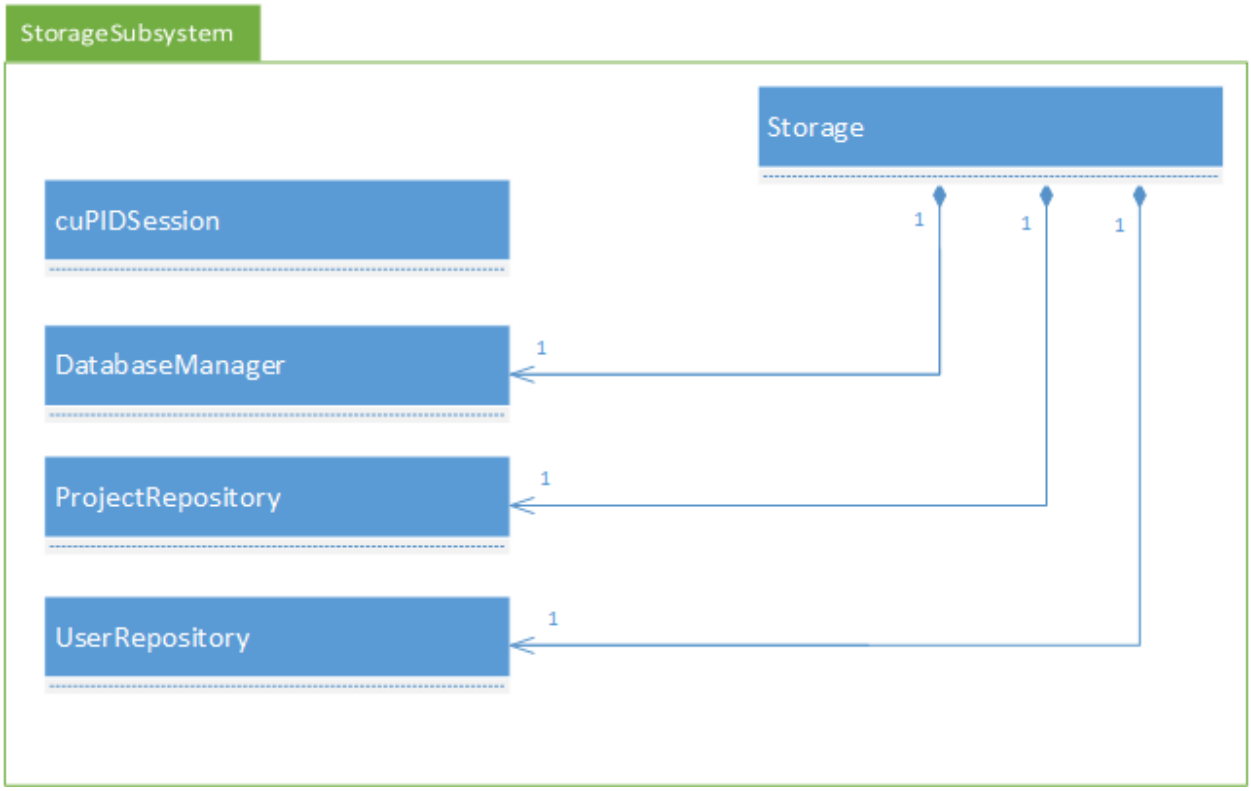


Figure 1 - UML Class Diagram for fhe Storage Subsystem

The next subsystem to be discussed is the *ProfileManagementSubsystem.* The *ProfileManagementSubsystem* embodies all the entity objects, boundary objects and logic behind interacting with profiles in the cuPID system (*FR-01*). The classes that make up the *ProfileManagementSubsystem* are enlisted as follows: ProjectPartnerProfileController(PPPController), Qualification (*EO-09*), ProfileWidget, ProjectPartnerProfile(PPP) (*EO-08*). The PPPController was designed to handle all interactions that a StudentUser can execute on his/her PPP. These interactions are inline with the Use-Cases from the Requirement Analysis Document, and they include: EditProjectPartnerProfile (*UC-04*), CreateProjectPartnerProfile (*UC-05*), SaveProjectPartnerProfile (*UC-05*). The PPPController sets up the ProfileWidget boundary object for the particular interaction that the StudentUser is about to execute. The Qualification class is a class that holds a type of qualification needed by our algorithm and also a value that is associated with the qualification. The ProfileWidget class serves as a multipurpose boundary object that adapts based on the particular functionality that the StudentUser is to execute at any given time. Lastly, the ProjectPartnerProfile class is an entity object that holds the data populated by the StudentUser while creating or editing his/her profile for the cuPID system. Figure 2 shows the illustration of the ProfileManagement subsystem as a UML class diagram.

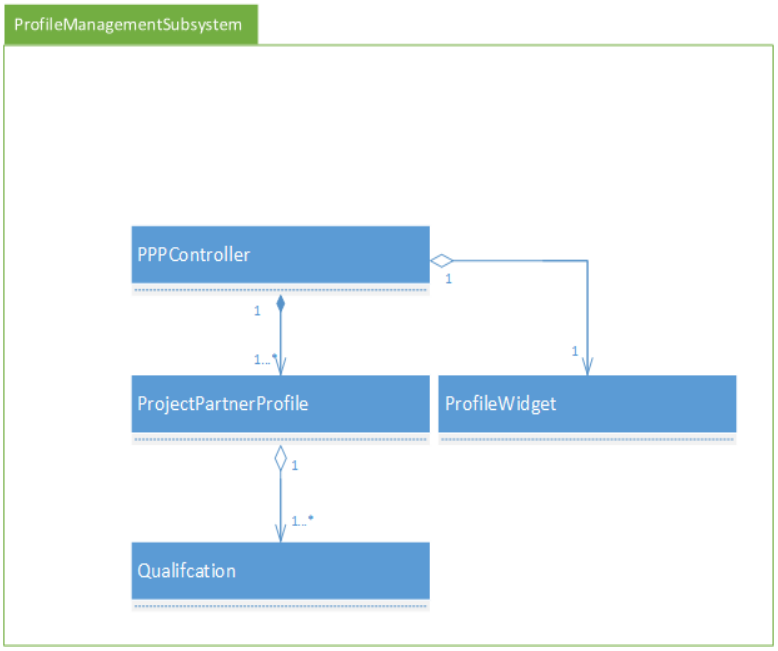


Figure 2 - UML Class Diagram for the ProfileManagement Subsystem

The third subsystem in the submitted prototype cuPID application is the *ProjectManagementSubsystem*. This subsystem contains all the entity objects, boundary objects and logic behind interacting with projects in the cuPID system (*FR-02, FR-03*). The classes that make up this subsystem are enlisted as follows: Project (*EO-04*), Configuration (EO-05), CreateProjectWidget, EditProjectDialog, ProjectListWidget, ProjectDetailsWidget. The Project class is an entity object that holds all data related to a particular project. The Configuration class is also an entity object that holds a type and value of configuration related to the project in question. The CreateProjectWidget is a boundary object that provides the AdministratorUser with the ability to create a project (*UC-09*). The EditProjectDialog is a boundary object that provides the AdministratorUser with the ability to edit an already created project (*UC-10*). The ProjectListWidget is also a boundary object that provides both the StudentUser and the AdministratorUser with a means to view a list of projects available to them in the cuPID system. Finally, the ProjectDetailsWidget is a boundary object that provides both the StudentUser and AdministratorUser with a means to interact with a particular project currently in view. These interactions include: RegisterForProject (*UC-12*), UnregisterFromProject (*UC-11*), and EditProjectConfigurations (UC-10). It should be pointed out that the boundary objects in this class also handle the logic associated with updating entity objects that they are involved with. Figure 3 shows the illustration of the ProjectManagement subsystem as a UML class diagram.

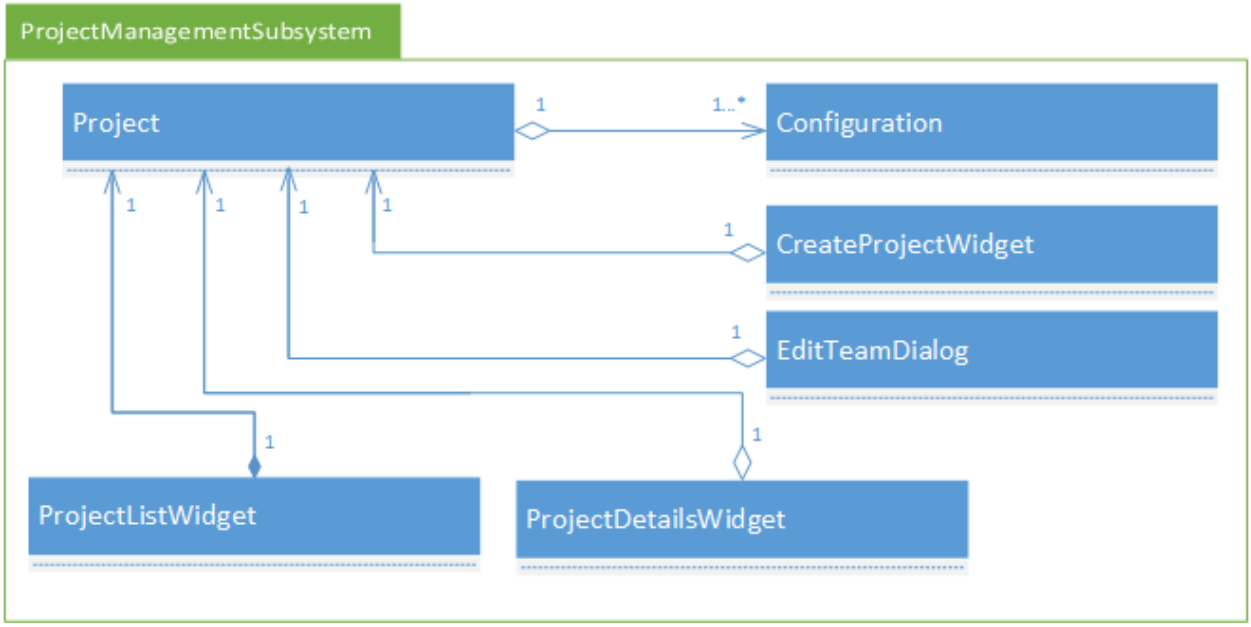


Figure 3 - UML Class Diagram for the ProjectManagement Subsystem

The final subsystem in the cuPID prototype system is the *UserAuthenticationSubsystem.* This subsystem is responsible for Authenticating a user into the cuPID system. It contains the following classes: User (EO-01), StudentUser (EO-03), AdministratorUser (EO-02), LoginForm, and SignUpForm. The User class is an entity class that represents the typical user of the cuPID system. This class is inherited by the AdministratorUser and StudentUser in order to provide specialized activities unique to the type of user. The LoginForm is a boundary object that provides the user with a means to login to the cuPID system. The SignUpForm is a boundary object that provides a user with a means of signing up for the cuPID system. Also in this subsystem, it should be noted that the boundary objects in this class also handle the logic associated with updating entity objects that they are involved with. Figure 4 shows the illustration of the UserAuthentication subsystem as a UML class diagram.

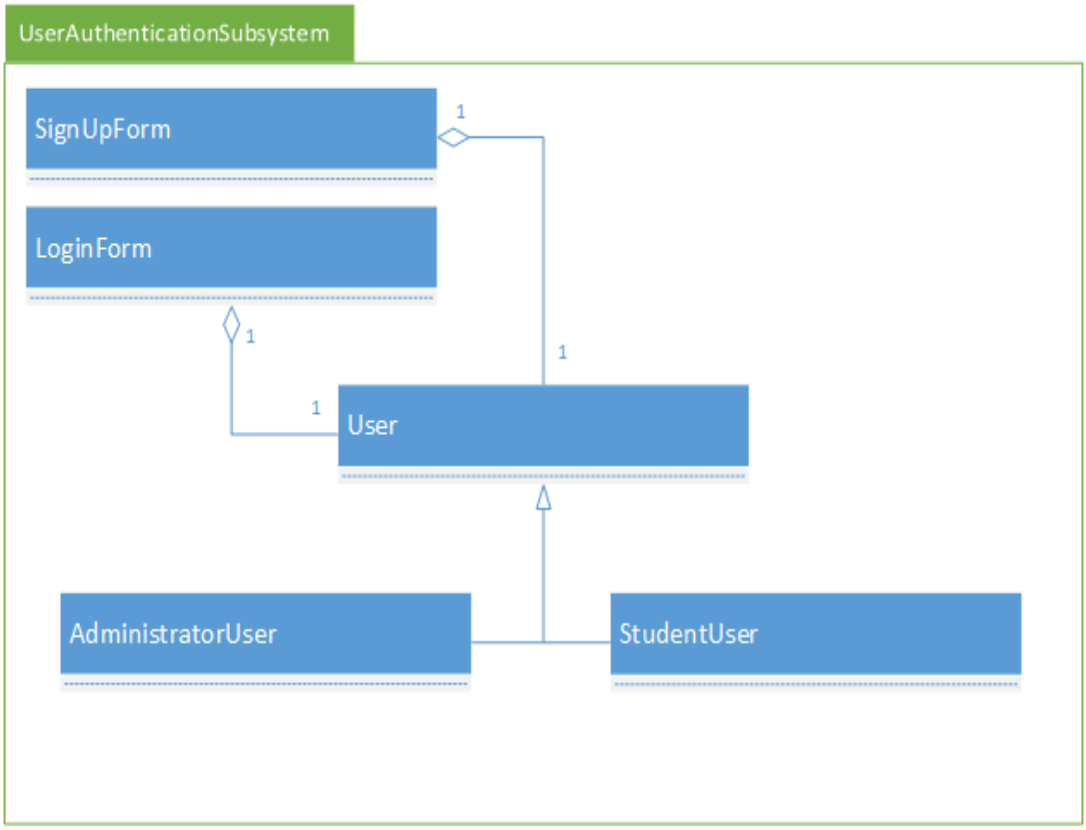


Figure 4 - UML Class Diagram for the UserAuthentication Subsystem

Table 1 in the *Appendix Section* of the document provides the reader with a summary of the subsystems that make up the prototype cuPID system and the respective classes that they comprise of.

Next the reader is provided with the full representation of the subsystems that comprise the cuPID prototype system provided to the client. This representation is described with the aid of UML class diagrams. For ease of identification, our subsystems are represented with packages in the class diagram provided. Figure 5 provides the user with the full representation of the cuPID system.

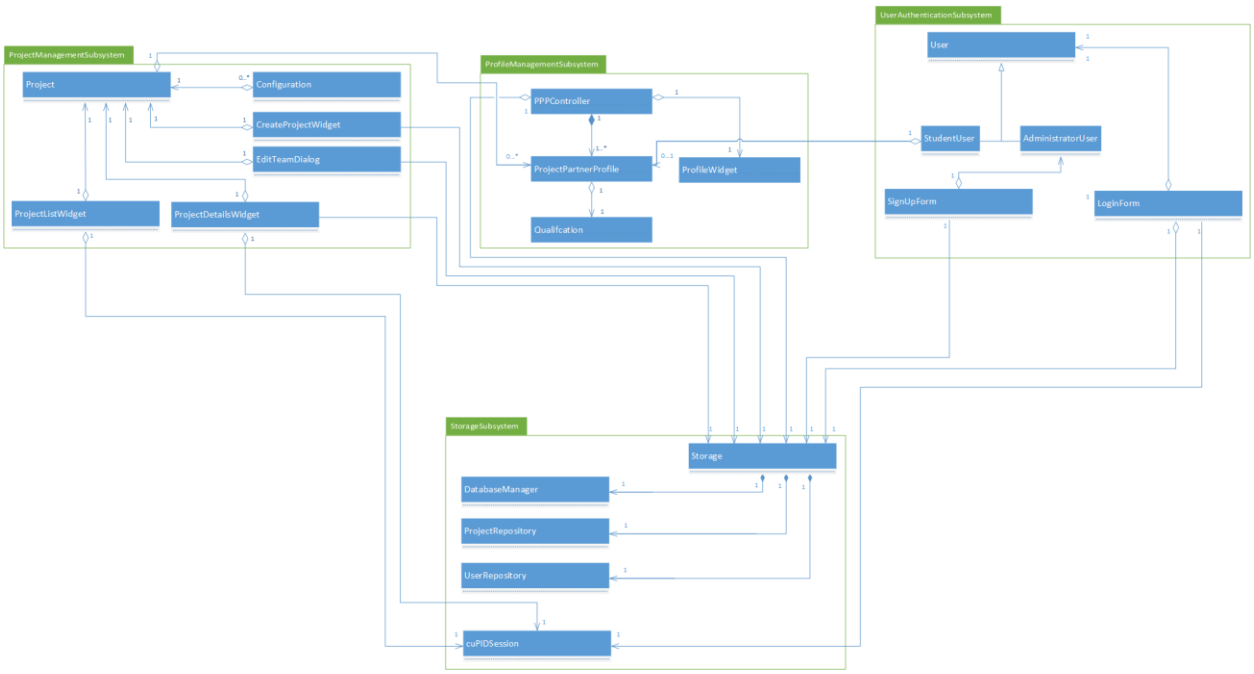


Figure 5 - UML class diagram of the cuPID prototype Application showing Subsystems

In order to illustrate the dependencies between the subsystems, we provide the reader the the UML component diagram of the the prototype subsystem. As stated, this diagram shows the dependencies and relationships between subsystems of the prototype system. This in turn helps identify interesting design criteria such as cohesion and coupling between the subsystems of the cuPID system. Figure 6 provides the user with the described representation of the dependencies between the subsystems mentioned.

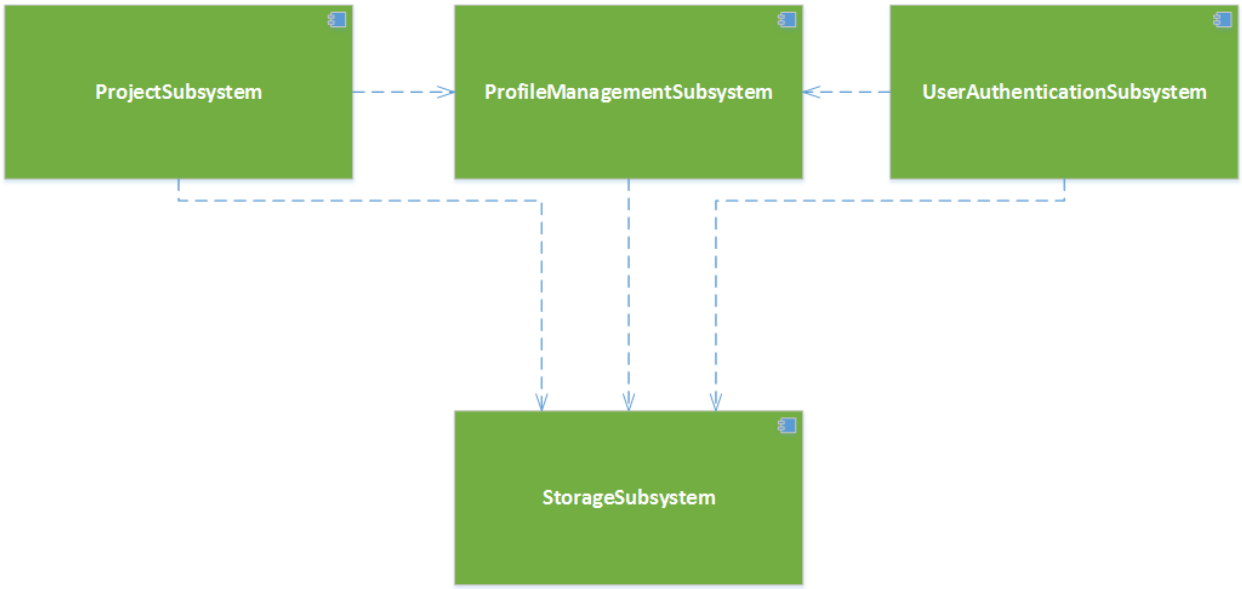


Figure 6 - UML component diagram of the cuPID prototype Application showing Subsystem dependencies

2.2 System Decomposition

This phase of the subsystem decomposition draws a target for the new, reworked cuPID system as our team will prospect through the best and proper design choices to transform the prototype build into the final product. Care has been taken to specify system functionalities from the requirements analysis phase, by transforming them into the logical subsystems that promotes high cohesiveness and low coupling. Among the design choices we state in this section, we aim to build a system that is client centric and formulate our decisions around the support and aid of the final end user interacting with our system. Our first subsection will address the Design Goals we've chosen based on gathered experience over developing the prototype and also with respect to the stated Non-Functional requirements in the Requirement Analysis Document, then in the Subsystem Decomposition subsection, we will show how each design goal maps onto each subsystem(s) for the cuPID system.

Design Goals

As we approach this step of the document, it's worth mentioning our reasoning and goals for our system decomposition. Specific design decisions were made in this section and may conclude in biased results towards a specific behaviour, therefore, we will depict the trades offs the development team made in order to fulfill these decisions. These goals are prioritized by the order each one of the five goals have been declared by the paragraphs below.

Design Goal: Reliability (DG-1)

Our number one goal for cuPID, a system that supports many student's profiles, an increasing number each year, is the aspect of system reliability. In many possible definitions, reliability can specifically defined as a system operating with deterministic behaviours, one hundred percent of its time. This goal is recognized through the ability of the system to prevent harmful input from infecting the it's data source (includes NFR-05, NFR-06, NFR-08, UC-18). Our system must be aware of it's faults, especially when performing IO operations and always designated a fall back plan, that does not include crash and die, in the case of an exceptional error (NFR-08).

Design Goal: Flexibility (DG-2)

Our second design goal pertains to the flexibility in the our system design in order to enable a modular system composition and modifiable architecture. We know the external environment is constantly changing so the architecture of cuPID should allow easy modification or replacement of module subsystems such as switching database from SQLite to another relational database for storing persistent data. A subsystem should also be extensible in order to provide an API for future web application version (includes NFR-14) and with our modular design, our system lowers costs for future maintenance and extensions. Within this flexible design, we will be describing our system decomposition below which will promote extensibility, modifiability, and traceability.

Design Goal: Performance (DG-3)

Our team believes the third way we can direct our client centric focus is by setting standards for performance within cuPID. We must ensure that the application is responsive at

all times (includes NFR-03, NFR-09, NFR-10, NFR-11) during heavy computations or IO operations to the database. The cuPID must be space efficient at boot by loading full objects only when necessary. After sometime, cuPID will trade off space efficiency for speed by implementing a caching layer to avoid round trips to the database. The system's indicators for cuPID are directly related to the speed of its operations and masking complexity. Our strategy to tackle performance related issues is to implement a form of lazy loading for core and non core entity objects, preprocessing to equalize the heavy computational load when needed, and model possible problems to bit level for faster and lighter computations. This is achieved using the *Proxy* design pattern where applicable

Design Goal: Usability (DG-4)

Although the last three design goals targeted towards the system operations, our fourth design goal is targeted at the end user. We must simplify the system interactions for the end sure by providing an easy to operate (NFR-02), well documented (includes NFR-01) and hintful environment (includes NFR-04) that new students each year can easily pick up and use instantly. The strategy to meet these goals will be developing a fluid set of user interfaces with pale comforting colours to view. Care has been taken to reduce the amount of mouse clicks and movement to accomplish important and frequent tasks in cuPID while providing a simplistic non-crowded interface. The trade offs for this type of focus will provide a unique challenge to the development team to build user interfaces that mask complexities some particular task and limit the total number of user interactions.

Design Goal: Code Reusability (DG-5)

The last design goal is dedicated to the developers, who will labour endlessly, through daylight and moonlight, sleepless creatures who have to dig up another body of code to fix a tiny bug just to create... another one.

The purpose of this design goal is targeted to the developers who have to read code and write changes within them because the client needs them within a short period of time. Our goal here is to create readable, reusable code that is properly documented. We want to aim to write code in its full generality to promote the reuse of components throughout the cuPID system and strive to lower dependencies between them. The components can represent user interface elements, helper classes, and with proper documentation for the functionality, it will enable other programmers to quickly read and understand the written code. The trade offs our development team is willing to make is to lower the quantity of non-essential features in the beginning stages of the system development in order to produce a higher quality code base that is easy to build on and extend in the future.

In summary, the five designs goals used by the development team as a guide through the construction of the cuPID system are: *system reliability, system flexibility, system performance, simple interface, and reusabili*ty through the provision of well documented code.

Subsystem Decomposition

In order to obtain the best possible decomposition of the cuPID system, several architectures were considered from Three tier to MVC to Repository. The major goal was to establish a subsystem decomposition that promoted high cohesion between classes in a particular subsystem and also promote loose coupling between subsystems that make up the cuPID system. In this section, a high level of the considered architectural designs are depicted to the the user and also UML component diagrams have been added in order to aid the creation of a mental model of the considered architectural systems. Before the description of the considered architectural designs, it will be of great aid to outline all the classes that will comprise the cuPID system and also outline a brief description of their roles in the system. Table 2 of the *Appendix* section outlines these classes and traceability is provided where necessary. It should be noted that Table 2 of the *Appendix* section contains a revision of the data dictionary provided in the requirement analysis document. The reader is suggested to visit Table 2 of the *Appendix* section as he/she deems necessary.

Based on the classes specified in Table 2 of the *Appendix* section, attempts were made to decompose the cuPID system to adapt to various architectural styles, all attempted architectures are described below, and finally, the reader is presented with the chosen architecture along with the reasons for the decision to go with the proclaimed architectural style.

Attempted Subsystem Decomposition: System Architecture: 3-Tier (SA-01)

The 3-Tier architecture style is an approach to design a three layer subsystem to represent a data driven application. Each layer encompases a different aspect of the system where the top layer (closest to the user) is the interface layer. Sandwiched between the interface layer and the bottom layer is the application logic layer. At the bottom of this architecture style is the storage layer. The diagram (*Figure* 1 of the *Appendix* section) shows the UML component diagram for this subsystem architecture to refer to.

The interface layer is comprised of top level boundary objects that the user interacts with such as windows, forms, and user interface elements. The main objective of the interface layer is to communicate with the lower, application logic, layer by sending the results of user's actions such as mouse click events or form posts, as well as facilitate the displaying of information that came up from the application logic layer.

The application logic layer is made up of control and entity objects that and reflecting the logic of the application. Sometimes called the business logic layer, this layer facilitates the stream of information from user and to the information. The purpose of this layer is to process the business rules of the data and raise notifications required by the application.

The storage layer realizes the retrieval and persistence of information. The purpose of this layer is to facilitate queries provided by the application layer and return the necessary objects that are needed.

The cuPID system underwent a comprehensive audit for the 3-Tier architecture style during the system decomposition phase and was determined non ideal. The development team thought highly of the 3-tier structure due to it's well formed layering of specific components of the system, but this architecture would have proven to be more difficult to set up and maintain

throughout its layering. The following is a list of benefits that the 3-Tier architecture could prove as advantageous to the design of the cuPID system:

3T-B-1. The architecture promotes a clean layering mechanism for each logical subsystem characterized by the tier system where each layer can cache information on its own. This leads to better performance in the case that the interface layer can save already retrieved data.

3T-B-2. While application logic and storage layer are down for maintenance, interface layer's cache can still sufficiently process a limited amount of requests.

3T-B-3. Clear separation between view and logic layers which scale horizontally across each layer, ie. we can load balance the interface layer among many application logic layer instances (adding more machines/servers).

Although the 3-Tier architecture provided the team with some substantial benefits, some flaws were noticed. These flaws include:

3T-LM-1. Resources must be manually propagated from the bottom layer (storage) to the upper layer subsystems.

3T-LM-2. Interface and application logic that are closely related (perform similar tasks) are actually very far from each other in other subsystem.

3T-LM-3. Adding functionality into this design takes more effort and could possibly imply working across multiple subsystems

3T-LM-4. Since interface and application logic are in their own respective subsystems, debugging and maintenance span multiple locations in the code base.

3T-LM-5. The physical separation of the layers may affect the performance between the three tiers and overall system

As aforementioned above, the number of disadvantages outnumber the advantages. Due to the complex structure of the three-tier architectural style, the design benefits prove to be far exceeding the expectations of the cuPID system. The development team is determined to work in an agile manner (DG-5) and selecting this architectural style will not be beneficial to the system based non-functional requirement (NFR-15) for a single host system. Much of the benefits, *3TB-1* and *3TB-2*, cannot be applied to cuPID as there aren't any backup logic layers to replace the interface layer. Due to the high separation of layers, the point made by DG-5 on reusable components become obsolete as *3TLM-2* separates related components far from from view. Moreover, performance has been a primary concern for cuPID, stated in *NFR-08*, *NFR-09*, *NFR-10*, which by selecting this architecture style, cuPID cannot positively benefit from a physical separation of it's layer as indicated in *3TLM-1* and *3TLM-5*. Much more work will need to be done for the data to flow from the storage later to the user interface.

The representative interfaces for the subsystem partitions on each layer are tightly coupled across application logic and interface layers. This tight coupling is unnecessary for the cuPID system since if the partitions across the layers could be merged, then it would created smaller, but more subsystems that are highly cohesive. The logic changes of the the application layer need not to propagate to the upper layers in this manner. The cuPID development team attempts to resolve this issue in the final system architecture design.

On the positive notes of this architectural style, some design goals that really synergize with cuPID are reliability (DG-1) and usability (DG-3). Since the interface layer can have a cache layer of itself, cuPID can still operate when it's data source become unavailable leading to a reliable user interface in the case of backend fault or when backend maintenance is required to take place. Layering interface away from application logic can also be beneficial for the usability of cuPID since the the layer can be replaced by any UI library or web browser. The interface layer also makes it easy to scale horizontally (3TB-3) to increase performance as adding more application logic layers can help distribute the computations it needs.

In conclusion, three-tier architecture offers a very powerful platform to build distributed, multi system applications, which is far from what cuPID is meant to be. Using this style will deviate from the design goals the development team is focusing on. Therefore, the decision to not design the cuPID system with the 3-tier architecture was unanimous.

Attempted Subsystem Decomposition: System Architecture: ModelViewController (SA-02)

The Model View Controller (commonly called MVC) architecture is a type of system design architectural pattern that comprises of three subsystems: the Model subsystem, the View subsystem and the Controller subsystem. The Model subsystem is responsible for handling all the application logic, including the persistent storage of entity objects. The View subsystem is responsible for providing a set of interfaces to the user for interaction with the system. Finally the Controller subsystem is responsible for integrating the logic of the Model subsystem with the available interface provided by the View subsystem. Hence, the Controller serves as the bridge between the View subsystem and the Model subsystem.

Attempts were made to adapt the cuPID system to adhere to the MVC architecture, and an illustration of this adaptation is represented in Figure (2) of the *Appendix* section. Figure (2) of the *Appendix* section shows the dependencies between the subsystems that comprise the cuPID system while adhering to the MVC architectural pattern.

Based on the adaptation represented in the referenced Figure (2) of the *Appendix* section, some advantages and disadvantages related to the use of this architectural pattern surfaced. These benefits and flaws provided by the MVC architecture are outlined below in order to provide a solid justification as to why the MVC proved to be non ideal for the decomposition of the cuPID system.

The decomposition of the cuPID system based on the MVC architectural pattern provided the following benefits:

MVC-B-1. The architecture supports a modular representation of the system, such that related classes could be grouped together and could also be independently developed from one another.

MVC-B-2. The architecture also provided conformance to future extensibility of the cuPID system, because new functionality could be added by integrating a View, a Controller, and a Model logic for the said functionality without the expense of altering previous functionality of the system.

MVC-B-3. The MVC architecture is ideal for a dynamic data model. Hence, it offers an ideal interface (using the observer pattern) to update the views of the subsystems to be in sync with recent changes in the Model.

MVC-B-4. The architecture also provided a platform independent model such that regardless of the views used, the model was guaranteed to always operate in a stable manner.

Although the MVC architecture provided the team with substantial benefits, some flaws were noticed. These flaws include:

MVC-LM-1. The MVC architecture didn't not promote high cohesion between the classes in a subsystem. This is due to the fact that the logic required for the implementation of a given functionality had to be broken down into the Model, View, and the Controller subsystems respectively. Based on this, classes that worked together to provide a particular functionality had to be split up into a Model component, View component, and Controller component in order to conform with the MVC architecture.

MVC-LM-2. The MVC architecture is structured for dynamic models in order to maintain sanity of large systems. Due to this, the MVC architecture proves to provide more overhead than is necessary for the implementation of smaller systems.

MVC-LM-3. The MVC architecture promotes very high coupling from the Views and Controllers to the Model subsystem. This tends to be troublesome when interface changes need to be made to the Model subsystem.

First, it will be ideal to point out that it will not be wrong to execute the design of the cuPID system in conformance to the MVC architecture, but strict adherence to the design goals of the cuPID system stipulated in the *Design Goals Section* had to be obeyed. Based on these design goals, the MVC architecture did not prove to be the best architectural pattern. Justifications for this decision are stated in subsequent paragraphs.

In conformance to the *Flexibility Design Goal (DG-2)*, the MVC architecture was going to prove to be troublesome in terms of integrating a new subsystem functionality independent of the other functionalities currently provided by the system. This is due to the fact that the integration of new functionality had to be executed in the three respective subsystems of the architecture. Hence this did not promote modularity of the subsystems in terms of functionality integration, which in turn digressed from the flexibility design goal of the system.

Also, the main benefit of the MVC architecture which is highlighted in *MVCB-3,* did not prove essential for the cuPID system as the system is restricted to have only one user interacting with the system at any given point in time (NFR-21).

Due to the aforementioned reasons, the decision was made to not adhere to the MVC architectural pattern in the designing of the cuPID system.

Final Subsystem Decomposition: System Architecture: Repository (SA-03)

The repository architectural pattern provides a central means for subsystems to modify and access a single data structure. This structure is usually called the repository which in turn translates to a subsystem of it's own. In most cases, the Repository subsystem provides a fixed set of interfaces for other subsystems to utilize in order to access or modify the data of the application.

Attempts were made to adapt the cuPID system to adhere to the Repository architecture, and an illustration of this adaptation is represented in Figure (13). Figure (13) shows the dependencies between the subsystems that comprise the cuPID system while adhering to the Repository Architectural pattern. In the aim of adhering to the repository architectural pattern, it was logical to split the cuPID system into 6 major subsystems. These subsystems are enlisted as follows:

- ProjectManagement (SS-1)
- ProfileManagement (SS-2)
- AlgorithmExecution (SS-3)
- UserAuthentication (SS-4)
- DataAccessLayer (SS-5)
- Repository (SS-6)

In order to create a proper conceptual model of the mode of decomposition of the cuPID system using the Repository architecture, Table (1) provides the reader with the comprising subclasses of the cuPID system along with the classes they contain. Traceability is provided where applicable in order to maintain easy references to provided information. It should be noted that the classes provided in Table 1 remain unchanged.

Table 1 - cuPID Subsystem Decomposition and Comprising Classes

Subsystem ID	Subsystem Name	Comprising Classes
SS-01	ProjectManagement	*AbstractProjectManagerFactory, ProjectListManagerFactory, ProjectDetailsManagerFactory, ProjectDetailsManagerFactory, AbstractProjectController, ProjectListController, ProjectDetailsController, CreateProjectController, AbstractProjectView, ProjectListView, CreateProjectView, ProjectDetailsView*
SS-02	ProfileManagement	*PPPController, PPPWidget*
SS-03	AlgorithmExecution	*Team, MatchReport, InsomniaMatchingAlgorithm, CreatedTeamsResultView, CreateTeamsController*
SS-04	UserAuthentication	*AuthenticationController, SignUpForm, LogInForm*
SS-05	DataAccessLayer	*Project, PPP, PPPReal, PPPProxy, Qualification, Configuration, User, StudentUser, AdministratorUser, IMappable, DataAccessFacade*
SS-06	Repository	*DataAccessDispatcher*, *Repository , UserRepository, ProfileRepository, ProjectRepository, DatabaseManager*

The *ProjectManagement* subsystem is responsible for all logic related to the interaction that occurs with a project such as Creation(UC-09), Editing(UC-10), Registration(UC-12) and Unregistration(UC-11). Figure (7) provides the reader with the UML class diagram of the *ProjectManagement* subsystem, along with it's comprising classes.

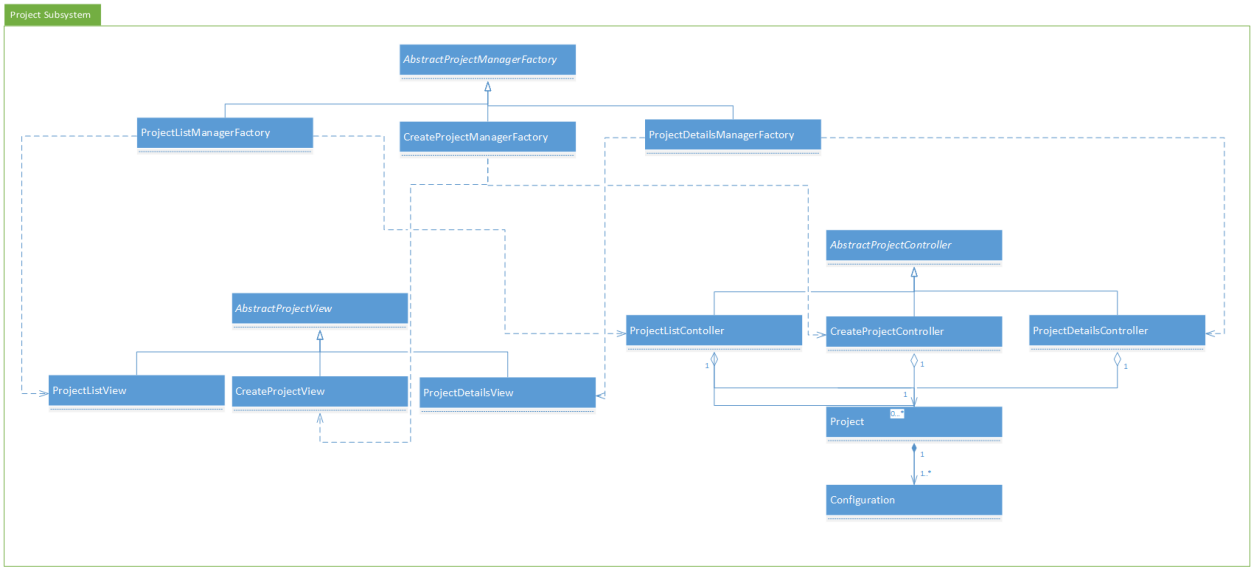


Figure 7 - UML class diagram for the ProjectManagement Subsystem

The *ProfileMangemant* subsystem is responsible for all logic related to the interaction that occurs with a profile such as Creation(UC-05) and Editing(UC-04). Figure (8) provides the reader with the UML class diagram of the *ProfileMangement* subsystem, along with it's comprising classes.

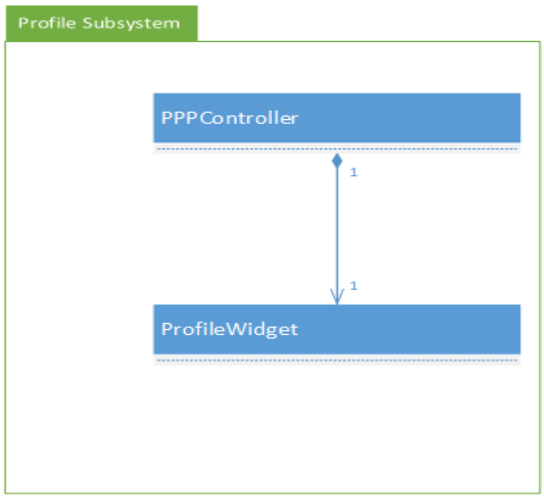


Figure 8 - UML class diagram for the ProfileMangement Subsystem

The *AlgorithmExecution* subsystem is responsible for all logic related to the creation of teams for a particular project(UC-08). Figure (9) provides the reader with the UML class diagram of the *AlgorithmExecution* subsystem, along with it's comprising classes.

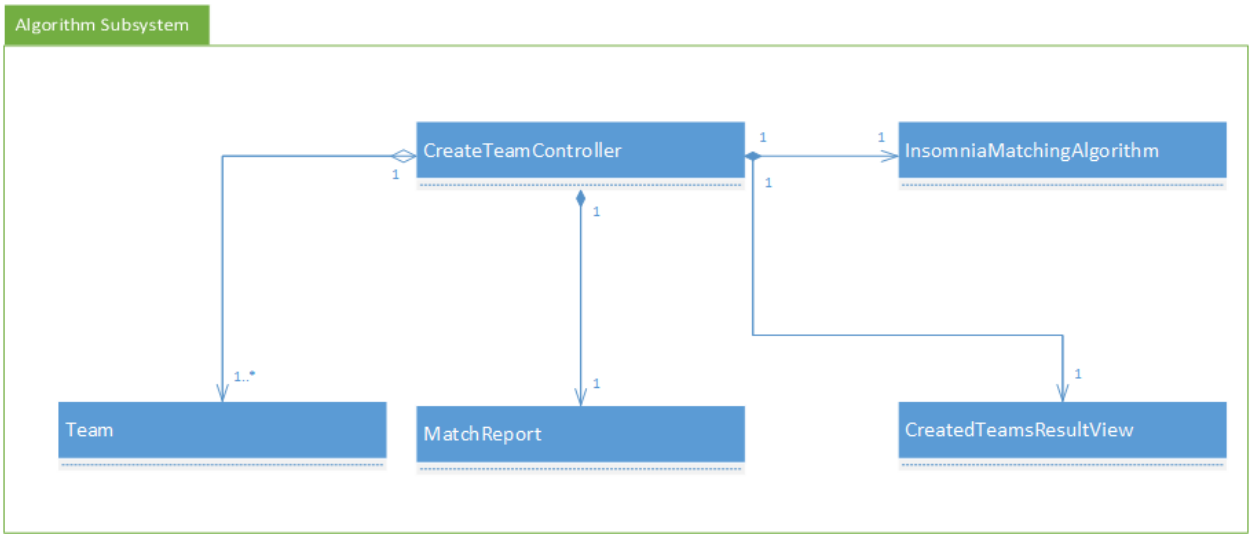


Figure 9 - UML class diagram for the AlgorithmExecution Subsystem

The *UserAuthentication* subsystem handles the logic related to the interaction that occurs while trying to authenticate a particular user into the cuPID system. Figure (10) provides the reader with the UML class diagram of the *UserAuthentication* subsystem, along with it's comprising classes.

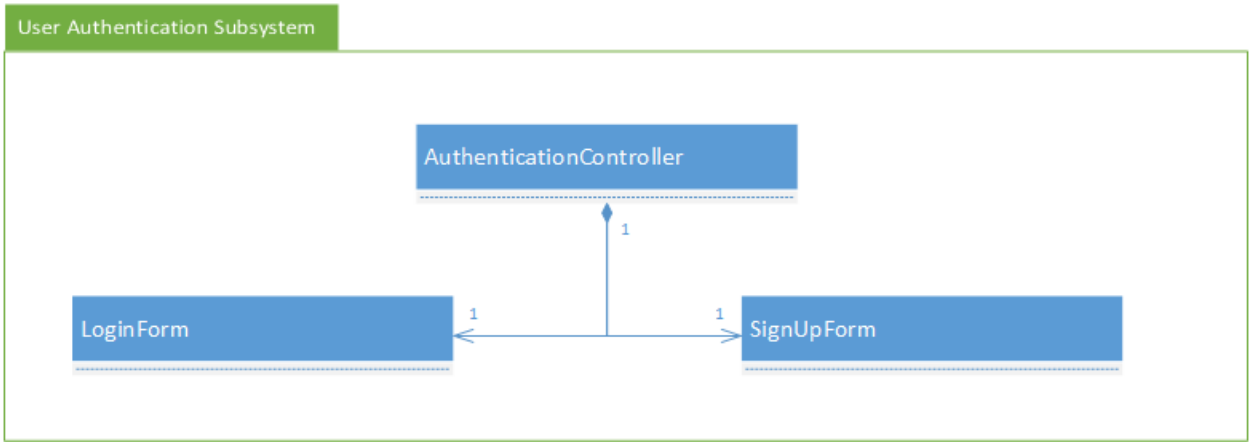


Figure 10 - UML class diagram for the UserAuthentication Subsystem

The *DataAccessLayer* subsystem serves as an extra layer of abstraction to handle all requests to the Repository subsystem made by other subsystems. The *DataAccessLayer* subsystem also provides an abstraction to the responses received by the *Repository* subsystem in response to requests made by other subsystems. This subsystem was added to reduce the coupling the *Repository* subsystem. Figure (11) provides the reader with the UML class diagram of the *DataAccessLayer* subsystem, along with it's comprising classes.

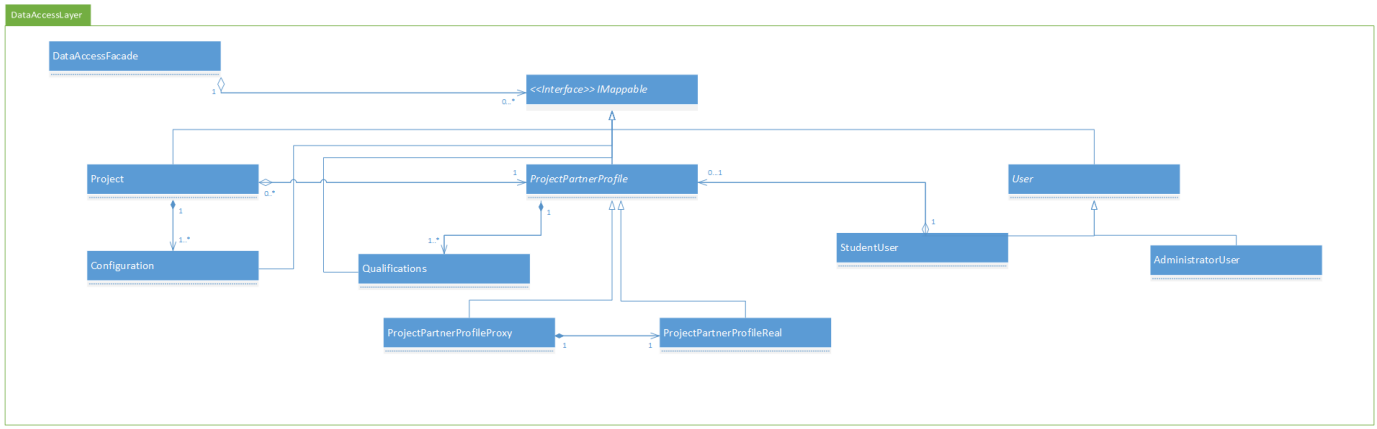


Figure 11 - UML class diagram for the DataAccessLayer Subsystem

Finally, the *Repository* subsystem provides a set of interfaces to the *DataAccessLayer* subsystem for the storage and retrieval of persistent data. Figure (12) provides the reader with the UML class diagram of the *Repository* subsystem, along with it's comprising classes.

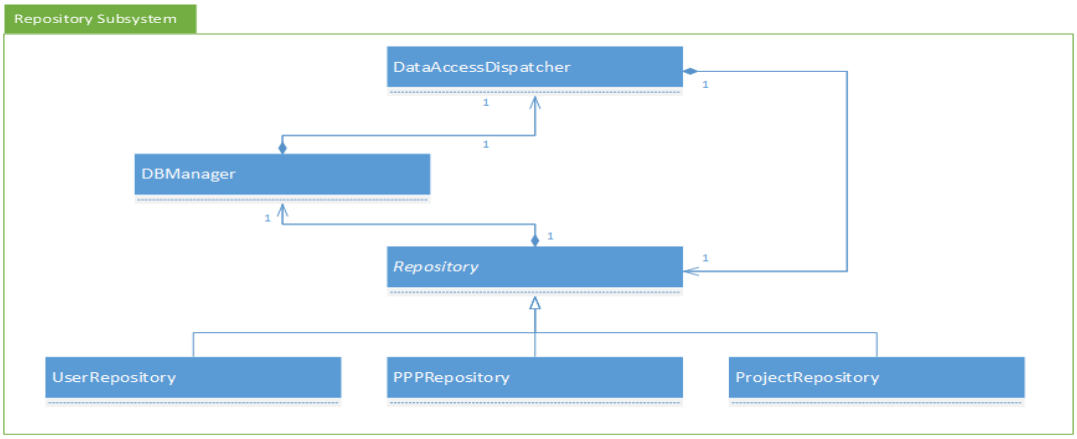


Figure 12 - UML class diagram for the Repository Subsystem

An illustrated amalgamation of all the stated subsystems above is represented below in the same format as a UML class Diagram. Packages are used to represent subsystems for ease of identification in the illustration provided. Figure(13) provides the reader with the stated illustration

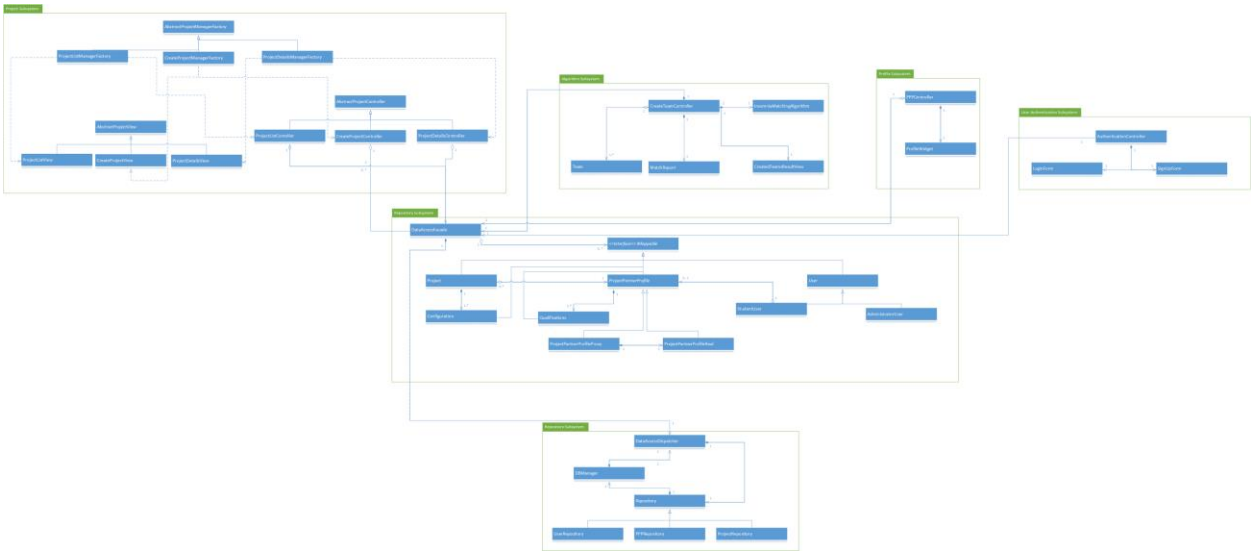


Figure 13 - UML class diagram of the cuPID system showing Subsystems

The adaptation of the cuPID system to the Repository architectural pattern provided a lot of benefits, and reasonable limitations which were handled by the constraints of the functionality of the cuPID system.

The advantages offered by adapting to the Repository architectural pattern are as follows:

R-B-1. The architecture supports a modular representation of the system, such that related classes could be grouped together and could also be independently developed from one another.

R-B-2. The architecture promotes high cohesion among classes in a particular subsystem. This is based on the fact that all subsystems are established by grouping all classes that work together to provide a given functionality

R-B-3. Using the repository architecture, new functionality could be easily added to the cuPID system by creating a new subsystem with set of classes that work together to provide the said functionality. Hence, this promotes easy extensibility

R-B-4. This architecture also promotes uniformity by delegating the task of maintaining proper state of entities to the *Repository* Subsystem. Hence, other subsystems do not keep track of state of entity objects, and this helps prevent duplicated and non-uniform data.

Based on the fact that no architectural pattern is perfect, the limitations of the Repository architecture in relation to the cuPID system are provided below:

R-LM-1. This architecture establishes a high coupling between Repository and other subsystems.

R-LM-2. The central Repository structure becomes a bottleneck based on the fact that all subsystems might be trying to access the Repository subsystem at the same time.

In the case of the cuPID system, the limitations of the Repository architecture presented above are resolved based on the following:

- The high coupling between the *Repository* and other subsystems was removed by creating a *DataAccessLayer* subsystem. This subsystems helps abstract the interfaces for the requests and responses to and from *Repository* subsystem. Due to this, the subsystems are now independent of the Repository subsystem. This enables the *Repository* subsystem to be flexible to change independent of the other subsystems.
- In the case of the bottleneck that might occur due to multiple subsystems trying to access the *Repository* subsystem at the same time, this is resolved by the statement of the fact that the cuPID system is constrained to have only one user operating the system at any given point in time (NFR-21). Hence, a scenario will never occur where multiple subsystems will need to access data at the same time, since a user can utilize only one subsystem at any point in time.

Based on the fact that all stated Design Goals were satisfied by the Repository architecture, and the limitations accompanying the adaptation of the Repository architecture could be properly resolved by the design and constraints of the cuPID system, the Repository architecture proved to be the ideal way of decomposing the cuPID system.

In order to illustrate the dependencies between the subsystems, we provide the reader the the UML component diagram of the the final subsystem decomposition. As stated, this diagram shows the dependencies and relationships between subsystems of the cuPID system. This in turn helps identify interesting design criteria such as cohesion and coupling between the subsystems of the cuPID system. Figure 14 provides the reader with the described representation of the dependencies between the subsystems mentioned.

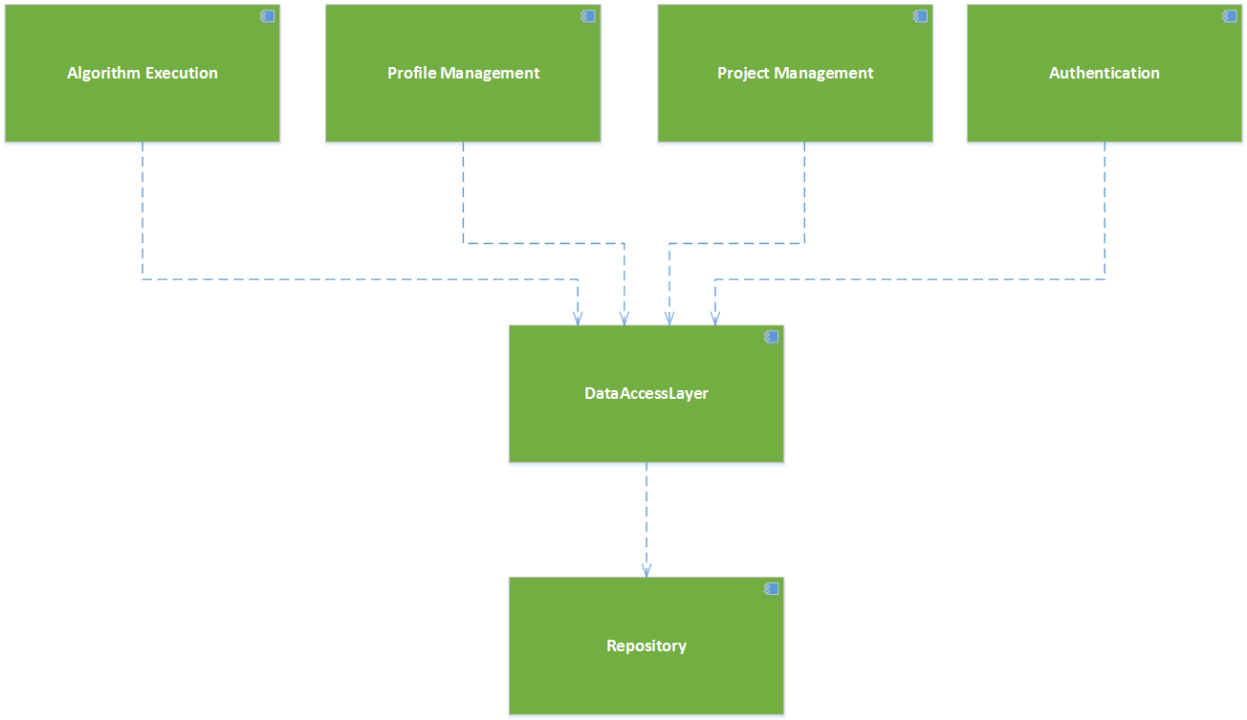


Figure 14 - UML component diagram of the cuPID system showing Subsystem dependencies

2.3 Design Evolution

First and foremost, it will be of utmost importance to state the fact that no system decomposition can depict a perfect overview of how subsystems should interact in order to provide the best possible system as a whole, this section is dedicated to the comparison and criticism of the work presented above in the *Section 2.2.* The aim of this is to highlight the discrepancies between the initial design of the prototype system and the final proposed design. Where applicable, justifications are provided for the reasons for such digressions.

In order to provide ease of readability, the evolution of the design of the cuPID system will be executed in a systematic manner by outlining the major points in the evolution and giving proper explanations and justifications for each of them as deemed necessary. Hence, the major differences and design choices between the design of the prototype and the final cuPID system are enlisted as follows:

- Dependence decoupling between stateful and non-stateful components of the cuPID system
- Universal Repository subsystem design And Implicit Auto-Saving
- Embedded caching in *DataAccessLayer* to abstract knowledge of caching functionality from other subsystems
- Lazy Loading of Computationally Intensive Persistent Objects from Database using Proxy Design Pattern
- Design Patterns inclusion for increased cohesion in subsystems
- Algorithm decoupling from *ProjectSubsystem*

Explanations and justifications will be given for the topics listed above respectively

Dependence decoupling between stateful and stateless components of the cuPID system

From the design of the cuPID prototype subsystem, it was noticed that classes that maintained state (Entity Objects) were scattered in different subsystems. This proved to be a pain when a general interface for saving these objects needed to be provided for the cuPID system. The initial thought was based on the fact that entity objects related with the functionality of a particular subsystem were to be placed within the said subsystem. This was a wrong choice considering the fact that some of these entity objects comprised of other entity objects which were contained in other subsystems. This approach was destined to increase the coupling between unrelated subsystems that were not meant to depend on each other in accordance with the chosen architecture (*Repository*).

After much thought, it was considered to be of great benefit to move all entity objects that maintained state into one subsystem, and then consequently provide public interfaces to access these objects as required by other subsystems. This was done by moving all the Entity Objects that maintained state into one subsystem called the *DataAccessLayer* subsystem. This subsystem contains a facade class that provided public interfaces to the the interfaces provided by each of the Entity objects. By doing this, subsystems were able to be decoupled further thereby providing one common interface for other subsystems to interact with related entity objects. This in turn helped encourage easy subsystem extensibility which is in accordance with our Flexibility design goal (DG-2)

Universal Repository Subsystem Design and Implicit Auto-saving

Although the prototype cuPID system was modelled to conform to the repository architectural style, it was far from a proper enactment of the specifications of the style. A major flaw in the conformance of this architectural style in the prototype was the fact that the Repository subsystem was also dependent on the Entity objects that it was saving. This caused a bidirectional coupling between the other subsystems and the repository subsystem.

As a major design choice, it was decided that the Repository subsystem was to maintain the most minimal knowledge of the Entity objects that it was saving in persistent storage. After thorough brainstorming, it was decided that an *IMappable* interface (EO-14) was to be defined for all entity objects that maintained state and were to be saved in persistent storage by the Repository subsystem. The *IMappable* interface defined an interface for Entity objects to represent themselves as a key-value mapping of their attributes. This helps to turn the bidirectional coupling between other subsystems (in this case, the *DataAccessLayer* subsystem) and the repository subsystem. The new design is such that each **Savable** Entity object implements the *IMappable* interface by providing an interface for representing themselves as a key-value mapping of their attributes. Based on this, the repository receives a mapping of key-value pairs and a particular action, and it (the Repository) uses this information to save the representative entity. It will be beneficial to point out that the same process applies to retrieving information from the *Repository* subsystem by the *DataAccessLayer* subsystem. In conformance with the *IMappable* interface, the entity objects also provide an interface for reading a given set of key-value mappings and providing a representative object based on the information received. This was done in accordance to the Reusability design goal (DG-5)

Embedded Caching in *DataAccessLayer* to Abstract Knowledge of Caching Functionality from Other Subsystems

In the decomposition of the cuPID prototype system, it was noticed that the knowledge of the cuPIDSession Object (contained in the Repository subsystem) caused unnecessary coupling, between the other subsystems and the Repository subsystem.

In the final design of the cuPID system, this problem was solved by entirely getting rid of the cuPIDSesssion class and completely embedding the caching functionality in the modus operandi of the DataAccessLayer (DAL) subsystem. This means that as the DAL subsystem goes about its' operations of retrieving and saving modified entities, it keeps a private cache of recently retrieved entities needed by other subsystems. By doing this, the performance of the cuPID system is guaranteed to be optimized because the DAL does not constantly make repetitive queries to the *Repository* subsystem in order to retrieve previously retrieved data. Also, this helps encapsulate the knowledge of a cache from other subsystems, thereby reducing coupling. It should be pointed out that this design choice was made in order to comply to the stated Performance design goal (DG-03)

Lazy Loading of Computationally Intensive Persistent Objects from Database using Proxy Design Pattern

One Major flaw of the cuPID prototype was that a considerable amount of time was spent loading large entity objects like (PPP entities). This was not a smart way to go about this based on the fact that some of these information that were loaded were not being used at the moment.

The design team decided that it was of utmost importance to provide an optimal performance in terms of speed as much as possible (DG-03). In order to conform to this, the choice of using the *Proxy* design pattern to imitate the imitate the PPP entity as much as possible until it was deemed necessary to fully load the PPP was applied to the design of the final cuPID system. Using the *Proxy* design pattern with the PPP entity object, we provide subsystems with as minimal information as possible needed to complete their respective tasks until it is deemed necessary to fully load the full PPP entity. This also increases the level of cohesion between the classes in the *DAL* subsystem. For the sake of uniformity across discussed topics, it will be proper to state that this design choice was made in conformance to the stated Performance design goal (DG-03)

Design Patterns inclusion for increased cohesion in subsystems

Based on the fact that the cuPID prototype system was a rushed delivery without much thought on proper design, there was a considerable lack of design patterns which in turn help encourage high cohesion between classes in the subsystem of which the design pattern is applied. This lack of proper design patterns was another cause of unneeded coupling between subsystems and very low cohesion between the classes in a subsystem.

The final cuPID prototype system solves this by providing a rich use of proper design patterns where applicable. This was done in order to provide a systematic approach to implementing particular functionality. The chosen design patterns of the final cuPID system design are: *Abstract Factory (DP-01), Facade (DP-02), Proxy (DP-03),* and *Singleton (DP-04).* These stated design patterns in relation to the cuPID system are further explained in *Section 3.3* (Design Patterns section) of this document. It should be pointed out that design patterns where only implemented where necessary and this was done in conformance to the Reusability design goal (DG-05)

Algorithm Decoupling From ProjectSubsystem

The delivered prototype of the cuPID system didn't include the feature of team creation, but based on the design of the prototype system, it was inevitable that the *AlgorithmExecution* Subsystem was going to be highly coupled with the project subsystem and vice versa. This fact was put in mind during the design of the final cuPID system decomposition.

In the final cuPID system decomposition, we present an *AlgorithmExecution* subsystem that is completely independent of the *ProjectManagement* subsystem. This enables extra functionality of being able to launch team creation from any part of the application on a visible project. Besides the extra functionality that this mode of separation provides, this design also helps to remove the coupling between the *AlgorithmExecution* subsystem and the *ProjectManagement* subsystem. With this new design, the *AlgorithmExecution* subsystem talks directly with the *DAL* subsystem in order to retrieve required Project details. This design design was made in order to conform to the Reusability Design Goal (DG-01)

No design evolution will be complete without highlighting the limitations and critique of the current design of the final cuPID system. Based on the fact that all design choices were made in order to conform to the stated design goals of the insomnia team, it will be beneficial to highlight that there was a bias in the choice of design patterns and architectural styles. Hence, it goes without saying that the current design of the final cuPID system satisfies the design goals of the team at the expense of other design goals that might be deemed necessary in other projects. We will like to point out that decisions were slightly myopic in the fact that further scalability of the cuPID system is not foreseen, and hence, the Repository subsystem lacks the proper mechanism of resolving deadlocks which might occur in a scaled system. Although this is not the case for cuPID based on stated constraints (NFR-15, NFR-21), it will definitely be significant to factor such functionality in future versions of the system in order to resolve such situations.

Section 3: Design Strategies

This section of the document is set to outline the design strategies reflected in the cuPID system. These design strategies are broken up into three major sections: Hardware and Software Mapping (*Section 3.1)*, Persistent Data Management (*Section 3.2*), and Design Patterns (*Section 3.3*). The aim of this whole section is to present justifications for all questionable design decisions made in the System Decomposition section of the document (Section 2)

3.1 Hardware and Software Mapping

This subsection is broken down into two parts to separate the mappings between hardware hardware platform configurations and relationships between run-time components. Each of these topics will establish a means for communication between components and nodes of the system.

These relational mappings will be shown as nodes and components of the system. To define these definitions briefly, a component is a self-contained entity, composed of a subsystem or a group of subsystems, that provides services to other components or external actors. A node, however, is a physical device or an execution environment in which components are executed within and contain other nodes within its own environment. These system hardware and software mappings are created to follow the implementation section of the non functional requirements defined in the requirements analysis phase. The UML deployment diagram depicted below shows the the hardware mapping for the cuPID system followed by a detailed description regarding the diagram.

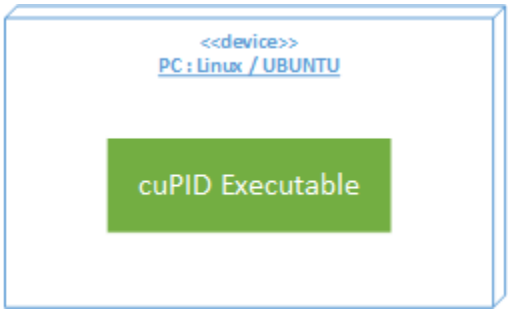


Figure 15 - UML Deployment diagram showing hardware to software mapping of the cuPID system

The cuPID system, presented by Team Insomnia, is a single host application used in project team generation. Based on the definition of a node, the deployment diagram above represents the cuPID system itself and which is encapsulated inside the physical runtime environment, the host. In accordance to *NFR-15*, the system must run on one host loaded with the Ubuntu 14.04 LTS operating system and communicates with no other nodes over a network or external devices. With that said, the physical platform of the cuPID system is installed on one per operating system basis, leading to just one node representing the entire system. This draws a limitation to the system's data storage solution when installing copies across each operational node as data between hosts cannot be shared. The development team noticed this side effect

and in order to follow through with the set design goals of the system (DG-5, DG-2), designed the repository subsystem (SS-6) to be the only subsystem that contains state information about the application. In future considerations, this subsystem could be built into a component and moved into a node on its own and provide data to all cuPID systems over a network connection (NFR-14).

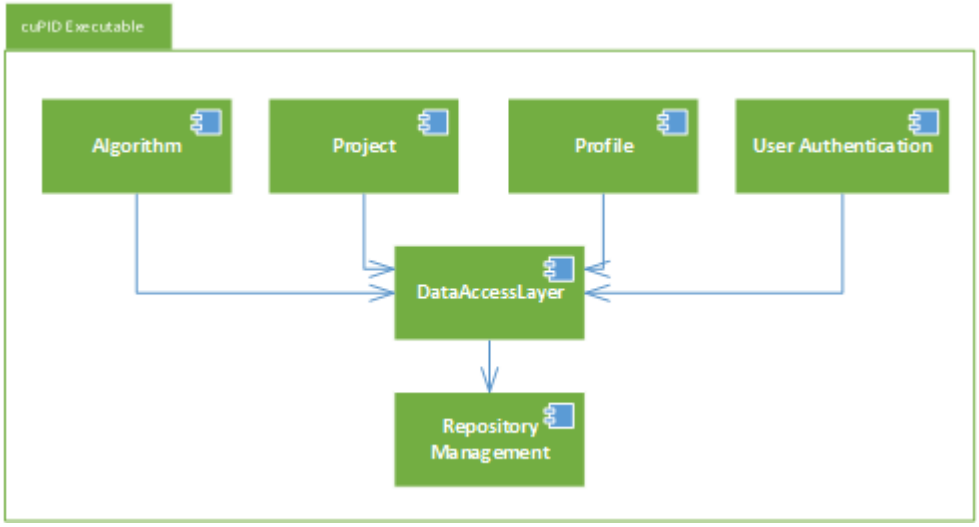


Figure 16 - UML Component Diagram for the Repository Architecture Pattern

Within the cuPID deployment diagram, the component inside was the cuPID executable that presented the running process that is the system. The diagram above expands that component into a UML Component diagram, shown in *Figure 16* and depicts the interfaces in which each component depends on. This diagram specifies the relationships between groupings of subsystem components within the only existing node of the UML deployment diagram, referencing *Figure 15*. The cuPID strategy to map components within its respective node is by using the repository architectural style due to a superior fit with the Insomnia Team's non functional requirements and design goals as stated in *Section 2.2*. The relationships shown by arrows in the deployment diagram depict the dependence of each components with the directionality of the arrows showing the components it depends on.

Based on the repository architecture, the selected components resemble the relationship between associated components and with the repository component acting as the backbone of the system. Care was taken to separate stateful from stateless components such that the data storage is far apart from the presentation interface. This architectural pattern highly complements the data driven aspects of cuPID in which functionality can be designed and implemented in modular subsystems. Each functionality can be assigned to one developer to develop and implementation should be independent of other subsystems (DG-2). Each component however, is highly coupled to one and only other component, the *DataAccessLayer* (SS-5) referred to as the DAL. In this design, each core functional subsystem with a dependency on the DAL, can retrieve any information it needs from the DAL without a dependence of other components. We will refer to these components that depend on the DAL as "top level components". These top level components were mapped on a 1 to 1 basis in

accordance to the subsystem due to their interactions with the DAL through shared interfaces where information is only sent to and retrieved from DAL.

The DAL is designed to only dependent on the repository's component interface. The relationship between these two components is simply that the repository provides the DAL with read and write privileges to the persistent storage mechanism. With the separation between the two components, an error in the repository that defunctionalizes the component will not break the functionalities of the upper level components due to the caching layer implemented in the DAL. The repository component is designed to be independent of all other components, and has absolutely no notion what top level component of the cuPID system had made the request to it. In addition, the repository can only understand a specific request structure in order to fulfil the read or write request, and the only other component that knows about this structure, is the DAL. Hence, when any arbitrary system wants to use the service interfaces that the repository provides, the system only has to pass in the request in the correct structure. At this stage, the repository component only understands how to fulfil these request, fulfilling the stateful nature of this subsystem. Following the system design goal (DG-2), the repository becomes a highly cohesive component and its easy to extend the range of its actions to classes specific to the database schema within the component.

In conclusion, the hardware and software mappings of cuPID results in one node due to the single host nature of the system. The repository architecture was chosen to represent the component dependencies for this data driven system and provided cuPID with a flexible modular design that separated stateful information storage with stateless, modular subsystems handling the application logic.

3.2 Persistent Data Management

The developers of cuPID agreed on implementing a relational database design over flat files for cuPID's persistent data solution. This choice promotes the elimination of data redundancy, open the capability to form complex queries on larger datasets, and provides a greater level of security over flat files. In this subsection, the entity relationship diagram will be discussed in detail to address the reasoning to the system's storage solution. Such details will answer questions to what data should be persisted, and what format, or schema, is the data saved under, how the data is accessed, and where the data is stored. The primary concern over the persistent data management section is performance since this aspect of the system will present to manifest into a bottleneck so care has been taken to design a fast and sensible database schema under SQLite (NFR-09). Over the course of this subsection, the aforementioned inquisitions will be answered one by one.

What data should be persisted?

The cuPID system is driven by user sponsored data therefore all inputs must be saved into the database. By referring back to the functional requirements, of the *Requirement Analysis* document, this section can draw out all information that is required to be persisted. The table below will re-evaluate these requirements with reference of the comprising classes of the cuPID system. Table (2) below presents the reader with the list of savable entities in the cuPID system.

Table 2 - Table of savable entities in relation to the functional requirements

Functional Requirement	**Saveable Entities**
FR-01 - Project Partner Profile	*ProjectPartnerProfile* (EO-08), *Qualifications* (EO-11) and their associations with one StudentUser's Profile
FR-02 - StudentUser Interacts with Projects	Student registration status with project
FR-03 - AdministratorUser Interacts with Projects	*Project* (EO-04), *Configuration* (EO-05) and it's association to one particular project
Others	User accounts (EO-01), cuPID general settings

The objects required to be saved incorporates all user generated material which will be loaded during runtime when requested.

What format should it be persisted under?

Care has been taken to select relational database persistence to model the data cuPID system represent with its UML diagrams and its associations between classes. In reference to the final system architecture UML (SA-03), each association between the entity objects depicted in *Table 2* above, can be mapped on to the database schema solution. In order to convert the UML to a database relational diagram, below is a direct mapping of relationships between the database tables used to represent cuPID system entity relationships in the form of Entity Relational Diagram (ERD).

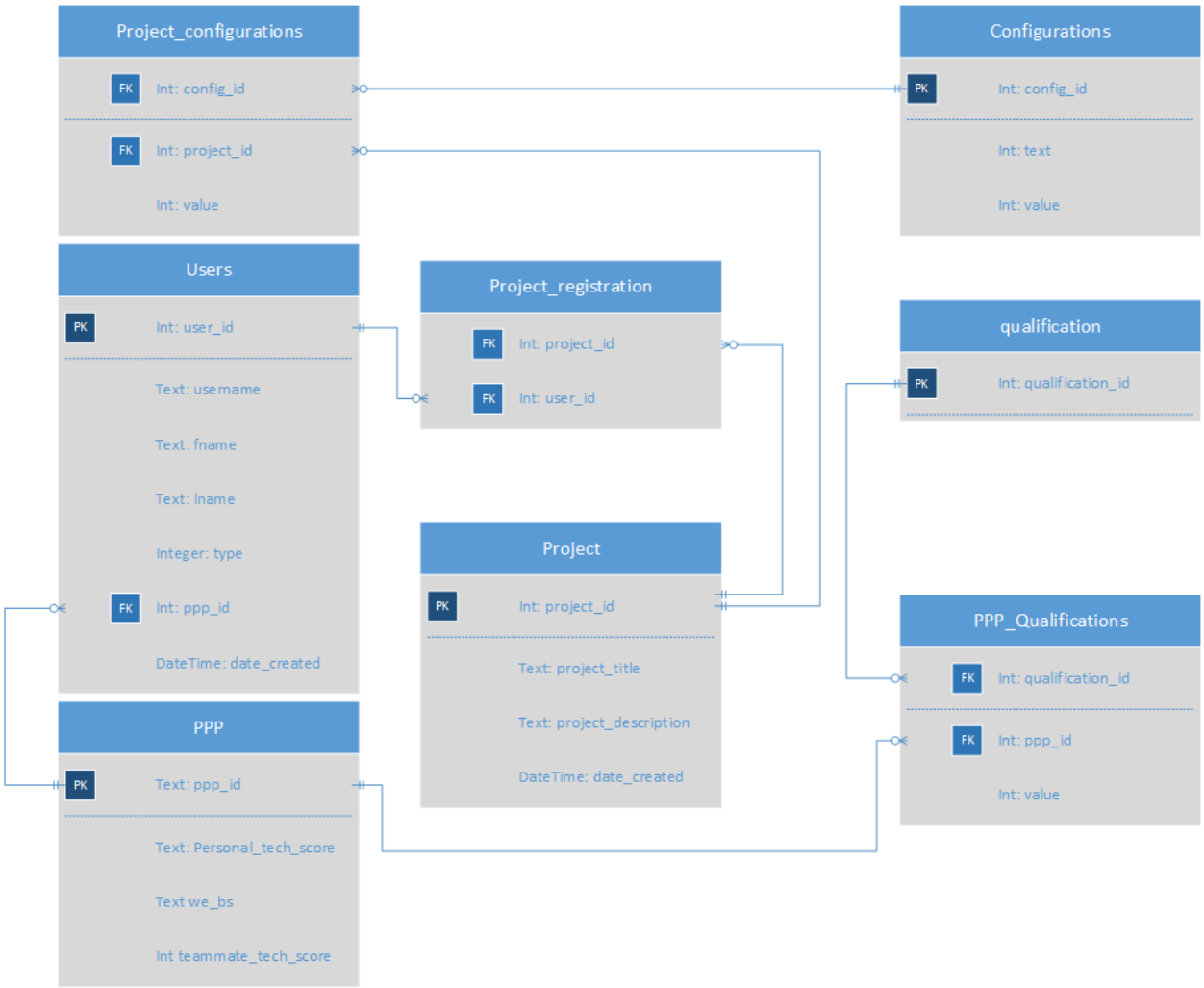


Figure 17 - Entity Relational Diagram for the cuPID backend data system

As seen in the entity relational diagram, each table represents a physical entity object mapping to its respective persistence storage location. A careful assessment to this database schema proved to the development team that no data is repeated more than once due to a primary id representing the entire row of data. This row of data is referenced as a foreign key on other database tables which helps maintain integrity of the data. Many to many relationships

has been addressed through the use of weak entities, tables without their own primary key and the candidate keys are a composition of foreign key referenced from other table. A detailed decomposition of each table is as follows:

Users (DB-1)

user_id - An integer primary key used to identify unique users with. This key is the primary multiplexing key to save and update existing users who are registered on the cuPID system. This primary key automatically increments itself with consecutive user inserts.
username - A mandatory text field for any length string that is guaranteed to be unique which represents the user's login name.
fname - The first name of a user that cuPID uses to welcome. This name is used by the algorithm to show the match report after teams are made by the algorithm.
lname - The last name of the user used in the same way was the first name.
type - An integer to associate the type of the user which indicating whether this user is an administrator user (number 1) or student user (number 2).
ppp_id - A nullable foreign key field used only for student users to associate a project partner identifier profile to the user. A null field shows that this user does not have a PPP (EO-08).
date_created - An automatic timestamp used to record the date and time when this user registered on the cuPID system.

PPP (ProjectPartnerProfile) (DB-2)

ppp_id - An integer primary key used to identify a project partner profile for the student user. This field is auto increment as consecutive profiles are created. When a student user creates a PPP, this identifier will be associated to their user id of the users table.
personal_tech_score - This integer value represents the aggregate personal technical score of the student user. This score is generated during saving of the PPP.
teammate_tech_score - This integer value represents the aggregate teammate technical score of the student user. This score is generated during saving of the PPP.
we_bs - This integer value is constrained from 31 to 248 and represents the possible mappings for team ethics and work habits. The integer is converted to binary and each index of the binary bit string corresponds to a work ethic/habit qualification.

Qualification (DB-3)

qualification_id - An integer primary key where each integer represents a enumerable qualification. This key automatically increments itself one adding new qualifications.

Project (DB-4)

project_id - An integer primary key to represent a project created by an administrator user. This project id is auto incremented so two projects may never have the same id. Project id is significant because it is used to show users who registered with a project and also configurations that are set for this project.
project_title - A text field saved by the administrator user to depict the titles of the project. Project titles must exist when saving or else an error is flagged.
project_description - A text field saved by the administrator use to depict the project descriptions that are shown to users. This field must be entered or else an error will occur.

date_created - An automatic timestamp used to record the date and time when project is created on the cuPID system.

Configurations (DB-5)

config_id - An integer primary key used to represent a configuration to be set for a project in the cuPID system.
text - The text field description of the configuration in human readable form.

PPP_Qualifications (DB-6)

qualification_id - An integer foreign key referencing the qualification type defined by the qualifications table
ppp_id - An integer foreign key referencing the project partner profile that particular this qualification belongs to.
value - This integer field represent the setting value that the qualification was set for.

Project_Configurations (DB-7)

config_id - An integer foreign key referencing the configuration id from the configuration table
project_id - An integer foreign key referencing the project that the administrator user created
value - The value for the set configuration that is associated to a certain project

Project_Registration (DB-8)

project_id - An integer foreign key referencing the project that the administrator created
user_id - An integer foreign key refering a user that is associated to this project. When this id references an administrator, the relationship shows that administrator has created the project. When this user id references a student user, the relationship shows that a student has registered into this project

Associations to Note

- Users registration to projects

This association is shown through the project registration (*DB-8*) and in the form of a one to many table where a project can have many students. We created this weak entity table to reduce redundancy for projects and users. Users (*DB-1*) and Projects (*DB-4*) have their own respective rows and only the primary keys are referenced from those core entity tables. Designing our user registration to projects association in this way results in the effectiveness of the project registration table to express an atomic association and allows easy joining between users (*DB-1*) and projects (*DB-4*) table during particular events of the cuPID system that needs to find either the project or users registered in a project.

- Administrator's own projects

This database schema designed for the project registration (*DB-8*) granted cuPID with the convenience of remembering which administrators created which projects. In the cuPID requirement analysis, it was determined that administrator users can only view their list of created projects (*FR-03-01*) and thus, the association is included in this table. Accessing and

administrator's own projects is implemented in the same way as a student user getting a list of projects that they are registered for.

- Project configurations

Each project has a set of configurations that are set for the algorithm (*EO-07*) to use. The cuPID data management design reduces the data redundancy between core entity objects and increases data reusability and extensibility by allocating configurations into their own table (*DB-5*). By composing a configuration lookup table, this approach grants cuPID the possibility to easily extend its set of project configuration without the need to change other elements of the database schema. The values for each configuration is stored in also a weak entity table, project configurations (*DB-7*) without crowding the space of the lookup take. By separating configuration and project configuration associative binding, the design cohesively groups together common entities that perform the associative bindings with values and enhances the performance of our lookup table.

- PPP qualifications

Project partner profile configurations work in similar ways as the previously described project configuration. The goal of designing the PPP to qualification table (*DB-6*) was to provide an extensible set of qualifications and use the qualification look up table (*DB-3*) to bind with a profile id. This enabled a clean way to update qualifications associated with a profile, without changing the profile table while maintaining the performance of cuPID's look up tables.

How is the data accessed?

The cuPID system's primary data access point is the Data Access Layer (*SS-5*) and all upper application logic subsystems depend on it as stated in our Repository Architecture and decomposition. In order to align the design of the cuPID persistent data management solution with the design goals for system flexibility/modularity (*DS-2*), this layer serves as the access point (facade pattern) to the repository subsystem (*SS-6*). By separating the interface with the implementation, our design has successfully established a fine separation between stateless application design and the stateful repository (storage) system. The repository system can survive independently of the application accessing it and will transfer state back to any requesting subsystems, mainly Data Access Layer (*SS-5*). The format used to facilitate data transfer is standardized to map key value pairs of application entity objects where the keys are object properties and values are the property values. The advantage of this approach is the total decoupling of the repository subsystem from the application logic that requires state.

Where is the data stored?

Within the repository subsystem (*SS-6*), data is accessed from a SQLite database file by using SQL queries. This database file is stored in the install directory of the cuPID system. When cuPID boots up for the first time, the contents of this database file is transferred into main memory to avoid disk seeks and increase the performance of the application when reading from storage. Occasionally, when the *Repository* subsystem (*SS-6*) determines an appropriate time to commit the SQL transactions, then SQLite will write the state of the storage back to disk.

3.3 Design Patterns

A design pattern can be defined as a general reusable solution to a commonly occurring problem within a given context of software design. Based on the fact that the cuPID system faced commonly occurring problems such as: Resource allocation, Uniform object creation, and performance optimization, it would have been considered foolish to create an implementation that made no use of already tried and tested design solutions to the stated problems. In the cuPID system, care was taken to ensure that all occurring problems were solved with a known design pattern in order to provide reliability and assurance of correctness of the implementation. This section outlines the design patterns used in the cuPID system, and a detailed description and justification of the reasons for selecting each design pattern is given.

Design Pattern: Abstract Factory (DP-01)

The Abstract factory design pattern is an object creational design pattern that helps provide an interface for creating families of related or dependent objects without the need of specifying the concrete classes needed. The justification of the design choice as to why the Abstract Factory design pattern was used with respect to the cuPID system is provided as answers to the following questions below:

Where is it used?

In the cuPID system, the Abstract Factory design pattern was used in the *ProjectManagement* subystem (SS-01). This was used in the grouping of project management functionality. The classes related in this pattern are enlisted as follows: AbstractProjectManagementFactory (EO-16), ProjectDetailManagementFactory (EO-18), CreateProjectManagementFactory (EO-19), ProjectListMangementFactory (EO-17), AbstractProjectView (EO-20), ProjectDetailsView (EO-22), CreateProjectView (EO-23), ProjectListView (EO-21), AbstractProjectController (EO-24), ProjectDetailsController (EO-26), CreateProjectController (EO-27), ProjectListController (EO-25).

Why is it used?

The motivation behind using the Abstract Factory design pattern in the cuPID system was due to the fact that the *ProjectManagement* subsystem provided different interfaces for particular interactions that were not related to one another. Due to this, classes that were related in providing a particular functionality had to be grouped together. The Abstract Factory provided a means to group these classes into families (Factories). Hence, if the user wishes to create a project, then the CreateProjectFactory will take care of the creation of all the necessary classes needed for the project creation functionality. The same goes for viewing the details of a particular project (ProjectDetailsFactory) and viewing a list of projects (ProjectListFactory). The decision to use this pattern was based on the fact that it was necessary to separate the creation process for the objects from how the objects are composed and represented. Hence, this makes the creation of all the objects needed to provide a particular functionality seamless

What does it achieve?

The adaptation of the Abstract Factory Design pattern provides a means of seamlessly transitioning between Project functionality as deemed necessary due to interactions caused by the user. This greatly encourages addition of extra functionality the ProjectManagement subsystem, because there will be no need to worry about how the classes needed to provide a particular functionality are created.

The use of the Abstract Factory Design pattern also provides a means of isolating concrete classes. Based on the fact that the pattern encapsulates the creation of the objects, users of the factory are isolated from the implementation of the creation of the classes. This helps to greatly reduce coupling between subsystems and promotes stability to problems that may arise due to implementation changes.

Finally, the adaptation of the Abstract Factory Design pattern helps promote consistency among objects currently created. Based on the fact that only one factory can be used at any point in time, only classes related to the currently in use factory are loaded, thereby ensuring consistency in the subsystem.

Design Pattern: Facade (DP-02)

The facade design pattern helps provide a unified interface to a set of classes within a subsystem. This patterns defines a higher-level interface that can wrap a complicated subsystem with a simpler interface. Better abstraction is created through the use of a facade and clients can leverage the complexities of the subsystem with ease, and promotes When providing this facade to external subsystems, the pattern decouples dependencies to complex classes by specifying itself as the access point. Using just a facade to interface with a subsystem may limit the features and flexibility that “power users” may need, however, it does not fully abstract classes making complex actions still accessible over the facade.

Where is it used?

The facade pattern is used on the DataAccessFacade (*EO-15*) within the DataAccessLayer (*SS-5*) and DataAccessDispatcher (*EO-29*) within the Repository subsystem (*SS-6*). The DataAccessFacade provides a simple interface for the subsystems that depend on the DataAccessLayer. The DataAccessDispatcher provides a simple interface for the Repository subsystem used by the DataAccessLayer.

Why is it used?

This pattern is used in order to provide set of interfaces for other subsystems to access, thereby greatly reducing potential coupling between subsystems. The DataAccessFacade is important to all subsystems depending on it because it hides the implementation of how the user inputs are saved and retrieved from the database. Since this facade knows about the entities it is meant to process, it serves subsystems with the entity information that they need in order to display and function properly. This layer is important in the design of the cuPID system since it decouples the repository from all the subsystems that need to perform persistent data storage and is critical for the DataAccessLayer’s caching mechanism. With that said, the repository dependency has shifted to this layer and now the DataAccessFacade uses the simplified DataAccessDispatcher to interface with specific implementations of the repository. The DataAccessDispatcher is used to hide the physical implementations of the complex queries

for the *IMappable* (*EO-14*) interface and delegates the incoming requests to the correct class. This interface is used to decouple the dependencies to save and retrieve for the mappable entity objects of the DataAccessLayer.

What does it achieve?

All in all, the main objective of the facade class in the cuPID system is to lower coupling between subsystems. Our considerations conclude that the cleanest way to access a subsystem is through a common interface and pass requests that both subsystems can understand. For example, the entity objects contained in the DataAccessLayer subsystem may need a proprietary implementation of a save and retrieve functionality within the repository. Without this facade, each entity object will couple to an repository implementation class creating a number of coupling relationships between the two subsystems that scales linearly base on the number of entity objects that require saving. By implementing a facade and using a common request format to communication, cuPID subsystems become reusable among other possible subsystem that may want to interact with the said subsystem. The pattern reinforces the cuPID design goal for code reuse (*DG-5)*.

Design Pattern: Proxy (DP-03)

The proxy design pattern (sometimes called middle man or surrogate) is a design pattern used to control access to a particular object by serving as a middleman or placeholder wherever the particular object is needed. The justification of the design choice as to why the Proxy design pattern was used with respect to the cuPID system is provided as answers to the following questions below:

Where is it used?

In the cuPID system, the proxy design pattern was used in order to provided indirect access to the real PPP entity object (PPPReal EO-09). This was done by creating a Proxy PPP entity object (PPPProxy EO-10) that attempts to mimic the real object as much as possible. Refer to Figure (11) for the use of this pattern. The classes related in this relationship are: ProjectPartnerProfile (EO-08), ProjectPartnerProfileReal (EO-09), ProjectPartnerProfileProxy (EO-10). These classes are contained in the *DataAccessLayer* subsystem (SS-05).

Why is it used?

The major motivation behind using the proxy design pattern was the concept of faking information as much as possible until it is really needed. This motivation has helped optimize the performance of the cuPID system by a reasonable factor. Based on the fact that some information contained in the ProjectPartnerProfile Entity like: technical scores and qualification scores needed by the algorithm take a lot of time to retrieve from the database, it was decided that information for the PPP was to be loaded as was needed for a particular functionality. By doing this, the performance of the cuPID system was guaranteed to be improved, as time is not wasted on loaded information that might never be used.

What does it achieve?

The use of the proxy pattern in the cuPID system helps prevent the loading of heavyweight objects significantly, thereby optimizing the general performance of the system. The proxy design pattern also provides room for information hiding and also room for information protection if needed.

Design Pattern: Singleton (DP-04)

Sometimes an application only needs one, and only one, instance of an object. If a particular resource is needed by a system, it's wasteful to create, destroy, and recreate the class over and over. This is particular example is why the Singleton design pattern exists. It's benefits, when used correctly, can delegate class special responsibilities to a class to enable it to create and initialize itself, restrict multiple instantiations, and encapsulates a set of global variables and functionalities. In order to justify the use of this pattern, this section will be broken down to why the pattern is used, where it is used, and what this pattern achieves for the system.

Where is it used?

The cuPID system utilizes the Singleton pattern to the fullest within the data access layer since the ownership of the DataAccessFacade (*EO-15*) is not clear. During the run time of cuPID, only one DataAccessFacade can exist within the application and specifies the common interface into the DataAccessLayer subsystem (*SS-5*).

Why is it used?

The Singleton design pattern is used because the ownership of a single instance cannot be reasonably assigned to the classes that depend on it. In the repository architecture, all subsystem traffic converges at the repository, therefore making it unclear to who has complete ownership without initializing multiple instances of the same object in the depend subsystems. The motivation behind using this design pattern was to create a controlled means of access to the core component in the cuPID architecture, the repository.

What does it achieve?

This design allows the most important resource, accessing the repository, to become shared among all subsystems that depend on saving and retrieving information. This is a desirable outcome of the cuPID system as it runs on a single host therefore only one subsystem will use the data access layer at a time. By transitivity, the *DataAccessLayer* also has reference to the DataAccessDispatcher (*EO-29*), hence the entire storage solution for the cuPID system will only ever be created once without duplication. In pursuit of the design performance goal (DG-03), data is critical to the application and always used should not be needed to be instantiated and destroyed on each computation cycle avoiding inefficiencies throughout the subsystems.

Section 4: Subsystem Services

This section aims to provide a detailed description of the services offered by each subsystem where applicable. Descriptions of the operations that make up a particular service along with classes to which the stated operation belongs are also given in this section. Finally, we provide the representation of the interrelations between the subsystems using UML component diagrams. Ball and socket notation is used to represent the services provided between the subsystems. Figure (18) provides the UML component diagram showing the services provided by the subsystems where applicable.

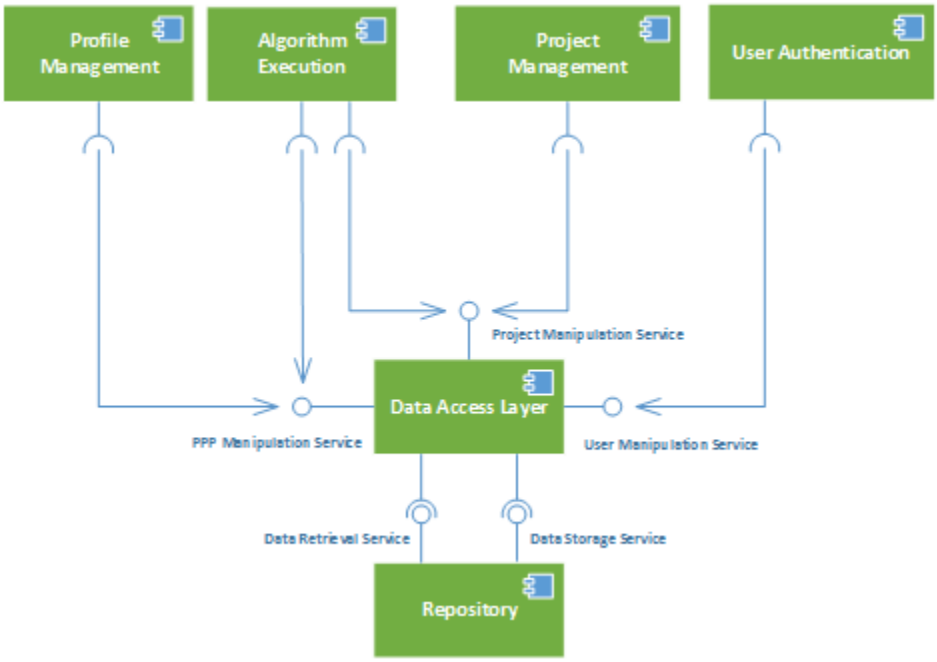


Figure 18 - UML Services Component Diagram

Subsystem Service: ProjectManipulation Service (SER-01)

This service serves to provide a means of accessing the functionality and public interfaces provided by the Project entity. The possible Actions defined by the *ActionType* parameter in this service operation are stated below:

- ***RegisterInProject***: a StudentUser registered into a given project
- ***UnRegisterFromProject***: a StudentUser unregistered from a given project
- ***DiscoverProjects***: a StudentUser wishes to see all available projects
- ***FetchUsersProjects***: fetch projects for a particular User
- ***FetchProject***: fetch a particular Project
- ***CreateProject***: an AdministratorUser created a Project
- ***UpdateProject***: an AdministratorUser updated the information in a project
- ***FetchPPPsForProject***: fetch all the PPPs for a particular project

The tabular format below provides an overview of the operations provided by this subsystem along with a description of each operation and classes related for each operation. Traceability is included where applicable for ease of backtracking

Service Name: ProjectManipulation Service (SER-01)
Providing Subsystem: DataAccessLayer (SS-05)
Receiving Subsystem: ProjectManagement (SS-01), AlgorithmExecution (SS-03)
Service Description: Provides an interface for manipulating the information for a particular project or projects.
Traceability: SS-01, SS-03, SS-05, EO-01, EO-02, EO-03, EO-04, EO-05, EO-07, EO-15, EO-25, EO-26, EO-27, EO-39,

ID	*Operation(s)*	*Description*	*Class*
01	execute(in action: ActionType, in currentUser: User*, inout project: Project*)	executes the action specified by the ActionType for a single project.	DataAccessFacade
02	execute(in action: ActionType, in currentUser: User*, out projectList: Project**)	executes the action specified by the ActionType for a list of project.	DataAccessFacade

Subsystem Service: ProfileManipulation Service (SER-02)

This service provides the means to access functionality and public interfaces provided by the ProjectPartnerProfile entity. The service is provided by the DataAccessLayer to give to the *ProfileManagement* Subsystem and also the *AlgorithmExecution* subsystem. The possible Actions defined by the ActionType object in this service operation are stated below:

- ***CreatePPP*:** create a PPP for student user
- ***FetchPPP*:** fetch the PPP for the current student that is logged in
- ***UpdatePPP*:** save the current state for an existing PPP for the current user logged in
- ***DeletePPP*:** delete a PPP that a student user created

The tabular format below provides an overview of the operations provided by this subsystem along with a description of each operation and classes related for each operation.

Service Name: ProfileManipulation Service
Providing Subsystem: *DataAccessLayer*
Receiving Subsystem: *ProfileManagement*, *AlgorithmExecution*
Service Description: Provides an interface for manipulating the information for a particular project partner profile.
Traceability: SS-02, SS-05, EO-01, EO-03, EO-08, EO-09, EO-10, EO-15, EO-34

ID	*Operation*	*Description*	*Class*
01	execute(in action: ActionType, in currentUser: User*, inout profile: PPP*)	executes action specified by the ActionType for the specified PPP	DataAccessFacade

NB: PPP represents ProjectPartnerProfile

Subsystem Service: UserManipulation Service (SER-03)

This service serves to provide a means of accessing the functionality and public interfaces provided by the User entities (User, StudentUser, AdministratorUser). The possible Actions defined by the *ActionType* object in this service operation are stated below:

- ***createAccount:*** User wishes to create account
- ***loginUser:*** Login the given user

The tabular format below provides an overview of the operations provided by this subsystem along with a description of each operation and classes related for each operation.

Service Name: UserManipulation Service
Providing Subsystem: *DataAccessLayer*
Receiving Subsystem: *UserAuthentication Subsystem*
Service Description: Provides an interface for manipulating the information for a particular User
Traceability: SS-04, SS-05, EO-01, EO-02, EO-03, EO-15, EO-36

ID	*Operation*	*Description*	*Class*
01	execute(in action: ActionType, inout user: User)	executes action specified by the ActionType for the specified User	DataAccessFacade

Subsystem Service: DataRetrieval Service (SER-04)

This service serves to provide a means of accessing the information in response to a fetch request. The table below provides an overview of the operations provided by this subsystem along with a description of each operation and classes related for each operation. The tabular format below provides an overview of the operations provided by this subsystem along with a description of each operation and classes related for each operation.

Service Name: DataRetrieval Service
Providing Subsystem: *Repository*
Receiving Subsystem: *DataAccessLayer*
Service Description: This service is provided by the Repository subsystem to output information that is requested from the database. The only subsystem that uses this service is the *DataAccessLayer*. In order to reduce coupling from the repository to the entities in the *DataAccessLayer* subsystem, an *IMappable* (EO-14) interface was defined.
Traceability: SS-05, SS-06, EO-14, EO-15, EO-25, EO-26, EO-27, EO-28, EO-29, EO-30, EO-31, EO-32, EO-33, EO-36, EO-39

ID	*Operation*	*Description*	*Class*
01	retrieveAllProjects(out projectList: IMappabble*)	retrieve all projects that exist in the database	DataAccessDispatcher
02	retrieveProjectUsingID(inout project: IMappabble)	retrieve a specified project that matches the ID	DataAccessDispatcher
03	retrieveProjectsFor User(in user: IMappable, out projectList: IMappable*)	retrieves the project that match the given User information specified as an IMappable	DataAccessDispatcher
04	retrievePPPsForProject(in project: IMappable, out profileList: IMappable*)	retrieves the registered PPPs for the project	DataAccessDispatcher
05	retrieveUserWithUsername(in userName: String, out loggedInUser: User)	retrieves information pertaining to a user if they exist	DataAccessDispatcher
06	retrievePPPForUser(in user: IMappable, out ppp: IMappable)	retrieves the PPP for a user if it exists	DataAccessDispatcher

NB: Please note that IMappable (E0-14) defines an interface in which all Savable entity objects must implement. This interface provides a means for each entity to represent themselves as key-value mappings of attributes.

Subsystem Service: DataStorage Service (SER-05)

This service serves to provide a means of accessing the information provided by a save request. The table below provides an overview of the operations provided by this subsystem along with a description of each operation and classes related for each operation.

Service Name: DataStorage Service
Providing Subsystem: *Repository*
Receiving Subsystem: *DataAccessLayer*
Service Description: This service is provided by the Repository subsystem to output information that is requested from the database. The only subsystem that uses this service is the DataAccessLayer. In order to reduce coupling from the repository to the entities in the DataAccessLayer subsystem, an *IMappable* (EO-14) interface was defined.
Traceability: SS-05, SS-06, EO-14, EO-15, EO-25, EO-26, EO-27, EO-28, EO-29, EO-30, EO-31, EO-32, EO-33, EO-34, EO-36, EO-39

ID	*Operation*	*Description*	*Class*
01	userCreatedPPP(in currentUser: IMappable, in profile: IMappable)	store the newly created PPP into the repository	DataAccessDispatcher
02	userUpdatedPPP(in currentUser: IMappable, in profile: IMappable)	store the newly updated PPP into the repository	DataAccessDispatcher
03	userDeletedPPP(in currentUser: IMappable, in profile: IMappable)	delete the PPP from the repository	DataAccessDispatcher
04	createUser(in newUser: IMappable)	store the newly created user into the repository	DataAccessDispatcher
05	userCreatedProject(in currentUser: IMappable, in newProject: IMappable)	store the the newly created project for the current user into the repository	DataAccessDispatcher
06	userUpdatedProject(in currentUser: IMappable, in updatedProject: IMappable)	store the newly updated project into the repository	DataAccessDispatcher
07	userRegisteredInProject(in currentUser: IMappable, in ppp: IMappable, in project: IMappable)	save the registration status for a student on a specific project	DataAccessDispatcher
08	userUnRegisteredInProject(in project IMappable, in ppp: IMappable)	save the unregistration status for a student on a specific project	DataAccessDispatcher

NB: Please note that IMappable (E0-14) defines an interface in which all Savable entity objects must implement. This interface provides a means for each entity to represent themselves as key-value mappings of attributes.

Section 5: Class Interfaces

In this proceeding section of the system design, the designed subsystem services are described further to identify the interfaces for all classes that work together to provide the operation of a particular service. Through the use of UML class diagram notations, this section will lead the reader through each class that is involved with any service showing their attributes and operations along with the abstraction levels (private, public or protected) of each attribute and operation. For each service, the operations along with interacting classes are provided for ease of understanding.

DataRetrieval Service (SER-04)

The DataRetrieval Service provides a vast list of services to the *DataAccessLayer(DAL)* subsystem. These services are all made available by the *DataAccessDispatcher* class which acts as a Facade class. This class (*DataAccessDispatcher)* provides an amalgamated list of public interfaces which can be used to retrieve different types of data from the Database. In order to reduce coupling from the *Repository* subsystem to the entity objects in the DAL, an IMappable (EO-14) interface is used as the means of communication between the DAL and the *Repository* Subsystem. The IMappable interface defines a means for classes to represent themselves as key-value pair mappings which will be easy for the Repository to obtain entity information. When necessary, the DataAccessDispatcher modifies the values in the IMappable object to provide necessary information needed by the classes that provide the particular functionality. The operations provided by this service are provided below, along with the with interacting classes that work together to provide the classes. Care has also been taken to add traceability where applicable.

Operation: + retrieveAllProjects(out projectList: IMappabble*) : bool
Interacting Classes: DataAccessDispatcher (EO-29), ProjectRepository (EO-33), DBManager (EO-28)
Traceability: SS-05, SS-06, SER-04

Operation: + retrieveProjectUsingID(inoutout project: IMappabble) : bool
Interacting Classes: DataAccessDispatcher (EO-29), ProjectRepository (EO-33), DBManager (EO-28)
Traceability: SS-05, SS-06, SER-04

Operation: + retrieveProjectsForUser(in user: IMappable, out projectList: IMappable*) : bool
Interacting Classes: DataAccessDispatcher (EO-29), ProjectRepository (EO-33), DBManager (EO-28)
Traceability: SS-05, SS-06, SER-04

Operation: + retrievePPPsForProject(in project: IMappable, out profileList: IMappable*): bool
Interacting Classes: DataAccessDispatcher (EO-29), ProjectRepository (EO-33), DBManager (EO-28)
Traceability: SS-05, SS-06, SER-04

Operation: + retrieveUserWithUsername(in userName: String, out loggedInUser: User) : bool
Interacting Classes: DataAccessDispatcher (EO-29), UserRepository (EO-31), DBManager (EO-28)
Traceability: SS-05, SS-06, SER-04

Operation: + retrievePPPForUser(in user: IMappable, out profile: IMappable) : bool
Interacting Classes: DataAccessDispatcher (EO-29), ProfileRepository (EO-32), DBManager (EO-28)
Traceability: SS-05, SS-06, SER-04

Figure 19 below illustrates the UML class diagram which shows the relationships present in this subsystem, along with the class attributes, and member functions.

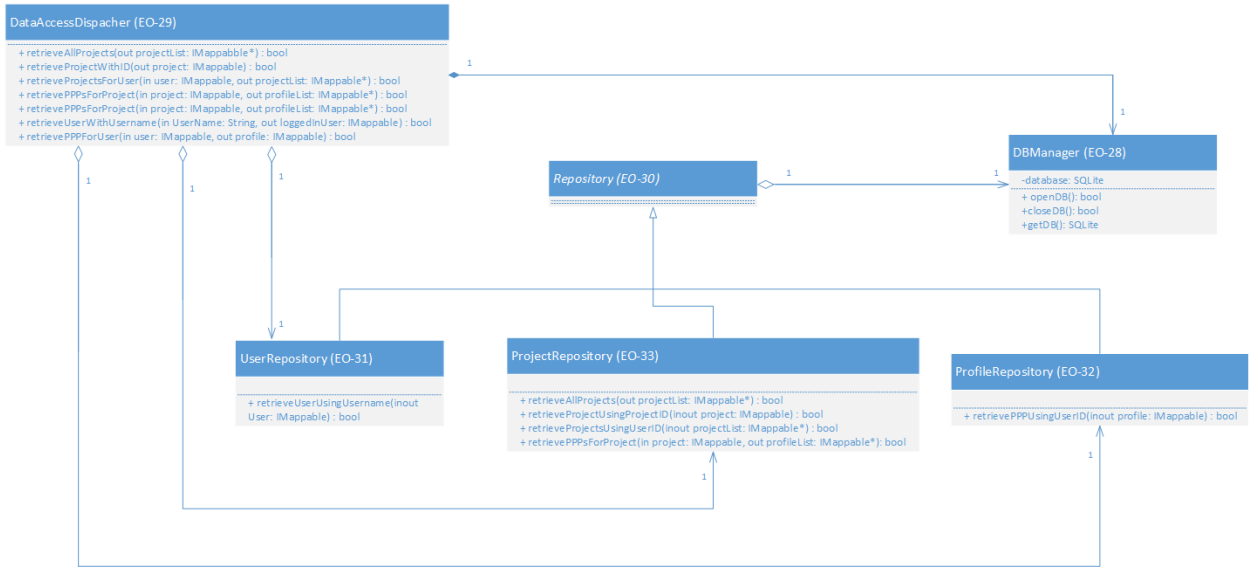


Figure 19 - UML Class Diagram for DataRetrieval Service

The classes represented in the UML class diagram above work together to provide the DataRetrieval service of the *Repository* subsystem. A brief description of these classes is provided. (Please refer to Table 2 in the Appendix section for a detailed description of these classes.)

- **DataAccessDispatcher (EO-29)**: A facade class that provides a set of public interfaces for interacting with the repository subystem.
- ***Repository*** **(EO-30**): An abstract class that provides a set of interfaces for performing database operations.
- **UserRepository (EO-31**): A concrete repository class that provides a set of interfaces for performing operations related to the user entity object.
- **ProjectRepository (EO-33**) : A concrete repository class that provides a set of interfaces for performing operations related to the project entity object.
- **ProfileRepository (EO-32)** : A concrete repository class that provides a set of interfaces for performing operations related to the profile entity object.
- **DBManager (EO-28)**: A class responsible for providing access to the SQLite database.

DataStorage Service (SER-05)

This data storage service provides operations from the Repository subsystem (*SS-6*) to save *IMappable (EO-14)* entity objects passed from the DataAccessLayer (DAL) (*SS-5*). This service is designed with facade design pattern by DataAccessDispatcher to interface requests from the DAL to seek the correct derived repository classes. All other components interface the DataStorage Service through this facade class, hence, the operations of the interface all belong to the DataAccessDispatcher class. This services utilizes the full class structure of the repository service in order to provide persistent data functionality to the cuPID application. Since the acting subsystem only understands how to perform an action through the IMappable interface, this type is a serialization of all savable entity objects into key value pairs where the key represent a the object property's name, and value represents the property's value. The dispatch performs changes into the IMappable entity in order to facilitate and structure the request used, to further delegate the save functionality below to a repository that know how to save that entity. . The operations provided by this service are provided below, along with the with interacting classes that work together to provide the classes. Care has also been taken to add traceability where applicable.

Operation: + userCreatedPPP(in currentUser: IMappable, in profile: IMappable) : bool
Interacting Classes: DataAccessDispatcher (EO-29), DBManager (EO-28), ProfileRepository (EO-32)
Traceability: SS-5, SS-6, SER-04, SER-05

Operation: + userUpdatedPPP(in currentUser: IMappable, in profile: IMappable) : bool
Interacting Classes: DataAccessDispatcher (EO-29), DBManager (EO-28), ProfileRepository (EO-32)
Traceability: SS-5, SS-6, SER-04, SER-05

Operation: + userDeletedPPP(in currentUser: IMappable, in profile: IMappable) : bool
Interacting Classes: DataAccessDispatcher (EO-29), DBManager (EO-28), ProfileRepository (EO-32)
Traceability: SS-5, SS-6, SER-04, SER-05

Operation: + createUser(in newUser: IMappable) : bool
Interacting Classes: DataAccessDispatcher (EO-29), DBManager (EO-28), UserRepository (EO-31)
Traceability: SS-5, SS-6, SER-04, SER-05

Operation: + userCreatedProject(in currentUser: IMappable, in newProject: IMappable) : bool
Interacting Classes: DataAccessDispatcher (EO-29), DBManager (EO-28), ProjectRepository (EO-33)
Traceability: SS-5, SS-6, SER-04, SER-05

Operation: + userUpdatedProject(in currentUser: IMappable, in updatedProject: IMappable) : bool
Interacting Classes: DataAccessDispatcher (EO-29), DBManager (EO-28), ProjectRepository (EO-33)
Traceability: SS-5, SS-6, SER-04, SER-05

Operation: + userRegisteredInProject(in currentUser: IMappable, in ppp: IMappable, in project: IMappable) : bool
Interacting Classes: DataAccessDispatcher (EO-29), DBManager (EO-28), ProjectRepository (EO-33)
Traceability: SS-5, SS-6, SER-04, SER-05

Operation: + userUnRegisteredInProject(in project IMappable, in ppp: IMappable) : bool
Interacting Classes: DataAccessDispatcher (EO-29), DBManager (EO-28), ProjectRepository (EO-33)
Traceability: SS-5, SS-6, SER-04, SER-05

The UML class diagram shown belows demonstrates the active classes that provide the persistent storage service to the cuPID system.

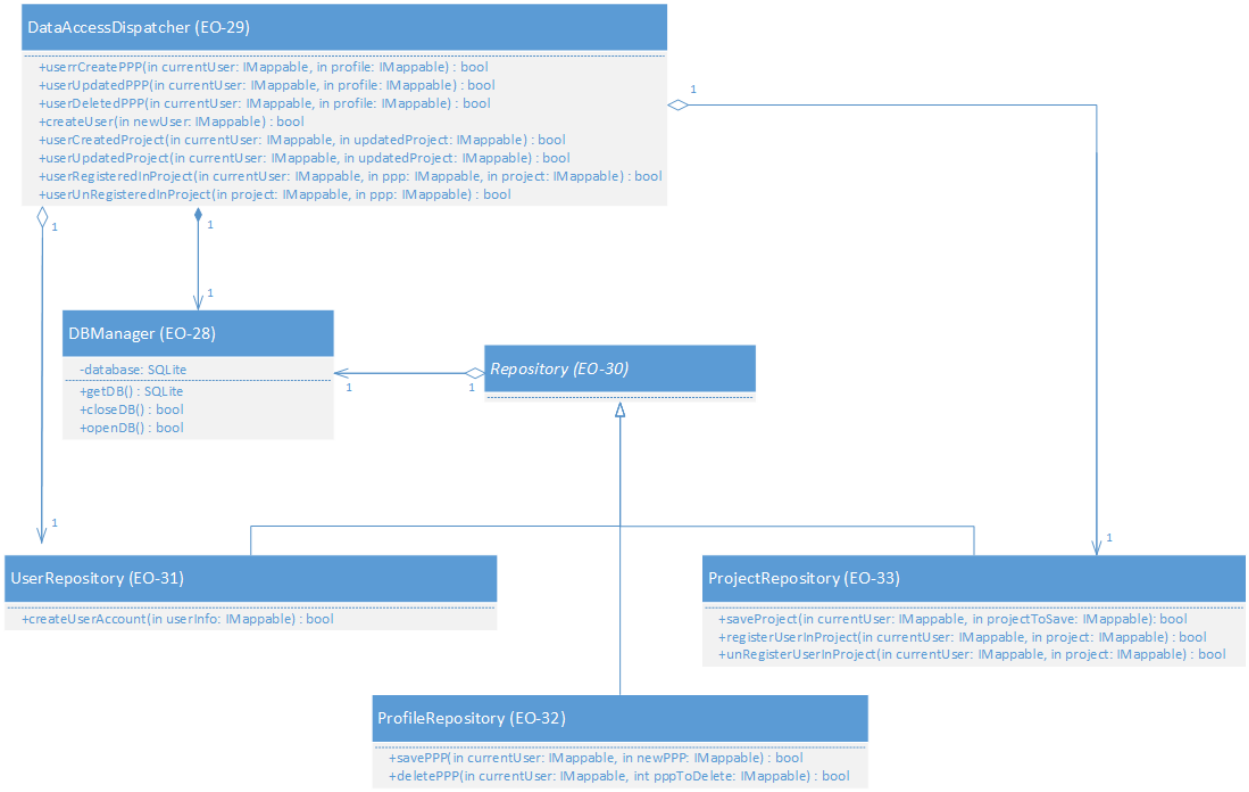


Figure 20 - UML Class Diagram for DataStorage Service

The classes represented in the UML class diagram above work together to provide the DataStorage service of the *Repository* subsystem. A brief description of these classes is provided. (Please refer to Table 2 in the Appendix section for a detailed description of these classes)

- **DataAccessDispatcher (EO-29)**: A facade class that provides a set of public interfaces for interacting with the repository subsystem.
- ***Repository*** **(EO-30**): An abstract class that provides a set of interfaces for performing database operations.
- **UserRepository (EO-31**): A concrete repository class that provides a set of interfaces for performing operations related to the user entity object.
- **ProjectRepository (EO-33**) : A concrete repository class that provides a set of interfaces for performing operations related to the project entity object.
- **ProfileRepository (EO-32)** : A concrete repository class that provides a set of interfaces for performing operations related to the profile entity object.
- **DBManager (EO-28)**: A class responsible for providing access to the SQLite database.

ProfileManipulation Service (SER-02)

The *ProfileManipulation* service provides a set of interfaces for interacting with a Profile in the cuPID subsystem. The service provides only one operation. This operation takes an ActionType which specifies the possible actions that can be executed on a Profile. The operations provided by this service are provided below, along with the with interacting classes that work together to provide the classes. Care has also been taken to add traceability where applicable.

Operation: execute(in action: ActionType, in currentUser: User, inout profile: PPP*)
Interacting Classes: DataAccessFacade (EO-14), *ProjectPartnerProfile* (EO-08), ProjectPartnerProfileReal (EO-09), ProjectPartnerProfileProxy (EO-10), Qualification (EO-11), StudentUser (EO-03)
Traceability: SS-05, SS-01, SS-02, SS-03, SER-04, SER-05

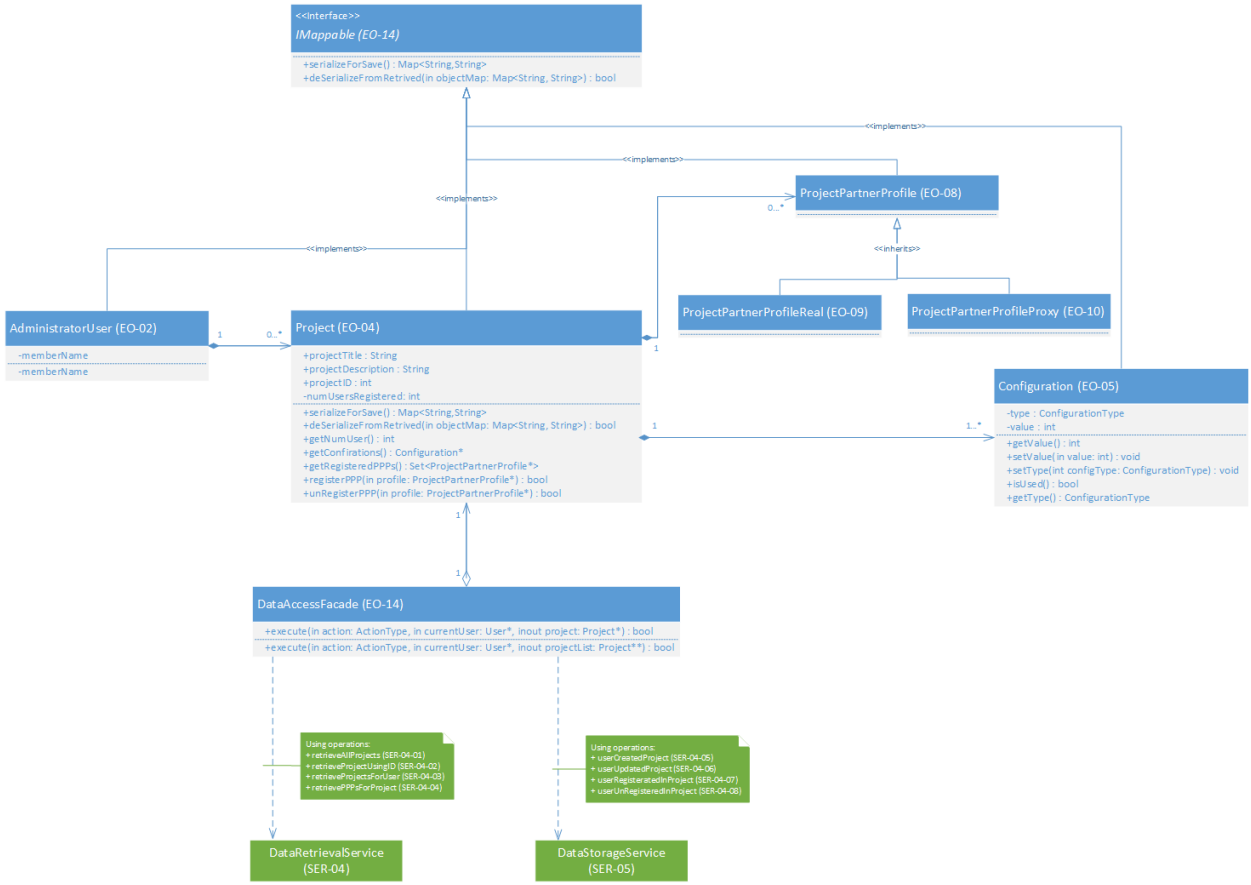


Figure 21 - UML Class Diagram for ProfileManipulation Service

The classes represented in the UML class diagram above work together to provide the *ProjectManipulation* service (SER-01) of the *DataAccessLayer* subsystem (SS-05). A brief description of these classes is provided. (Please refer to Table 2 in the Appendix section for a detailed description of these classes.)

- **DataAccessFacade (EO-14)**: A facade class responsible for providing a combination of all the interfaces for interacting with the DataAccessLayer subsystem (SS-05).
- **IMappable (EO-14)**: This interface specifies a means for classes to represent their defining attributes as key-value pairs in a map in order to be easily stored and retrieved from the database.
- ***ProjectPartnerProfile (EO-08)***: An abstract class that provides a set of interfaces for interacting with the *ProjectPartnerProfile* entity.
- **ProjectPartnerProfileReal (EO-09):** A concrete class that provides a set of interfaces for interacting with the *ProjectPartnerProfile* entity.
- **ProjectPartnerProfileProxy (EO-10):** A proxy class that imitates the interfaces of the ProjectPartnerProfileReal (EO-09) as much as possible in order to prevent computationally intensive loading of the real entity.
- **Qualification (EO-11):** A class responsible for storing a particular qualification associated with a StudentUser.
- **StudentUser (EO-03):** A concrete *User* class that provides a set of interfaces that a StudentUser is able to execute in the cuPID system.

ProjectManipulation Service (SER-01)

The *ProjectManipulation* service provides a set of interfaces for interacting with a Project in the cuPID subsystem. The service provides two operations. Both operations are similar, but one executes on a single Project while the other executes on a list of Projects. These operation take an ActionType which specifies the possible actions that can be executed on a Project. The operations provided by this service are provided below, along with the with interacting classes that work together to provide the classes. Care has also been taken to add traceability where applicable.

Operation: execute(in action: ActionType, in currentUser: User, inout project: Project*)
Interacting Classes: DataAccessFacade (EO-14), Project (EO-04), *ProjectPartnerProfile* (EO-08), ProjectPartnerProfileReal (EO-09), ProjectPartnerProfileProxy (EO-10), AdministratorUser (EO-01)
Traceability: SS-01, SS-03, SS-04, SER-01, SER-04, SER-05

Operation: execute(in action: ActionType, in currentUser: User, out projectList: Project**)
Interacting Classes: DataAccessFacade (EO-14), Project (EO-04), *ProjectPartnerProfile* (EO-08), ProjectPartnerProfileReal (EO-09), ProjectPartnerProfileProxy (EO-10), AdministratorUser (EO-01)
Traceability: SS-01, SS-03, SS-04, SER-01, SER-04, SER-05

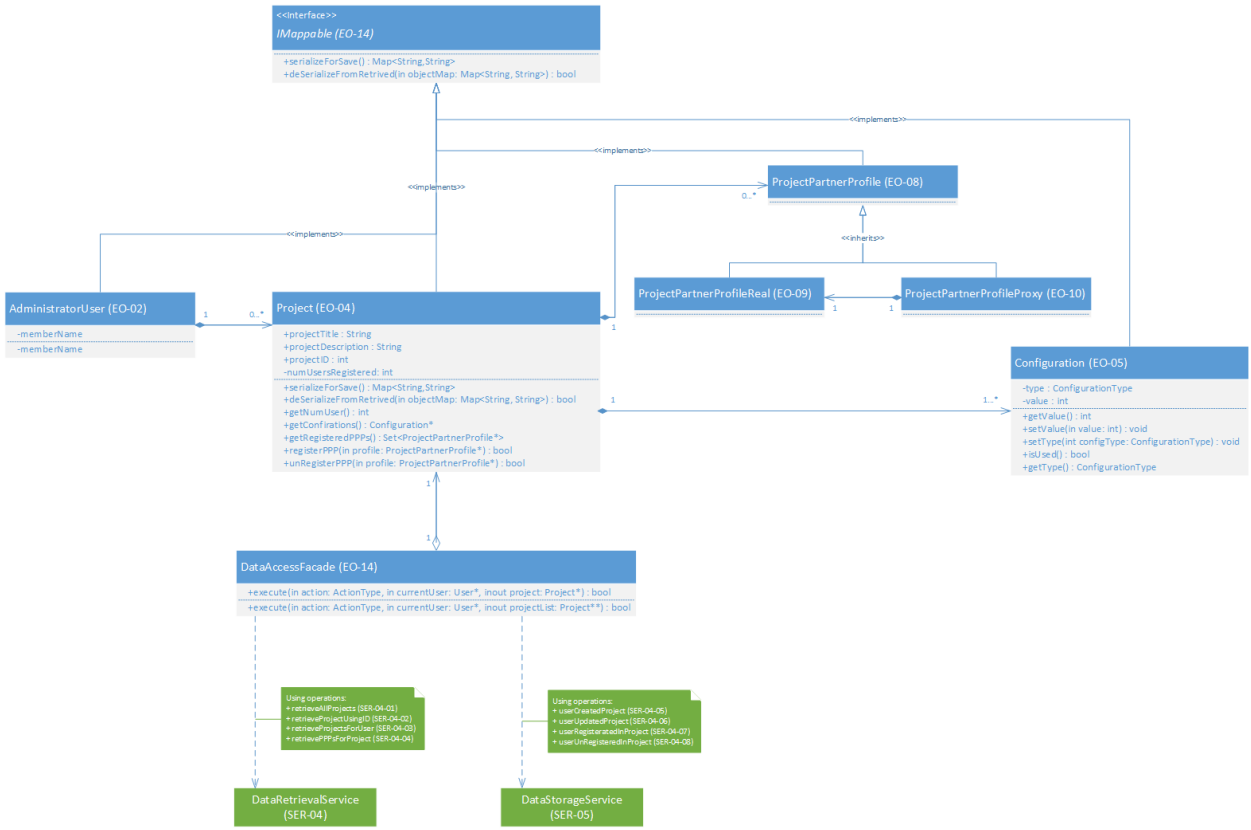


Figure 22 - UML Class Diagram for ProjectManipulation Service

The classes represented in the UML class diagram above work together to provide the *ProjectManipulation* service (SER-01) of the *DataAccessLayer* subsystem (SS-05). A brief description of these classes is provided. (Please refer to Table 2 in the Appendix section for a detailed description of these classes.)

- **DataAccessFacade (EO-14)**: A facade class responsible for providing a combination of all the interfaces for interacting with the DataAccessLayer subsystem (SS-05).
- **IMappable (EO-14)**: This interface specifies a means for classes to represent their defining attributes as key-value pairs in a map in order to be easily stored and retrieved from the database.
- **AdministratorUser (EO-02):** A concrete AdministratorUser class that provides a set of interfaces that a StudentUser is able to execute in the cuPID system.
- **Project (EO-04)**: This class helps provide a set of interfaces for interacting with the Project entity
- ***ProjectPartnerProfile (EO-08)***: An abstract class that provides a set of interfaces for interacting with the *ProjectPartnerProfile* entity.
- **ProjectPartnerProfileReal (EO-09):** A concrete class that provides a set of interfaces for interacting with the *ProjectPartnerProfile* entity.
- **ProjectPartnerProfileProxy (EO-10):** A proxy class that imitates the interfaces of the ProjectPartnerProfileReal (EO-09) as much as possible in order to prevent computationally intensive loading of the real entity.
- **Configuration (EO-05):** A class responsible for storing configuration values for a project which is used by the *InsomniaAlgorithm*.

UserManipulation Service (SER-03)

The *UserManipulation* service provides a set of interfaces for interacting with a User in the cuPID subsystem. The service provides only one operation. This operation takes an ActionType which specifies the possible actions that can be executed on a User. The operations provided by this service are provided below, along with the with interacting classes that work together to provide the classes. Care has also been taken to add traceability where applicable.

Operation: execute(in action: ActionType, inout user: User)

Interacting Classes: DataAccessFacade

Traceability: SS-04, SS-05, SER-03, SER-04, SER-05

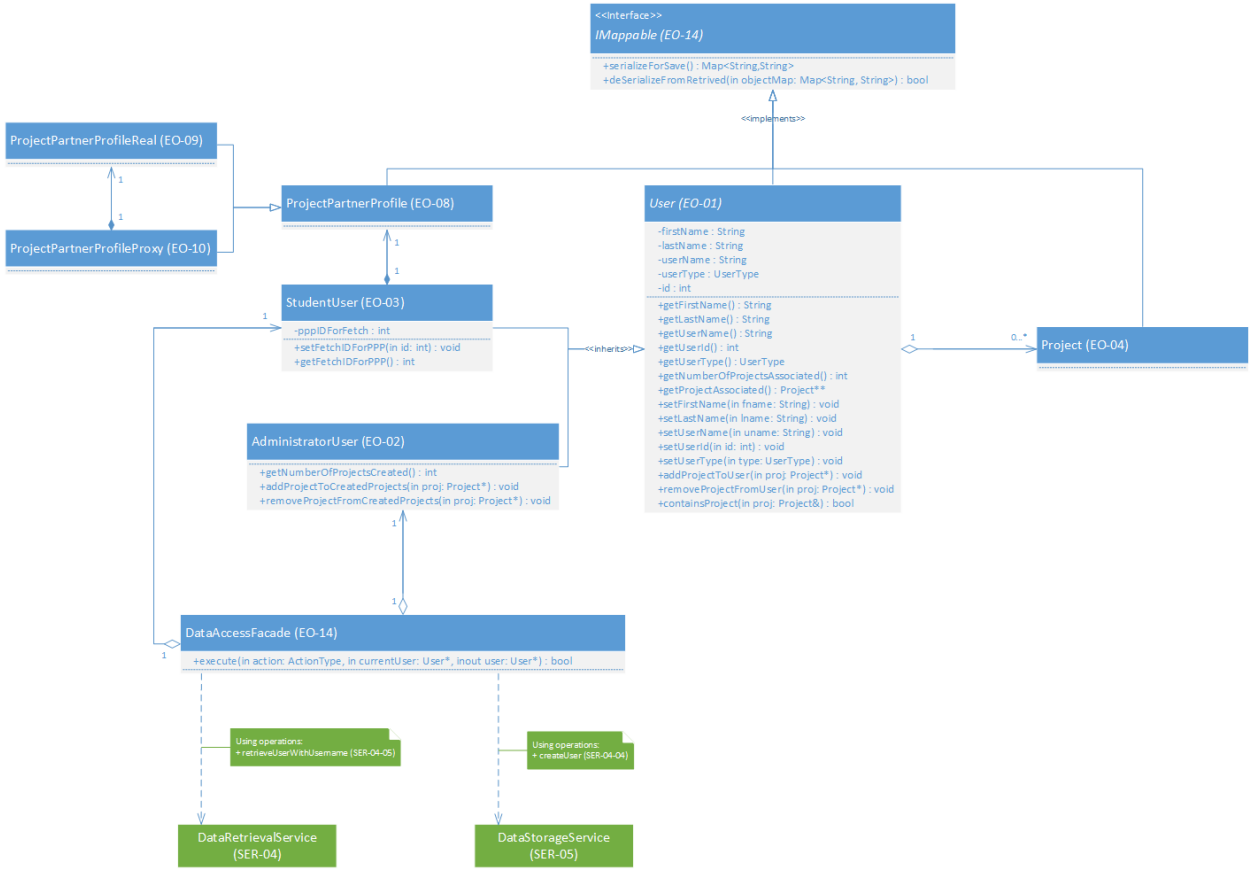


Figure 23 - UML Class Diagram for UserManipulation Service

The classes represented in the UML class diagram above work together to provide the *UserManipulation* service (SER-01) of the *DataAccessLayer* subsystem (SS-05). A brief description of these classes is provided. (Please refer to Table 2 in the Appendix section for a detailed description of these classes.)

- **DataAccessFacade (EO-14)**: A facade class responsible for providing a combination of all the interfaces for interacting with the DataAccessLayer subsystem (SS-05).
- **IMappable (EO-14)**: This interface specifies a means for classes to represent their defining attributes as key-value pairs in a map in order to be easily stored and retrieved from the database.
- ***User (EO-01):*** An abstract class that provides a set of base interfaces available for the user class in the cuPID system.
- **StudentUser (EO-03):** A concrete *User* class that provides a set of interfaces that a StudentUser is able to execute in the cuPID system.
- **AdministratorUser (EO-02):** A concrete *User* class that provides a set of interfaces that an AdministratorUser is able to execute in the cuPID system.
- **Project (EO-04)**: This class helps provide a set of interfaces for interacting with the Project entity
- ***ProjectPartnerProfile (EO-08)***: An abstract class that provides a set of interfaces for interacting with the *ProjectPartnerProfile* entity.
- **ProjectPartnerProfileReal (EO-09):** A concrete class that provides a set of interfaces for interacting with the *ProjectPartnerProfile* entity.
- **ProjectPartnerProfileProxy (EO-10):** A proxy class that imitates the interfaces of the ProjectPartnerProfileReal (EO-09) as much as possible in order to prevent computationally intensive loading of the real entity.

Appendix

This section serves to provide a reference to diagrams and tables not completely paramount to understanding the contents of this document. The essence of this section is to provide the reader with additional resources that will aid the understanding of the document, and also make available figures used in the justification of design choices made in the document. The reader is to use this section as deemed necessary to aid the understanding of concepts portrayed that elude his/her understanding during the course of reading this document.

Appendix - Table 1 - cuPID Prototype subsystems and Comprising Classes

Subsystem	Comprising Classes	Description	Traceability
StorageSubsystem	Storage	Main database dispatch class, used by the *ProfileManagement*, *ProjeceManagement,* and the *UserAuthentication* subsystem to interface with the repository	UC-18, NFR-03, NFR-08,NFR-09
	DatabaseManager	Database type, version, and object class to hold data on the type of database the system is using	UC-18, NFR-03, NFR-08, NFR-09, NFR-25
	UserRepository	Relational queries pertaining a user custom to the implementation of the *DatabaseManager*	UC-18, NFR-03, NFR-08,NFR-09
	ProjectRepository	Relational queries pertaining a project custom to the implementation of the *DatabaseManager*	UC-18, NFR-03, NFR-08,NFR-09
	cuPIDSession	A container class to store session data such as the current logged in user and the project that they are viewing	Not Applicable
ProfileManagement Subsystem	PPPController	Main controller to facilitate the create and update functionality of a profile	UC-03, UC-04, UC-05, UC-06
	ProjectPartnerProfile	An entity class representing the qualifications that the belong to the *StudentUser*	UC-03, UC-04, UC-05, UC-06
	Qualification	An atomic qualification entity with key value pair	UC-16
	ProfileWidget	A Boundary Object, tightly coupled with the PPPController to facilitate user interactions on the *ProjectPartnerProfile*	UC-04, UC-05, UC-06, NFR-04, NFR-06

ProjectManagement Subsystem	Project	An entity class that represents a project with its configurations and descriptive information	UC-11, UC-12
	Configuration	An atomic configuration entity with key value pair	UC-10
	CreateProjectWidget	A Boundary object, integrated with a controller to facilitate the creation of projects in the system	UC-09, UC-13, UC-19, UC-17, NFR-04,NFR-06
	EditTeamDialog	An UI Dialog integrated with a controller to facilitate editing a project's configuration and descriptive information	UC-10, UC-17, NFR-04, NFR-06
	ProjectDetailsWidget	A Boundary object integrated with a controller to facilitate viewing details of a project, registration functionality, and project editing	UC-08, UC-011, UC-12, UC-15
	ProjectListWidget	A Boundary object integrated with a controller to facilitate rendering multiple elements of the project list	UC-07, NFR-03,
UserAuthentication Subsystem	User	An entity class that represents a generic user and their information alongside the projects they are associated with	NFR-07
	StudentUser	A specialized entity class representing students and their profile	UC-15, NFR-07
	AdministratorUser	A specialized entity class representing admins of the system	NFR-05, NFR-07

	LoginForm	A boundary object integrated with a controller to facilitate the login process of the system	UC-17, UC-18, NFR-03, NFR-04, NFR-06, NFR-08, NFR-09
	SignUpForm	A boundary object integrated with a controller to facilitate the account creation process of the system	UC-18, NFR-03, NFR-04, NFR-06, NFR-08, NFR-09

Appendix - Table 2 - cuPID System Comprising Classes

ID	Classes (Entity/Control/Boundary)	Definition	Traceability
EO-01	*User*	A user that accesses the cuPID system.	All Use cases UC-01 - UC-19
EO-02	*Administrator User*	A user that has the ability to create projects and initiate the *InsomniaMatchingAlgorithm*	UC-02, UC-07, UC-08, UC-09, UC-10,
EO-03	*StudentUser*	A user that has only one Project Partner Profile used by the *InsomniaMatchingAlgorithm* to find a team match and a list of projects he/she is assigned to.	UC-02, UC-03, UC-04, UC-05, UC-06, UC-11, UC-12
EO-04	*Project*	An entity that contains a title, a description, a list of project configurations and a list of PPPs of the StudentUsers' who registered for the project.	UC-07, UC-08, UC-09, UC-10, UC-11, UC-12, UC-13, UC-14, UC-17, UC-18, UC-19
EO-05	*Configuration*	An entity that specifies the settings required by the *InsomniaMatchingAlgorithm*.	UC-08, UC-09, UC-10, UC-13, UC-19
EO-06	*CupidSystem*	The system that manages the creation of teams containing StudentUsers using the *InsomniaMatchingAlgorithm* initiated by an AdministratorUser. It also keeps track of all the projects and profiles currently created in the system.	All Use cases UC-01 - UC-19
EO-07	*InsomniaMatchingAlgorithm*	An entity that is used to create teams of StudentUsers based on the configurations it is initialized with.	UC-08, UC-12, UC-13, UC-14, UC-19, NFR-18
EO-08	*Project Partner Profile*	An entity that contains a bio and two list of qualifications: the StudentUser's personal qualifications and the qualifications of a teammate that the StudentUser would like to work with.	UC-01, UC-03, UC-04, UC-05, UC-06, UC-15, UC-16, UC-18
EO-09	*ProjectPartner ProfileReal*	A concrete subclass of the *ProjectPartnerProfile* class that encapsulates all the data related to a Profile	Not Applicable

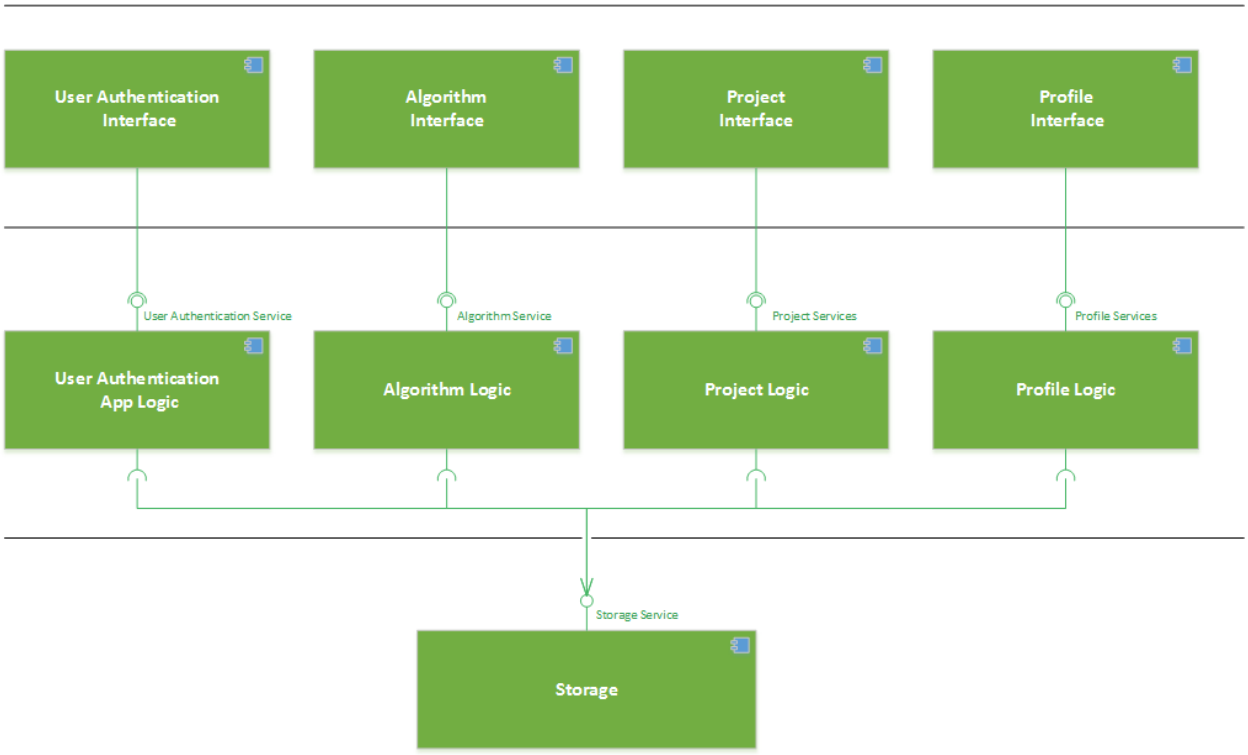
EO-10	*ProjectPartner ProfileProxy*	A concrete subclass of the *ProjectPartnerProfile* that exists as a proxy to evade computationally intensive loading of the *ProjectPartnerProfileReal* object	Not Applicable
EO-11	*Qualifications*	An entity that demonstrates capabilities of a user.	UC-01, UC-04, UC-05
EO-12	*MatchReport*	An entity that contains the results of launching the Insomnia Matching Algorithm.	UC-08, UC-14
EO-13	Team	An entity object that contains the list of matched StudentUsers.	UC-08, UC-14
EO-14	IMappable	An Abstract Interface that defines the method of serialization of entity objects to and from a key-value pair format recognized by the *Repository* entity	Not Applicable
EO-15	DataAccessFacade	A facade class responsible for providing a set of data access interfaces for ease of use by the rest of the component classes of the system	Not Applicable
EO-16	*AbstractProject ManagerFactory*	Abstract class responsible for creating the correct Controller and View for the particular Project action	Not Applicable
EO-17	*ProjectList ManagerFactory*	Concrete class responsible for creating the correct Controller and View for viewing a list of projects	UC-02, UC-07, UC-18
EO-18	*ProjectDetails ManagerFactory*	Concrete class responsible for creating the correct Controller and View for viewing details of a project	UC-02, UC-07, UC-13
EO-19	*ProjectCreation ManagerFactory*	Concrete class responsible for creating the correct Controller and View for creating a project	UC-02, UC-09
EO-20	*AbstractProject View*	Abstract class from which all kinds of Project Views of the cuPID system inherit from	Not Applicable
EO-21	*ProjectListView*	Concrete Project View class responsible for displaying a list of projects to the User	UC-07, NFR-03
EO-22	*ProjectDetailsView*	Concrete Project View class responsible for displaying details of a particular project to the User	UC-08, UC-09, UC-10, UC-11, UC-12, UC-12, NFR-04, NFR-06

EO-23	*ProjectCreation View*	Concrete Project View class responsible for providing interfaces to a User for creating a project.	UC-02, UC-09
EO-24	*AbstractProject Controller*	Abstract class from which all kinds of Project Controllers of the cuPID system inherit from	Not Applicable
EO-25	*ProjectList Controller*	Concrete Project Controller class responsible for managing the interaction between the *ProjectListView* and the *Project* entity	UC-07, NFR-03
EO-26	*ProjectDetails Controller*	Concrete Project Controller class responsible for managing the interaction between the *ProjectDetailsView* and the *Project* entity	UC-08, UC-09, UC-10, UC-11, UC-12, UC-12, UC-13, UC-15, UC-17, UC-18, NFR-02, NFR-22, NFR-23
EO-27	*CreateProject Controller*	Concrete Project Controller class responsible for managing the interaction between the *ProjectCreationView* and the *Project* entity	UC-09, UC-10
EO-28	*DBManager*	A class responsible for interactions with the specified database type. Also manages opening and closing of the database	UC-18, NFR-03, NFR-08, NFR-09, NFR-25
EO-29	*DataAccessDispatcher*	A class responsible for providing an interface for making queries to the database used in the cuPID system	UC-04, UC-05, UC-06, UC-18, NFR-03, NFR-08, NFR-09
EO-30	*Repository*	An abstract class that encapsulates the persistent storage functionality of the cuPID system	UC-01, UC-02, UC-18
EO-31	*UserRepository*	A concrete class that encapsulates the persistent storage functionality related to the User Entity of the cuPID system	All UCs related User Actor
EO-32	*ProfileRepository*	A concrete class that encapsulates the persistent storage functionality related to the Profile Entity of the cuPID system	UC-01, UC-04, UC-05, UC-06, UC-18
EO-33	*ProjectRepository*	A concrete class that encapsulates the persistent storage functionality related to the Project Entity of the cuPID system	UC-02, UC-07, UC-09, UC-10, UC-11, UC-12, UC-13, UC-18

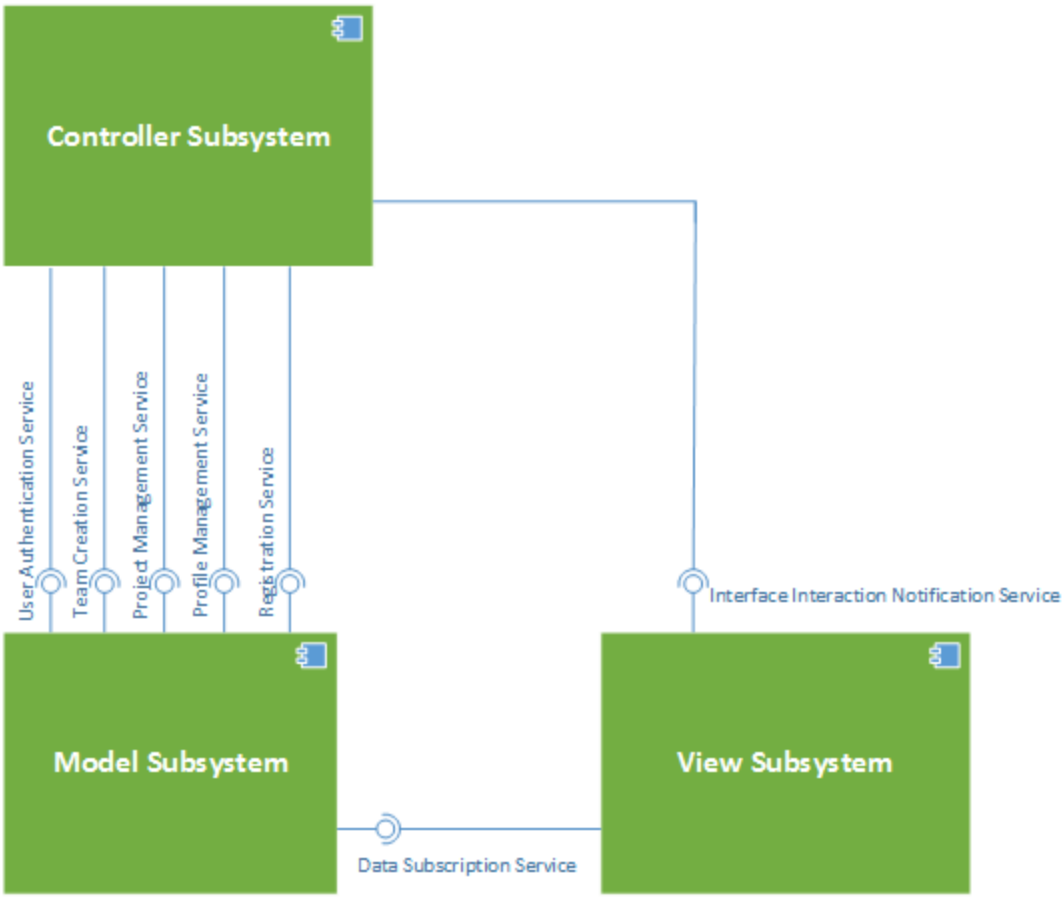
EO-34	*PPPController*	A versatile Controller responsible for managing the *ProfileWidget* and the related *ProjectPartnerProfile* entity	UC-01, UC-04, UC-05, UC-06, UC-16
EO-35	*ProfileWidget*	A Profile Boundary class responsible for providing possible interactions available to a user for his/her profile	UC-01, UC-04, UC-05, UC-06, UC-16, NFR-04, NFR-06
EO-36	*Authentication Controller*	A class responsible for handling all interactions related to authentication into the cuPID system	NFR-07, NFR-21
EO-37	*LoginForm*	A Boundary class that provides interfaces for Logging into the cuPID system	UC-17, UC-18, NFR-03, NFR-04, NFR-06, NFR-08, NFR-09
EO-38	*SignUpForm*	A Boundary class that provides interfaces for new Users to sign up to the cuPID system	UC-18, NFR-03, NFR-04, NFR-06, NFR-08, NFR-09
EO-39	*CreateTeams Controller*	A class responsible for managing the interaction between the CreatedTeamsResultView, the *Project* entity for which teams are being created and the InsomniaMatchingAlgorithm	UC-02, UC-14, UC-19, NFR-18
EO-40	*CreatedTeams ResultView*	A Boundary class that provides a set of interfaces to the User to view newly created teams for a *Project*	UC-02, UC-14, UC-19, NFR-18

Appendix - Table 3 - Abbreviations Used in the cuPID Software Design Document

Abbreviation	Full Meaning
FR	Functional Requirements
NFR	Non-Functional Requirements
PPP	Project Partner Profile
cuPID	Carleton University Project Partner Identifier
UC	Use Case
EO	Entity Object
UML	Unified Modelling Language
DAL	Data Access Layer
MVC	Model View Controller
3-T	3-tier
IO	Input Output



Appendix-Figure 1 - cuPID subsystem With Respect to 3-Tier Architectural pattern



Appendix-Figure 2 - cuPID subsystem With Respect to MVC Architectural pattern