TAEval

System Design Document

Team Romero's Severed Head
Sean Benjamin
Dylan Kristolaitis
Justin Kung
Steven Wu

Submitted to:
Dr. Christine Laurendeau
COMP3004 Object-Oriented Software Engineering
School of Computer Science
Carleton University

Contents

Figures

1. Introduction

1.1 Purpose of System

In a university setting, the main purpose of attending is for increasing knowledge in a directed, focused manner. One may learn a field of study through books borrowed from the local library, but that structure is a stark contrast to a term filled with lectures from a distinguished PhD accompanied by tests, assignments, and exams that direct the student from point A to point B. Analogously, the current structure of the TA-Instructor relationship is unfocused. In communication of duties, task obligations are set at the start of the semester; but for communication of statuses and feedback for dynamically changing tasks, we still resort to e-mail. In assessing the qualities of a candidate, TA applications still rely on providing references of faculty that need to be manually contacted to receive feedback for performance that is dated or not directly applicable to the job at hand. To allow the TAs to be successful in their job they need to have clear expectations about the tasks assigned to them for each of the courses that they TA. The instructors need to provide clear tasks and timely feedback to the TAs, to allow the TAs to complete their stated tasks at an appropriate level of satisfaction. Given that many TAs end up TA'ing repeatedly, it is invaluable for the future students in his or her section to benefit from the learning of the TA's previous errs and mistakes.

A unified system which would:

- allow the TA to know his exit criteria set by the Instructor for tasks
- allow the Instructor to know exactly if and when the task is completed from the TA
- allow the TA to receive feedback on previously completed tasks to improve upon the next instance of the same task
- allow the Administrator to be able to run reports on demand for TAs' evaluation data to judge their eligibility for future positions could solve the underlying problems with the current infrastructure

The TAEval system is the proposed system to be used by TAs, Instructors, and Administrators that will allow Instructors to assign tasks to TAs of the course they are instructing and to provide feedback to the TAs about how they are doing on their tasks.

The scope of the TAEval system is for tasks and evaluations to be assigned, completed and evaluated over the course of the term.

The TAEval system will be comprised of the following main features for the Instructor:
Instructors can create, modify and delete tasks.
Instructors will assign tasks to an associated TA.
Instructors will provide feedback and an evaluation rating for each task assigned to a TA.

The TAEval system will be comprised of the following main features for the Administrators:
Administrators will be able to manage system data such as courses, instructors, and TAs.
Administrators will be able to execute reports on evaluation data stored in TAEval's persistent data.

The TAEval system will be comprised of the following main features for the TAs:
TAs can view the tasks that have been assigned to them.
TAs can view evaluations on their tasks once they have been entered by the instructor.

For further details with regards to detailed system features, technical specifications, graphic user interface (GUI), data storage and inter-process communications refer to the TAEval system description.

1.2 Overview of Document

The purpose of the Phase #2 Model document is to design & plan ahead for the full realization of the TAEval system while building off of our first prototype. This document will mainly focus on decomposing the system into subsystems, defining our design strategies before implementation, and thinking about how each abstract subsystem interacts with another to accomplish the functionality we desire.

The document contains the following sections with regards to the TAEval system design:

- Prototype decomposition of phase #1 prototype: The decomposition will include class diagrams and UML component diagrams for phase #1 prototype.
- System decomposition of the entire TAEval system: This will include class diagrams and UML component diagrams of each of the subsystems and their dependencies.
- Design evolution discussion: Explaining the differences between the decomposition for the phase #1 prototype and the entire TAEval system.
- Hardware/software mapping: This section describes how the subsystem, components and nodes were mapped according to the Client/Server architecture. The breakdown of the components, subsystems and nodes are seen in an UML deployment diagram.
- Persistent data management: This section defines which entities are saved in storage and the type of data management system that was be used in the TAEval system.
- Design patterns: This section discusses which design patterns were used in the phase #1 prototype, along with some that we hope will be used in the TAEval system.
- Subsystem services: This section indicates the services offered by each subsystem, along with defining the operations that belong to each service, and it is accompanied with a UML component diagram.
- Class interfaces: This section describes each class that provides operations for a service and each class is represented in a UML class diagram.

2. Subsystem Decomposition

2.1 Phase #1 prototype decomposition

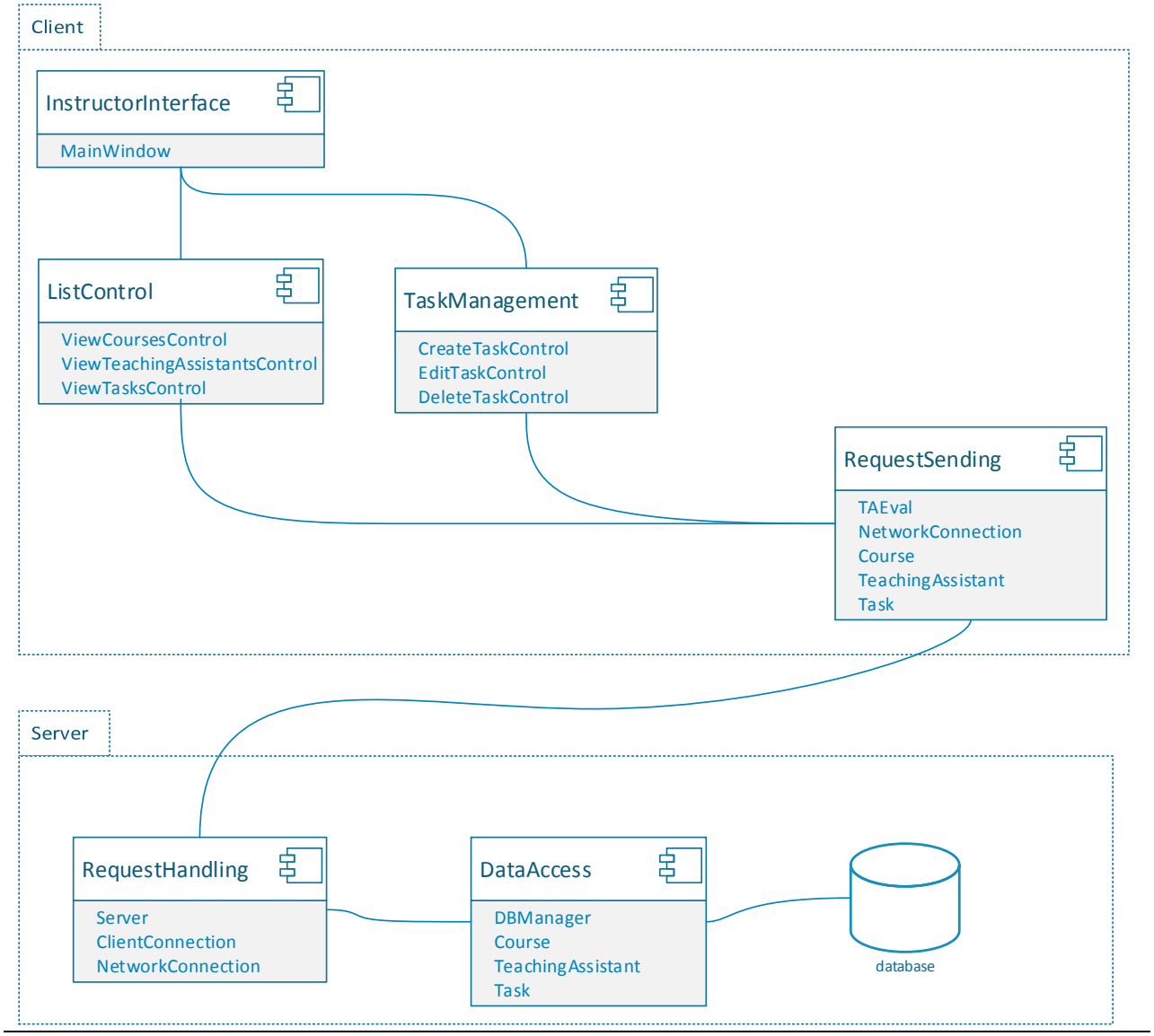


Figure 1: High-level System Decomposition

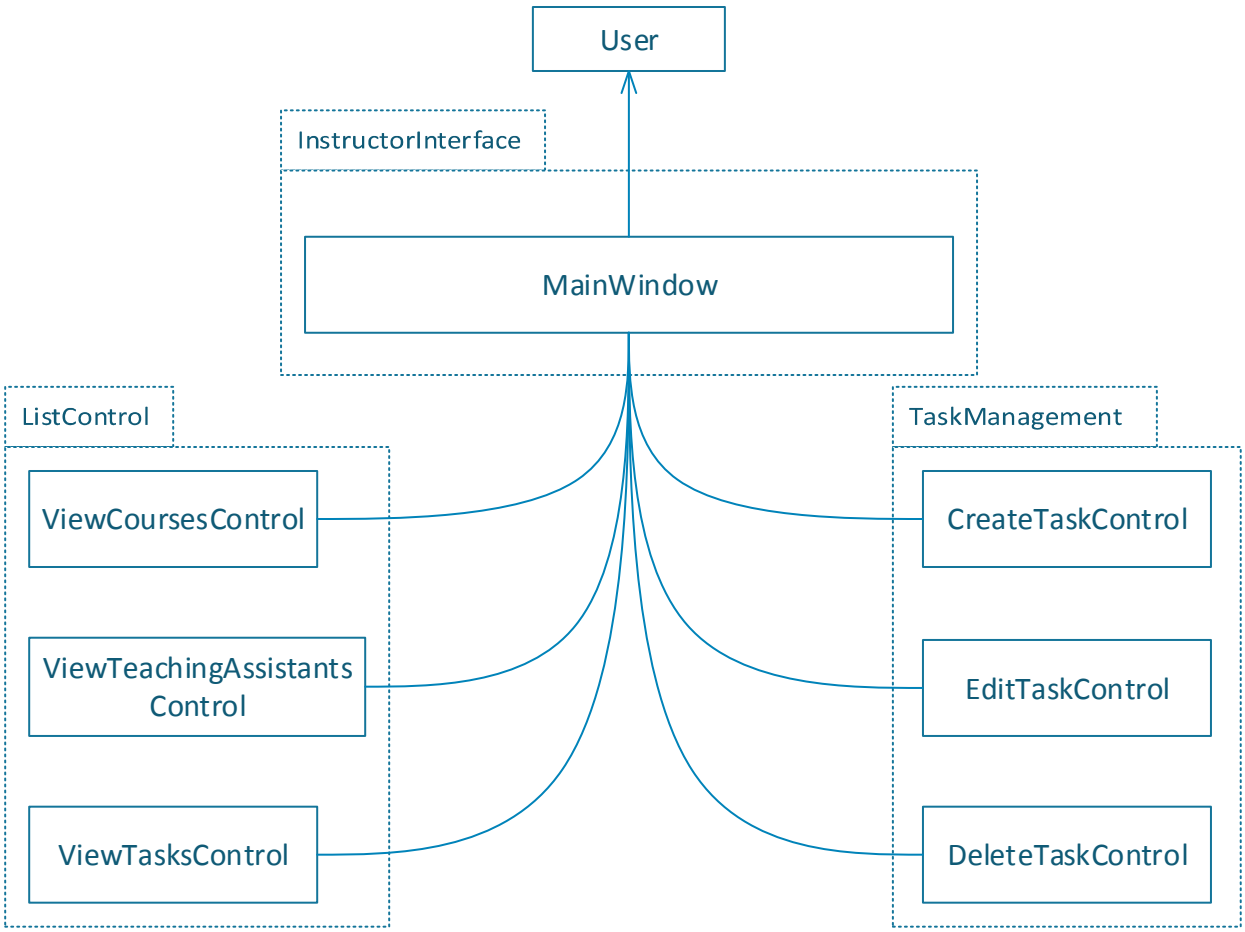


Figure 2: Detailed System Decomposition Part: A

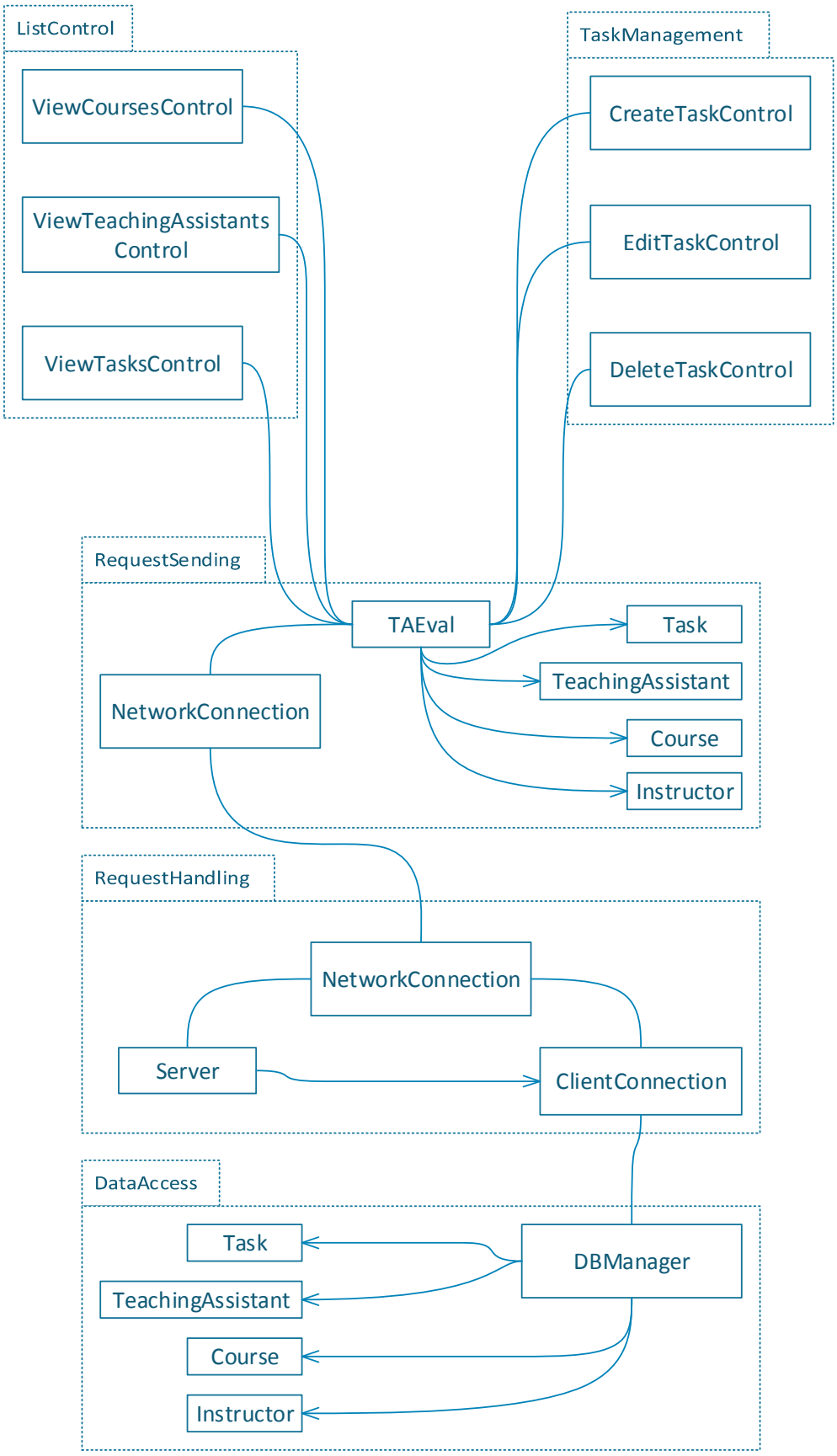


Figure 3: Detailed System Decomposition Part: B

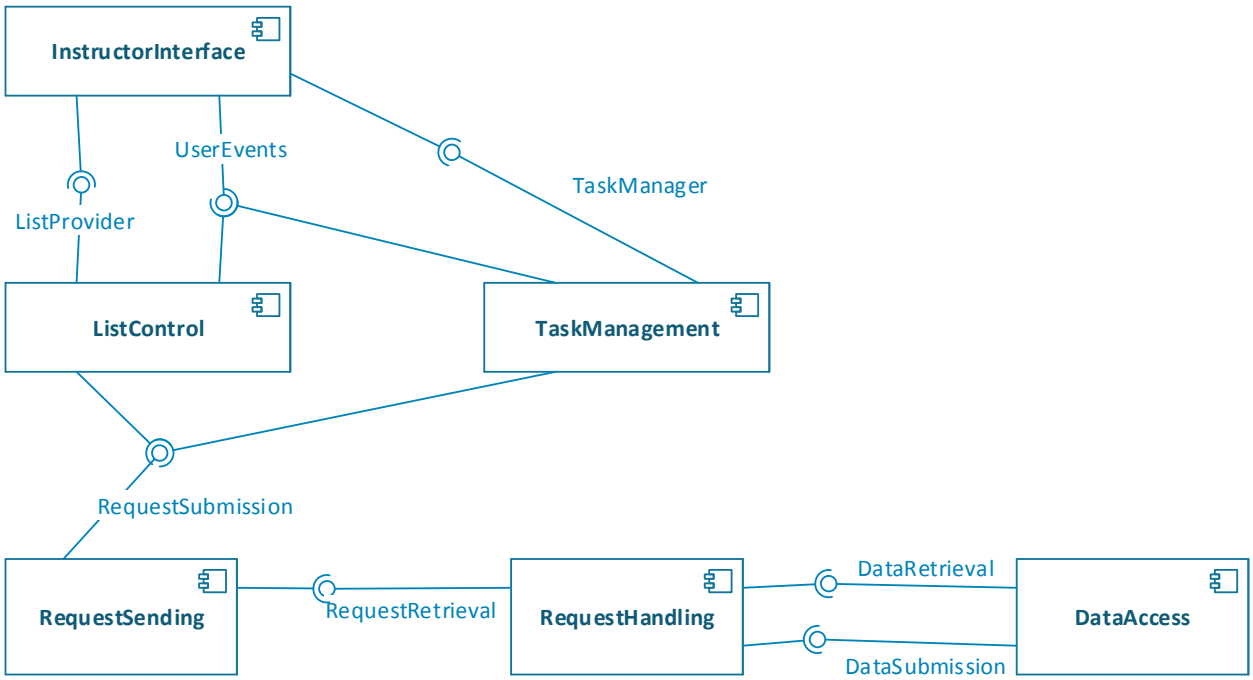


Figure 4: Component Diagram

2.1 Subsystem Description

Subsystem	**InstructorInterface**
Description	The *InstructorInterface* subsystem contains all of the user interface classes responsible for fulfilling an instructor's functional requirements. It provides a user event service to the *TaskManagement* and *ListControl* subsystems. When an instructor initiates an event this subsystem uses the manager service from the *TaskManagement* subsystem or the list provider service from the *ListControl* subsystem depending on the event.

Subsystem	**ListControl**
Description	The *ListControl* subsystem contains the classes responsible for requesting and organizing various lists on demand from the user. *ListControl* provides the list provider service to the *InstructorInterface* and makes use of the request submission service provided by the *RequestSending* subsystem.

Subsystem	**TaskManagement**
Description	The *TaskManagement* subsystem contains the classes which allow an instructor to make a request to create, modify and delete tasks. It accomplishes this by providing an event-driven management service to the *InstructorInterface* subsystem. *TaskManagement* then makes use of the request submission service provided by the *RequestSending* subsystem.

Subsystem	**RequestSending**
Description	The *RequestSending* subsystem contains the classes responsible for turning a user request into a data packet to be received by the storage server as well as initiating the client-server connection. It provides the request submission service to the *ListControl* and *TaskManagement* subsystems and makes use of the request retrieval service provided by the *RequestHandling* subsystem.

Subsystem	**RequestHandling**
Description	The *RequestHandling* subsystem contains the classes responsible for receiving a user request over a network and sending a response in return. It provides the request retrieval service to the *RequestSending* subsystem. The request retrieval service processes a user request and passes it along to the *DataAccess* subsystem via its data retrieval and submission services.

Subsystem	**DataAccess**
Description	The *DataAccess* subsystem contains the classes responsible for creating, modifying or retrieving persistent data in the database. It provides the *RequestHandling* subsystem with the data retrieval and submission services.

2.2 System decomposition

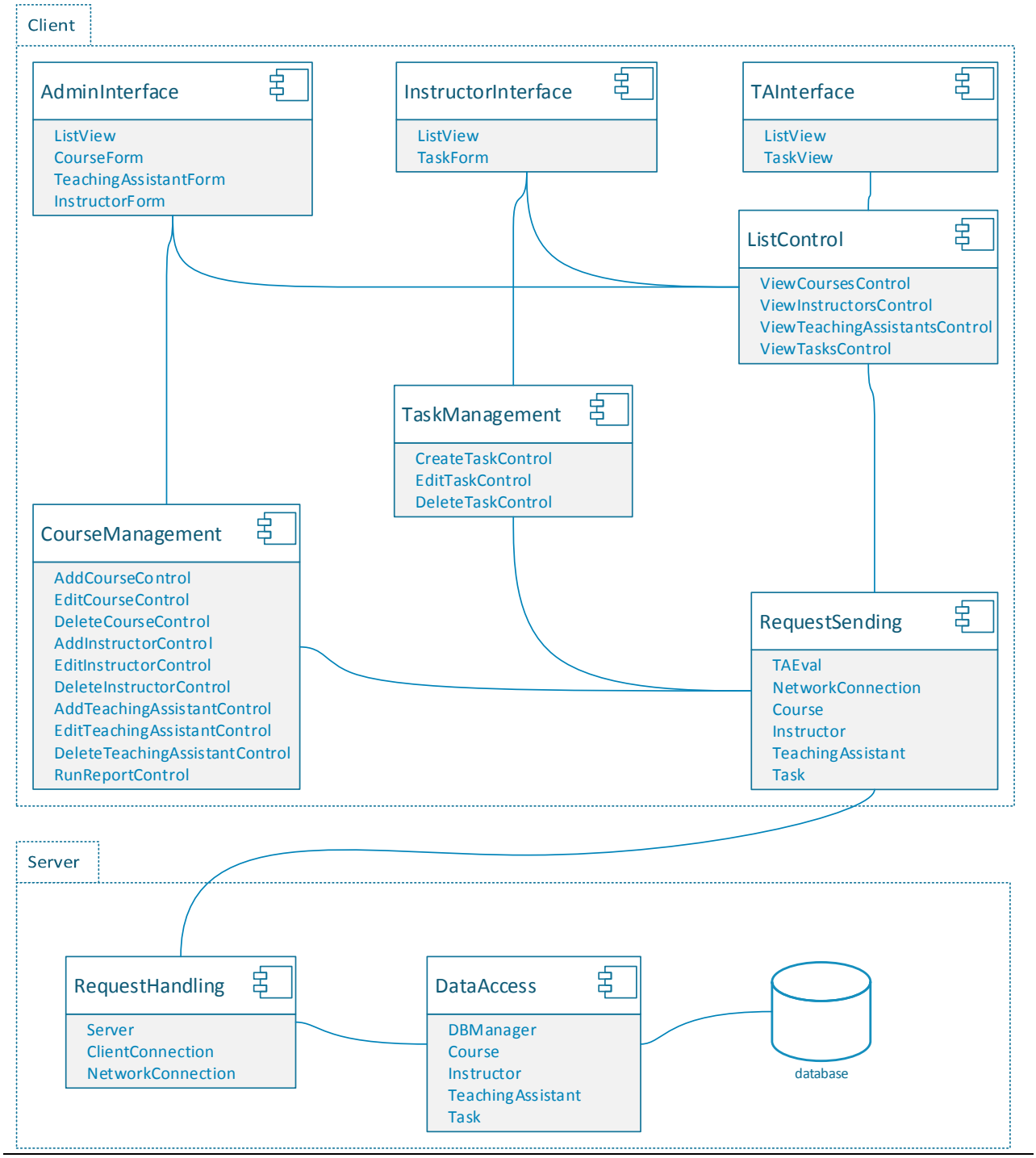


Figure 5: High-level System Decomposition

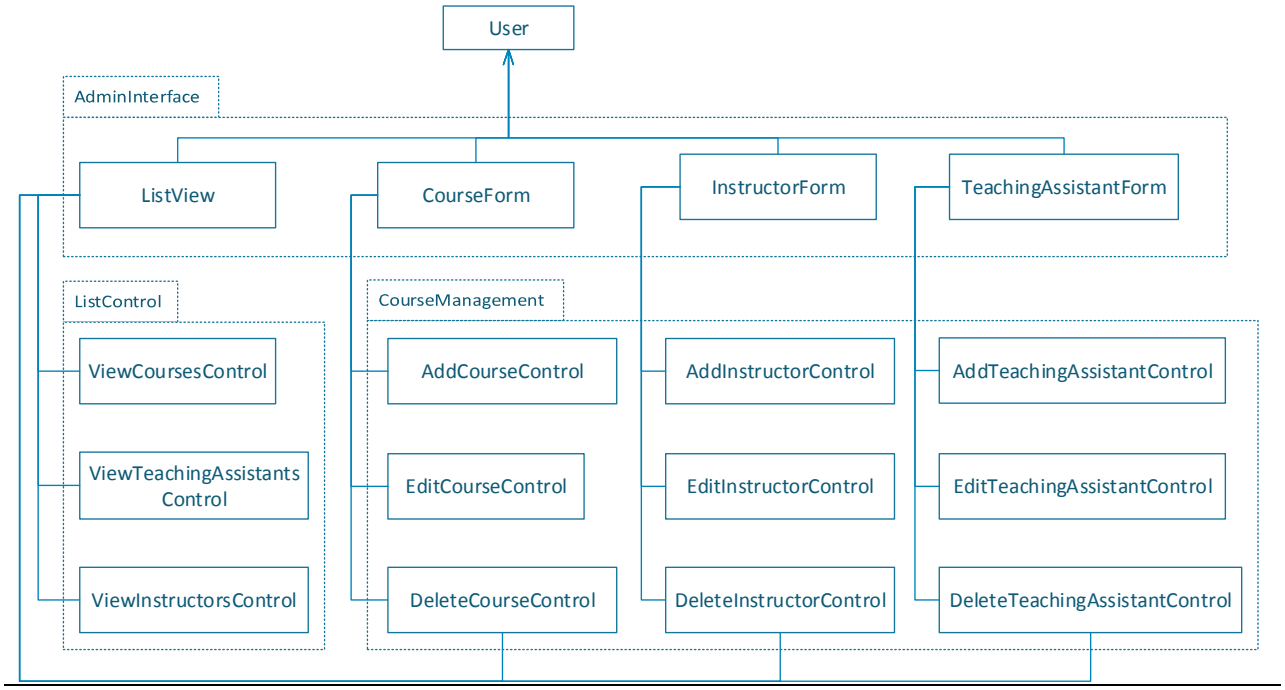


Figure 6: Administrator Detailed System Decomposition Part: A

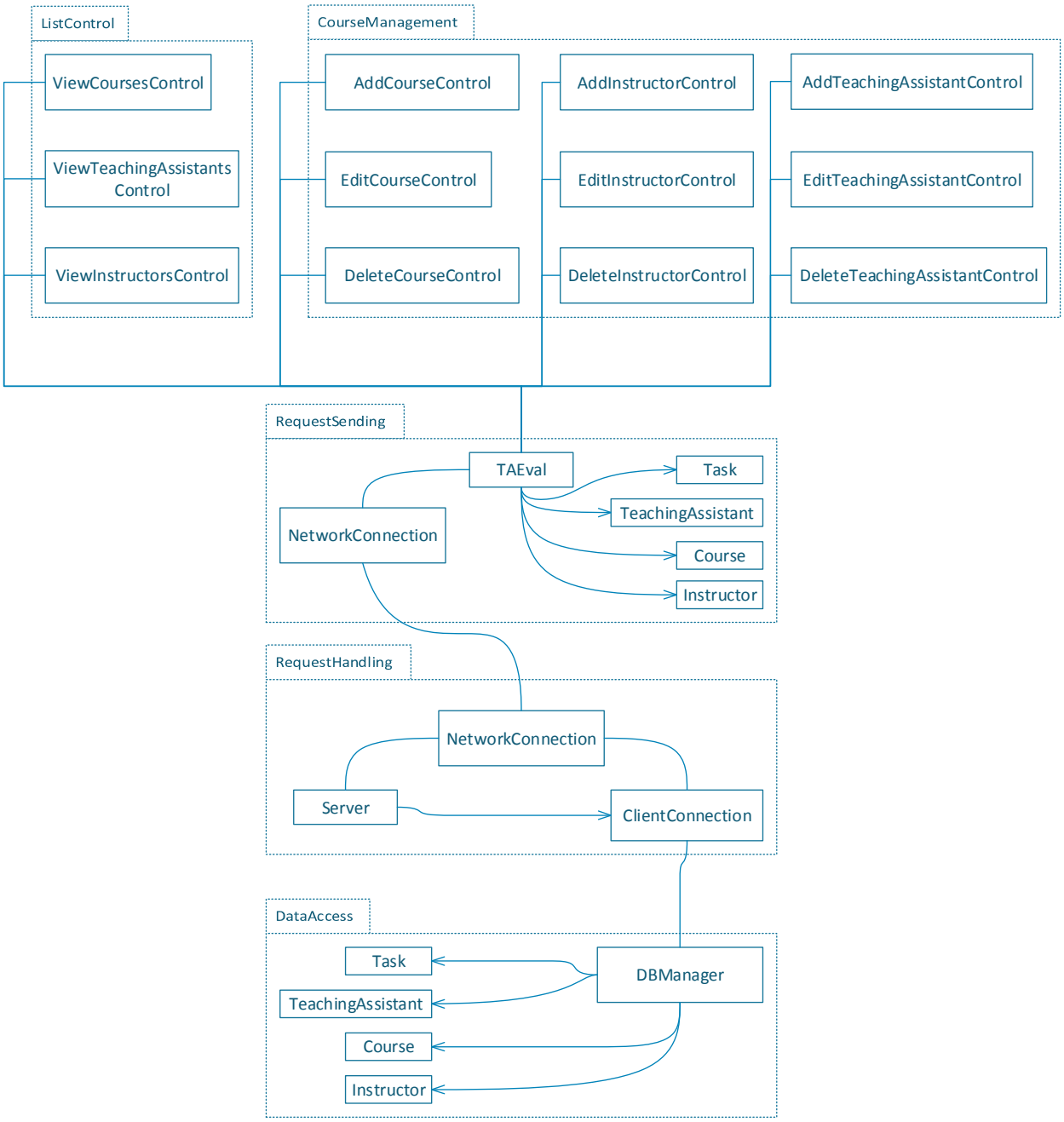


Figure 7: Administrator Detailed System Decomposition Part: B

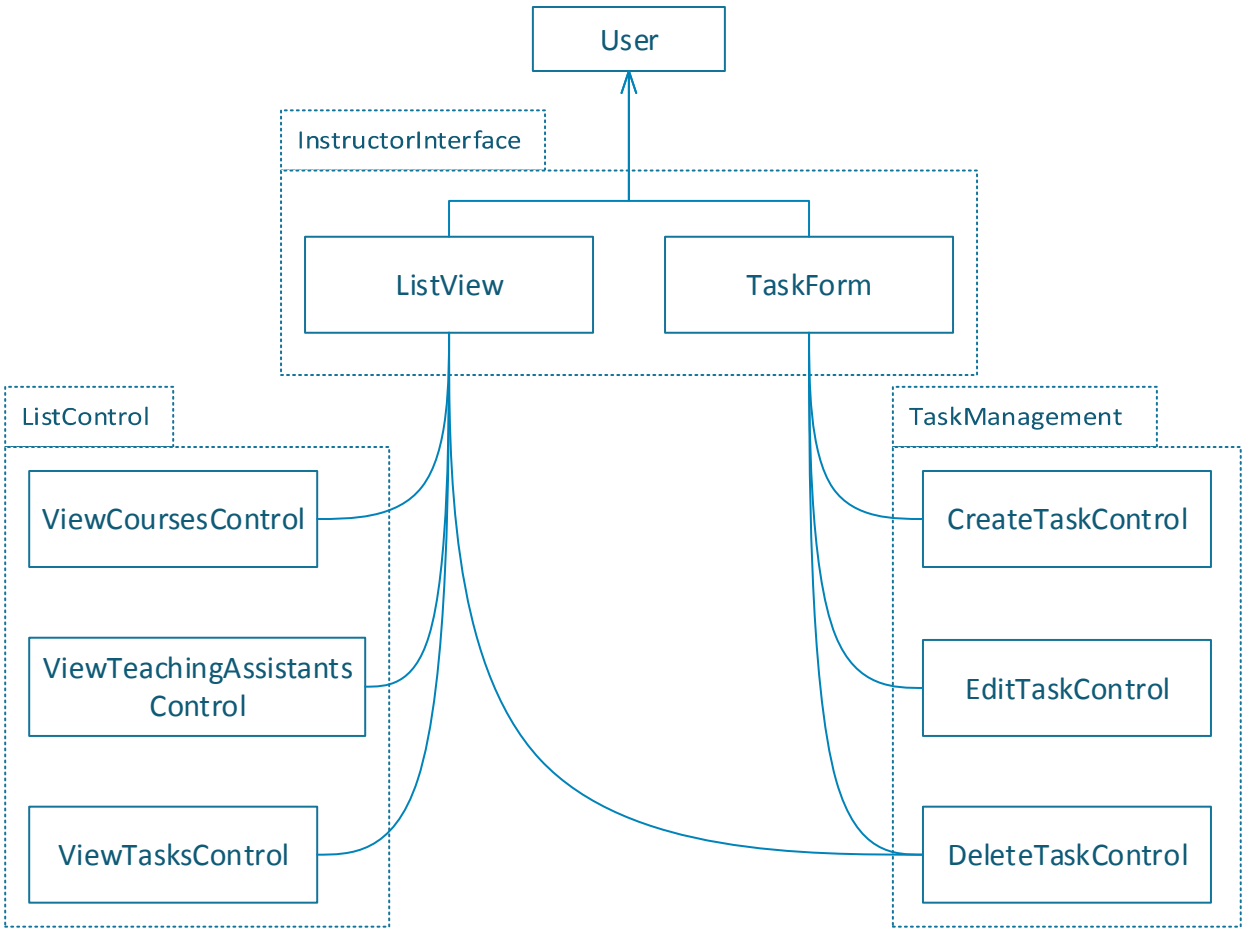


Figure 8: Instructor Detailed System Decomposition Part: A

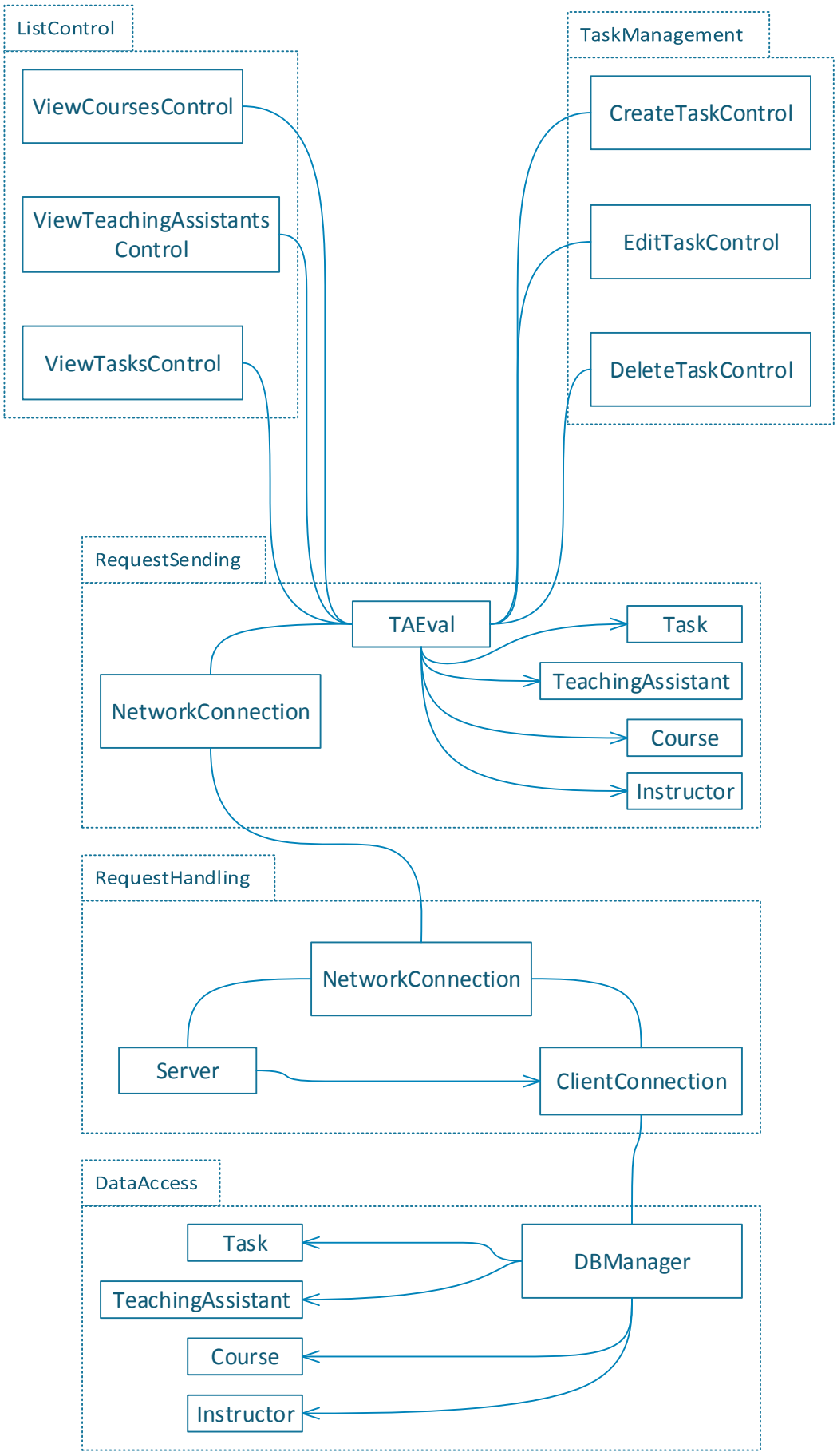


Figure 9: Instructor Detailed System Decomposition Part: B

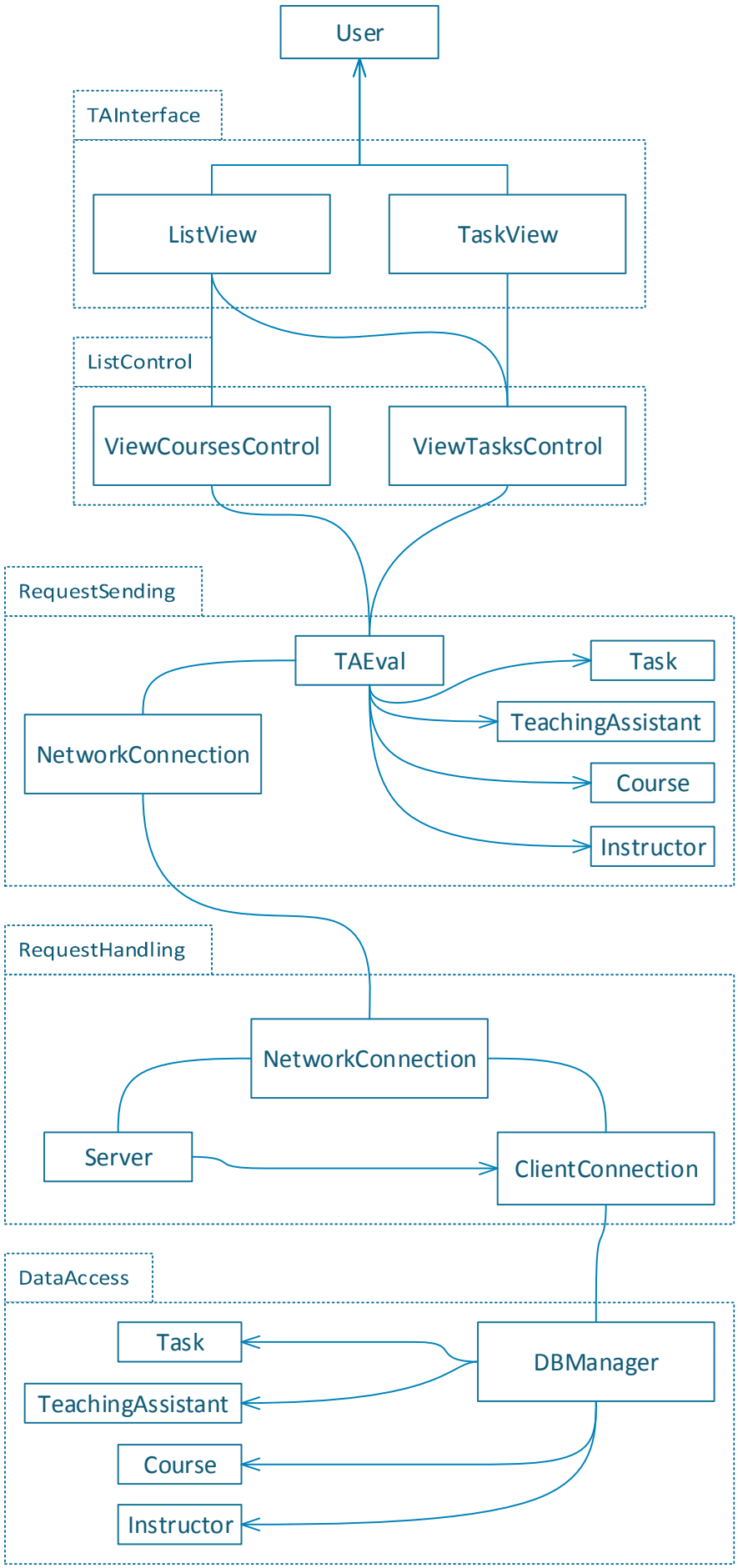


Figure 10: Teaching Assistant Detailed System Decomposition

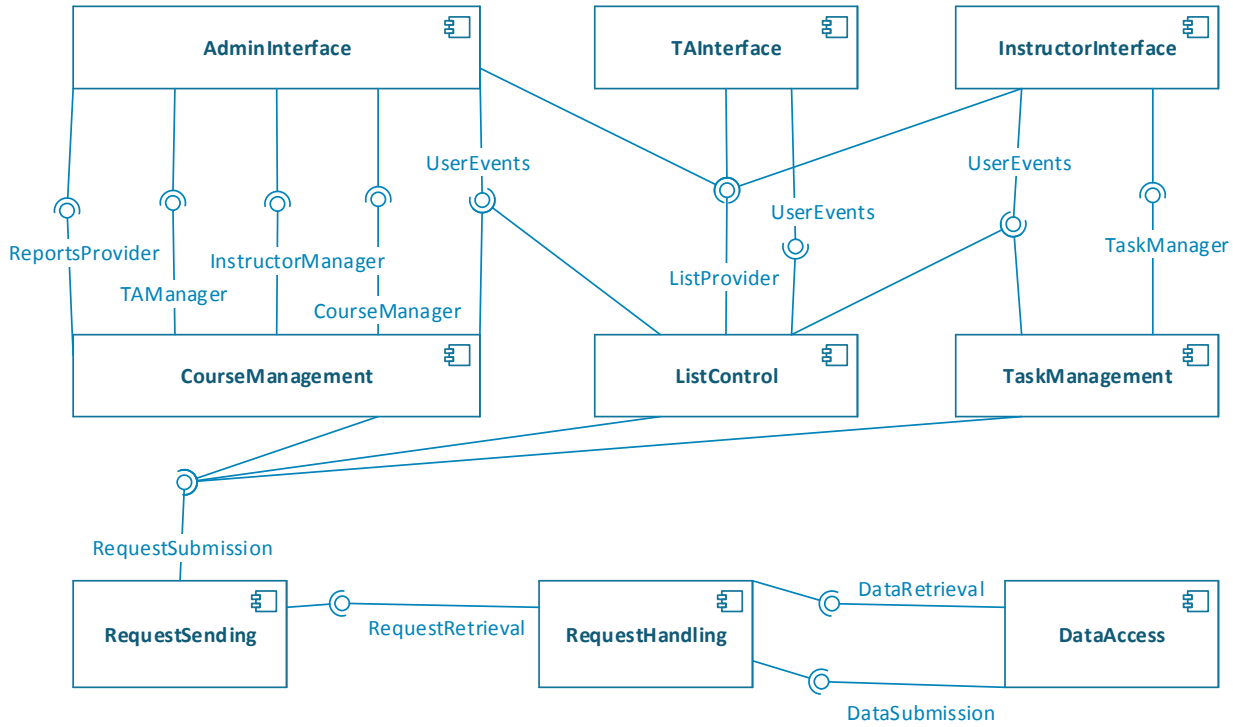


Figure 11: Teaching Assistant Detailed System Decomposition

2.2 Subsystem Description

Subsystem	**AdminInterface**
Description	The *AdminInterface* subsystem contains all of the user interface classes responsible for fulfilling an admin's functional requirements. It provides a user event service to the *CourseManagement* and *ListControl* subsystems. When an admin initiates an event this subsystem makes use of one of the manager or provider services from either the *CourseManagement* or *ListControl* subsystems depending on the type of event.
Traceability code	SB-01

Subsystem	**InstructorInterface**
Description	The *InstructorInterface* subsystem contains all of the user interface classes responsible for fulfilling an instructor's functional requirements. It provides a user event service to the *TaskManagement* and *ListControl* subsystems. When an instructor initiates an event this subsystem uses the manager service from the *TaskManagement* subsystem or the list provider service from the *ListControl* subsystem depending on the event.
Traceability code	SB-02

Subsystem	**TAInterface**
Description	The *TAInterface* subsystem contains all of the user interface classes responsible for fulfilling a teaching assistant's functional requirements. *TAInterface* provides a user event service to the *ListControl* subsystem and uses its lists provider service.

Traceability code	SB-03

Subsystem	**ListControl**
Description	The ListControl subsystem contains the classes responsible for requesting and organizing various lists on demand from the user. *ListControl* provides the list provider service to the *AdminInterface*, *InstructorInterface* and *TAInterface* and makes use of the request submission service provided by the *RequestSending* subsystem.
Traceability code	SB-04

Subsystem	**TaskManagement**
Description	The *TaskManagement* subsystem contains the classes which allow an instructor to make a request to create, modify and delete tasks. It accomplishes this by providing an event-driven management service to the *InstructorInterface* subsystem. *TaskManagement* then makes use of the request submission service provided by the *RequestSending* subsystem.
Traceability code	SB-05

Subsystem	**CourseManagement**
Description	The *CourseManagement* subsystem contains the classes responsible for allowing an administrator to request to add, edit and delete courses, instructors and teaching assistants and run various reports. It provides the reports provider service in addition to courses, instructor and TA management services to the *AdminInterface* subsystem. *CourseManagement* makes use of the request submission service provided by the *RequestSending* subsystem in order to pass along user requests.
Traceability code	SB-06

Subsystem	**RequestSending**
Description	The *RequestSending* subsystem contains the classes responsible for turning a user request into a data packet to be received by the storage server as well as initiating the client-server connection. It provides the request submission service to the *ListControl*, *TaskManagement* and *CourseManagement* subsystems and makes use of the request retrieval service provided by the *RequestHandling* subsystem.
Traceability code	SB-07

Subsystem	**RequestHandling**
Description	The *RequestHandling* subsystem contains the classes responsible for receiving a user request over a network and sending a response in return. It provides the request retrieval service to the *RequestSending* subsystem. The request retrieval service processes a user request and passes it along to the *DataAccess*

	subsystem via its data retrieval and submission services.
Traceability code	SB-08

Subsystem	**DataAccess**
Description	The *DataAccess* subsystem contains the classes responsible for creating, modifying or retrieving persistent data in the database. It provides the *RequestHandling* subsystem with the data retrieval and submission services.
Traceability code	SB-09

Subsystem	**Client**
Description	The *Client* is a physical subsystem that encompasses all of the functionality available to an end user on their machine. The *Client* subsystem is comprised of all of the interface subsystems as well as the *CourseManagement*, *TaskManagement*, *ListControl*, and *RequestSending* subsystems.
Traceability code	SB-10

Subsystem	**Server**
Description	The *Server* is a physical subsystem that encompasses all of the functionality required for retrieving or managing the program's persistent data and provides it as a service to the *Client* subsystem. The *Server* subsystem is comprised of the *RequestHandling* and *DataAccess* subsystems.
Traceability code	SB-11

2.3 Design Evolution

As the development of our system progresses, it is necessary for our original design to evolve to accommodate the addition of new features. However, due to careful planning, our original design has held up well, allowing new features to be seamlessly integrated into our existing system without having to make any significant changes to our current infrastructure.

In the first phase we had a single user interface (UI) subsystem that contained the UI for a teaching assistant (TA) user. The most significant design change reflected in the system decomposition is the addition of a new subsystem for both the instructor and admin UIs. With this change we also had to add a new controller class named ViewInstructorsControl to the *ListControl* subsystem and a new subsystem, *CourseManagement*, for handling the controller classes associated with an admin user's functionality.

In the first phase we designed and implemented our system to be scalable to any number of clients, limited only by hardware. As a result, no design changes are required to reach the goal of handling four clients concurrently. We also worked to keep our user interface decoupled from the application logic by using the observer pattern so we could easily add new interfaces without any changes to the overall design.

The only difference in design pattern usage in our new design is the addition of the Mediator pattern. In the first phase of the project the TA UI was simply a set of buttons used for running test cases. This basic interface did not require any sort of complex interaction between UI elements. In phase two we are going to be replacing the TA interface with a brand new interface and we will be adding two new interfaces for the instructor and admin users. Instead of providing a simple view for running test cases, these new interfaces will allow all three user types to fully interact with TAEval to access all of its functionality. As a result, we will use the mediator pattern in each of the three UI subsystems for mediating the behaviour of the UI elements. The details on our usage of the Mediator pattern will be further discussed in the section on design patterns.

3. Design Strategies

3.1 Hardware/Software Mapping

The TAEval system uses the Client/Server architectural style. The TAEval client receives inputs from the users via the Administrator, TA and Instructor interfaces and send their requests to a server via TCP/IP connection. Once a connection has been made to the server, a transactional request is sent to a central database. This type of Client/Server setup is considered a special case of the repository architectural style.

The TAEval system is based on the Client/Server architectural style due to the following non-functional requirements:

1. client processes must execute on a different machine than the server and support a single user.
2. client processes must communicate with the central server using TCP/IP sockets.
3. no client processes with execute on the server host.
4. server processes must execute on a central host and must manage updates and retrievals of data.

The TAEval system is broken down into two components: client and server, as per the Client/Server architectural style. There are 7 subsystems with in the client component and 2 subsystems within the server component.

The following are the user interface subsystems within the client component: *TAInterface*, *InstructorInterface* and *AdminInterface*. These interfaces allow the user to interact with the TAEval system.

The following are the control subsystems within the client component: *CourseManagement*, *TaskManagement* and *ListControl*. These subsystems interact with the user interface subsystems and the *RequestSending* subsystem. The control subsystems provide the controls to request entity management actions such as; adding a course, deleting a task and adding an instructor.

The *RequestSending* subsystem communicates between the two components: client and server via TCP/IP sockets. This subsystem communicates with the *RequestHandling* subsystem in the server component.

The *RequestHandling* subsystem handles the requests coming from the client component and requests database transactions to the *DataAccess* subsystem.

The *DataAccess* subsystem on the server provides the interface to the relational database.

The client/server components are compiled into two separate executables: TAEvalClient and TAEvalServer. The TAEvalClient executes as a client on an Ubuntu Linux user machine (node). The TAEvalServer executes on an Ubuntu Linux server machine (node).

3.2 Persistent Data Management

The TAEval system required some entities such as TA, Course, Instructor and Task to out-live a single execution of the system. The following were non-functional requirements that needed to be taken in consideration in order to make a decision on the type of persistent data management system that was necessary:

1. If the TAEval system crashes, it must be able to return to the last saved state.
2. There should not be any duplicate data anywhere in the system.
3. TAEval must support a minimum of four concurrent processes.
4. Data storage organization must be designed for ease of use, retrieval and efficient use of storage space.
5. Queries to the server must return only the minimum about of necessary data.
6. All data must be stored centrally on a single host.

The TAEval system stores the persistent data in a SQLite relational database for the following reasons:

- Relational database provides concurrency management, so it can be accessed by multiple clients. The database handles multiple processes accessing the same relational tuple or relational table.
- Allows data to be queried easily using SQL queries and commands, such as listing courses by term and year, getting task by TA student number and course ID, and getting task by task ID.
- The relational database allows for crash recovery and access control.
- The entity objects that need to be persist in the TAEval system map onto relational tables.
- Properly created relational database schemas will not allow for duplicate data, by using unique primary keys and foreign keys within the relational tables.

How does the TAEval database minimize duplication?

- Creating relational tables for many-to-many relationships between entities. The Entity-Relation diagram shows a many-to-many relationship between courses and TAs a relational table was needed to connect both entities without duplicating data in both tables.
- The employee number foreign key in the course table allows the table to reference the instructor who instructs the course. Without the foreign key, there would be an instructor tuple created for each course in the instructor table.
- The student number foreign key and course id foreign key in the task table allows each task to reference an existing student and course it is associated with. If the foreign key did not exist, we would have to create a relational tuple in the course table for each of the tasks for the course and the same would be true in the TA table. The TA table would have to create a relational tuple for each task.
- On the creation of an instructor, the TAEval system uses the DBManager class to look for duplicate first name, last name entries before inserting a new instructor. An auto generated employee number primary key is created for each instructor to allow for uniqueness of each instructor.

- On the creation of a TA, the TAEval system in the DBManager class looks for duplicate first name, last name entries before inserting a new TA. An auto generated student number primary key is created for each TA to allow for uniqueness of each TA.
- On the creation of a task, the TAEval system in the DBManager class creates an auto generated course id primary key and a course id and student number foreign key is associated with the task.

Objects in the Relation database:

Rational Tables	**Purpose**	**Attributes**
TA	The TA table contains personal information about each TA within the TAEval system	Student Number (auto-generated primary key), First Name, Last Name, Degree, Major and Year
INSTRUCTOR	The INSTRUCTOR table contains information about each instructor within the TAEval system	Employee Number (auto-generated primary key), First Name, Last Name and Department
TASK	The TASK table contains information about the task created by an instructor for a specific TA	Task ID (auto-generated primary key), Task Name, Evaluation Description, Evaluation Rank, Student Number of the student that the task is assigned to and Course ID with which the task is associated to.
COURSE	The COURSE table contains general information about the course	Course ID (auto-generated primary key), Course Name, Year the course is given, Term the course is given and the Employee number of the Instructor teaching the course.
TACOURSE relationship	The TACOURSE table is the relationship between the course entity and TA entity. This table is created because there is a many-to-many relationship between the two entities and the table will reduce repeat values in each table that may cause false information if both are not kept up to date.	Student Number (foreign key from the TA table) and Course ID (foreign key from the COURSE table). Both these attributes are primary keys.

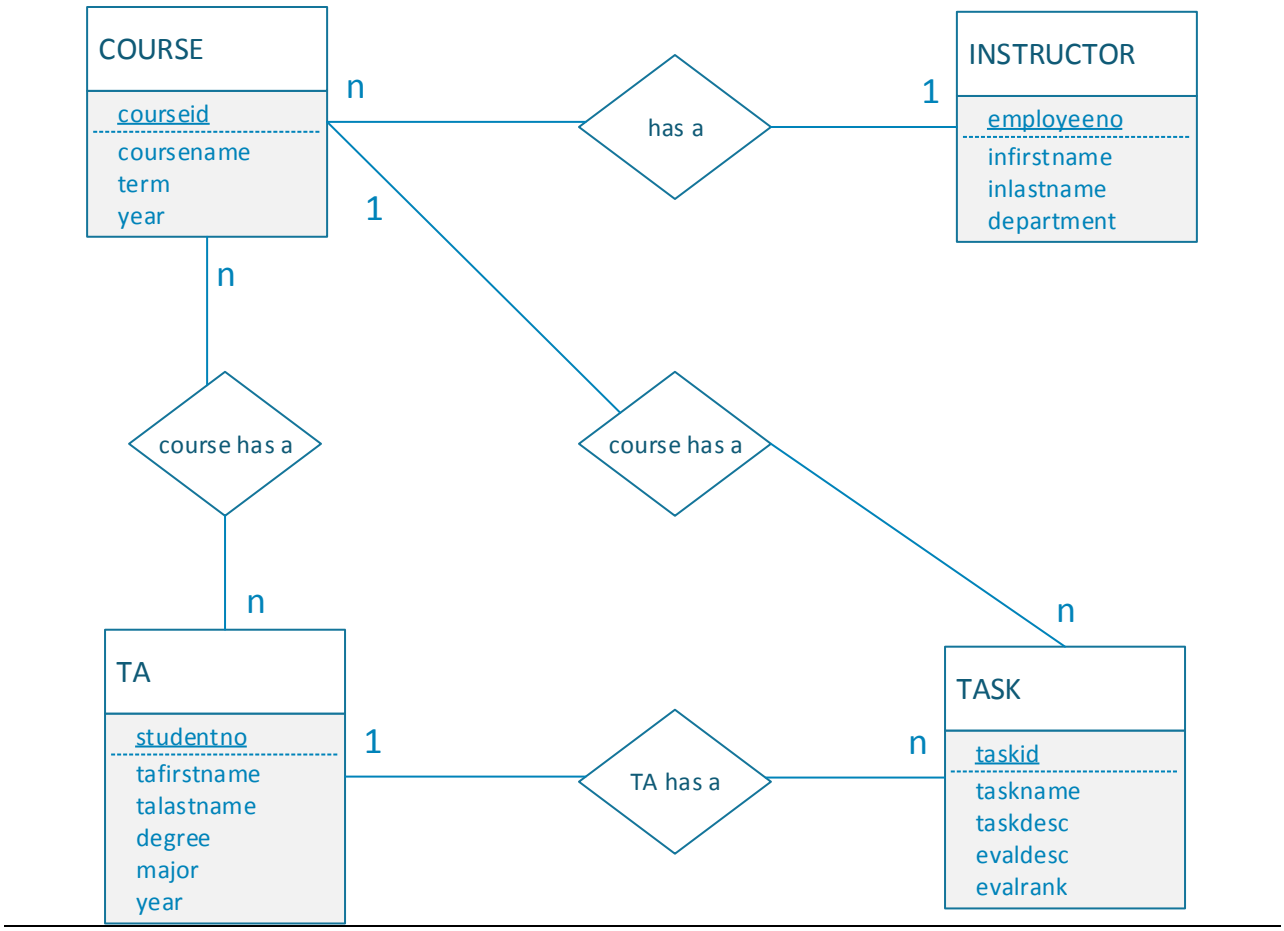


Figure 12: Entity-Relationship Diagram for TAEval Database

3.3 Design Patterns

STRUCTURAL PATTERNS

Façade

"Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use."

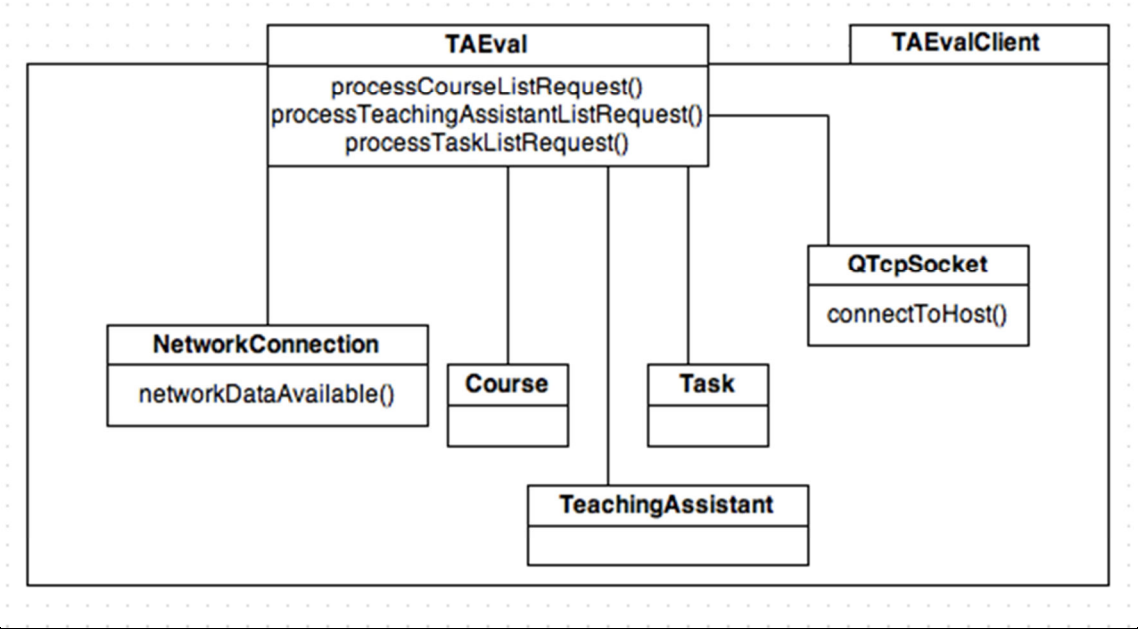


Figure 13: Façade UML Diagram

The façade design pattern was used for our class named TAEval. We wanted a single, simplified interface for all of our controllers to communicate with to perform the general tasks of sending requests from the client side to the server.

Our main window that is displayed to the user only needs a pointer to an instance of TAEval. From the main window any events generated by the user via UI elements flow from controllers to TAEval which processes the packet of data to make the appropriate call to the server. By doing this, it glues together the classes and encapsulates the functionality of any possible request the user could want to make into a convenient interface (such calls as requestCourseList, or processCourseListRequest for example).

The façade, allowing for TAEval to be the gateway of requests between the client and server, promotes decoupling by reducing the dependencies that would otherwise exist between the many different controller classes from the client side and the many different controller classes from the server. This promotes subsystem independence and portability (for instance, in

the event that we wanted to switch from sending packets of data using our NetworkConnection class to another method of communicating data, we would only need to change TAEval instead of the large number of controller classes we have for our application logic).
As the gateway of requests, it is the only entry point between the client and the server. It is desirable that, if we have dependent subsystems, to have them communicate through as minimum of a number of classes as possible.

There are inherent disadvantages to the benefits that this façade class brings; namely, it increases the number of possibilities for bugs, as changes to NetworkConnection or any of the model classes could directly break TAEval. In other words, it adds another level of complexity to our code.

BEHAVIOURAL PATTERNS

Observer:

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

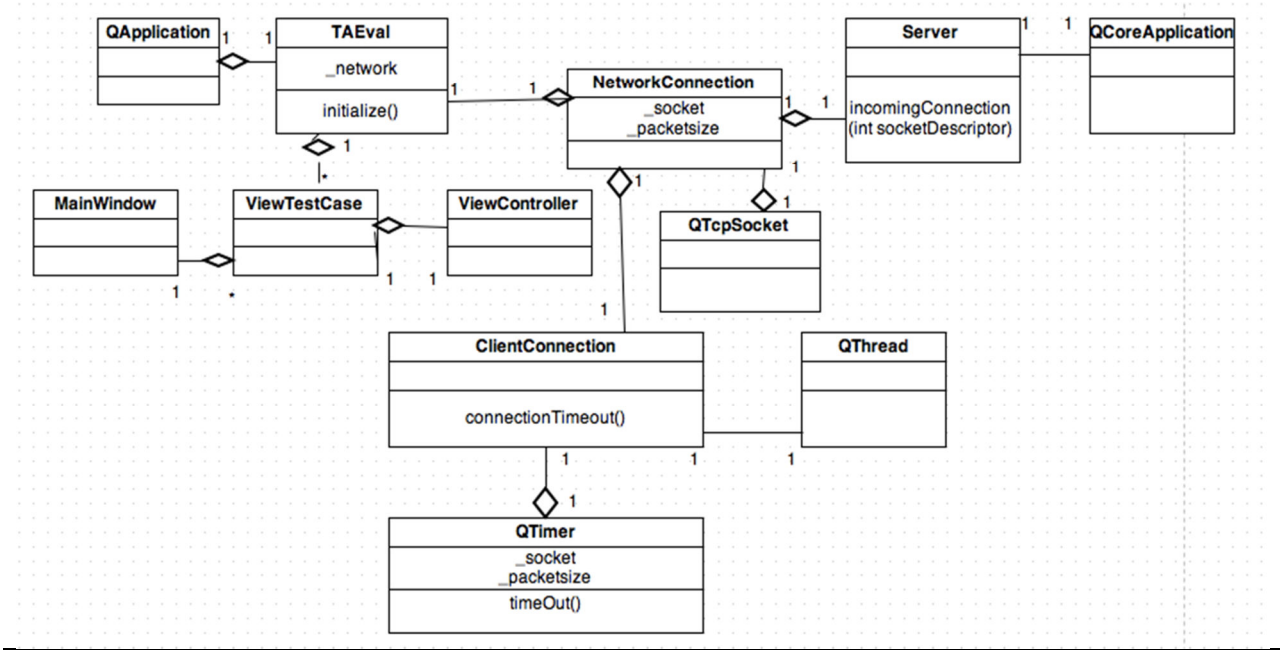


Figure 14: Observer UML Diagram

The observer design pattern was used throughout many classes via QT's signals and slots. In many cases, if a change occurs to one class it follows and requires that some form of change to occur in one or more classes. For example, a request made through the UI from an instructor user to view his courses requires many instantaneous changes that result in the instructor being able to view his requested list. Similarly, a deletion on the server side must be reflected to the client's current state. This bidirectional behavior of observers watching subjects is crucial in our application.

Signals may have one-to-many slots, where signals are emitted for slots to receive (for example; in TAEval, the instance's embedded NetworkConnection class emits a processPacket() signal when network data is available and ready to be sent. That signal was connected to TAEval's own processPacket() function, which then interpreted the packet immediately upon its arrival to call the appropriate function.

The use of signals and slots allows for great flexibility and simplicity. Any class may be an observer to one or many subjects, or it may be a subject to one or many observers, or, possibly both. The simplicity lays in the fact that a subject can emit its notifications without having to have knowledge of who its observers may be. Any observer who is interested in the specific notification can subscribe to receive them and handle the notification in their own way. This removes the unnecessary overhead of subject needing to store a dynamic list of interested observers to loop through calling update() for each.

An example of the benefits can be seen again with the processPacket() example; we don't know how many clients may be logged in on one server, but any update in state to the server can occur with the appropriate adjustments to the client's state without any awareness of the number of client connections. Because of this abstract connection, between classes that share the relationship of requiring updates when changes to state happen, the coupling between subjects and observers is minimal.

Another advantage to the abstract coupling is that we could re-use the entire span of subjects easier without re-using the observers, if we ran into the scenario of wanting the same signals emitted but different handling of the signals in new slots.
The disadvantages to the benefits are that, due to the flexibility of any class being eligible to be either a subject, observer, or both, it can be hard to see control flow from start point to end point as it spans many files.

Also, the lack of awareness that a subject has to its interested observers leads to an implicit control flow; we can only know the chain of events is correct at runtime or using a debugger. Though we tie a specific signal to a specific slot, it is possible for unexpected or undesired updates to happen without much verbose explanation as to what exactly got updated.

Finally, if either the observer or subject were deleted, we would need to remember to delete all links between them and update them accordingly for new implementations. An example of taking precaution with this can be seen in Main for the client side, as we make sure the TAEval instance is deleted after the application emits the signal that it's about to quit.

Proxy:

"Provide a surrogate or placeholder for another object to control access to it"

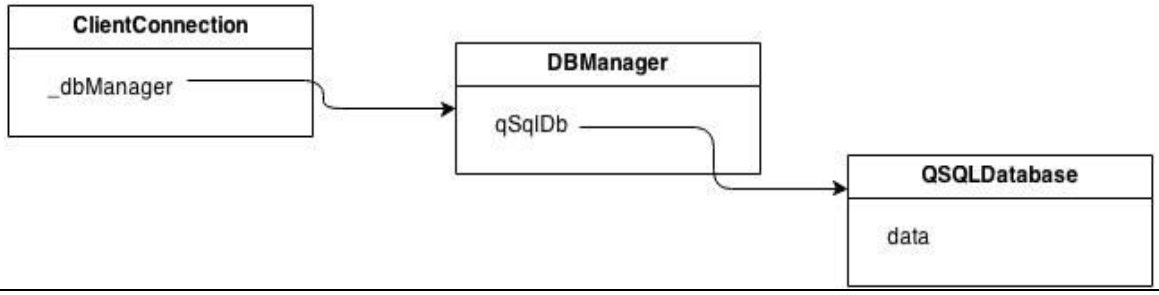


Figure 15: Proxy UML Diagram

The proxy pattern was used on the server side on the class DBManager. We want to control access to our SQL database with respect to its creation and initialization until we actually need to use it. Once the system is in production, the database could be potentially very large. For this reason, we want to maximize as much control as possible and defer creating queries and loading documents until absolutely necessary. It isn't necessary to create each of our many tables until we need them, and those table creations are in complete control in DBManager.

In this way, DBManager is a stand in for the actual database. It acts just like the database and takes care of instantiating it when required. Any requests made on demand are forwarded by the proxy directly to the database by keeping a reference to it.

Disadvantages include adding another layer of complexity to the code, which could introduce bugs.

Iterator:

"Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation"

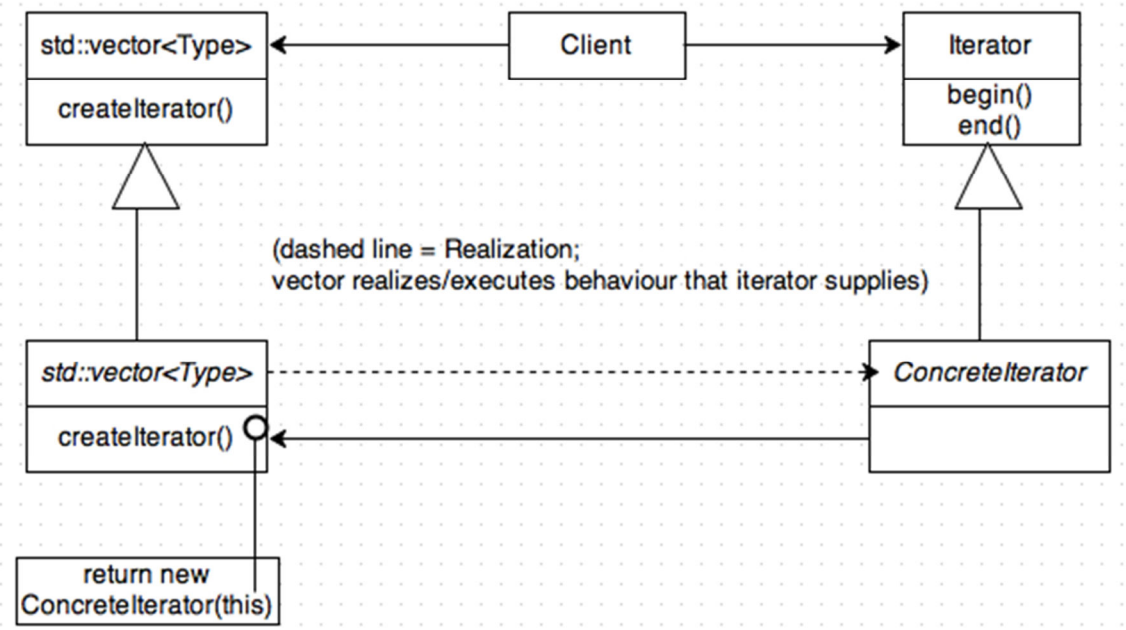


Figure 16: Iterator UML Diagram

The iterator pattern was used on the server side in both DBManager and ClientConnection when the client requests lists of instances, such as Courses or Tasks. With multiple clients being able to connect to a single server, it's desirable to have multiple traversals occurring concurrently on a single collection of objects, where each iterator keeps track of its own traversal state. As well, as the size of the persistent collections of data increases directly proportional to the application's usage, we want to be able to iterate over the collection without exposing its internal structure and loading every single object into RAM.

Though not implemented for Deliverable 2, the usage of iterators allows for supportability in the future for more complex traversals that are independent of the implementation of the collections. Such complex traversals could include filtering or sorting algorithms. Because of the independence with the actual collection itself, it allows for the role of the vector to be simplified as a data structure solely responsible for holding our model instances.

There are some disadvantages that come with the behavior of the iterator. It is possible to have an iterator on an unordered collection such as a set, and so if a programmer observes the elements of the set being returned as interpreted as having an order to them, future code relying on this assumption could raise problems, as the iterator would provide the elements in an arbitrary order in this case. Issues may also arise during multithreading if the collection doesn't update the iterator about its change of size.

Mediator:

"Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently."

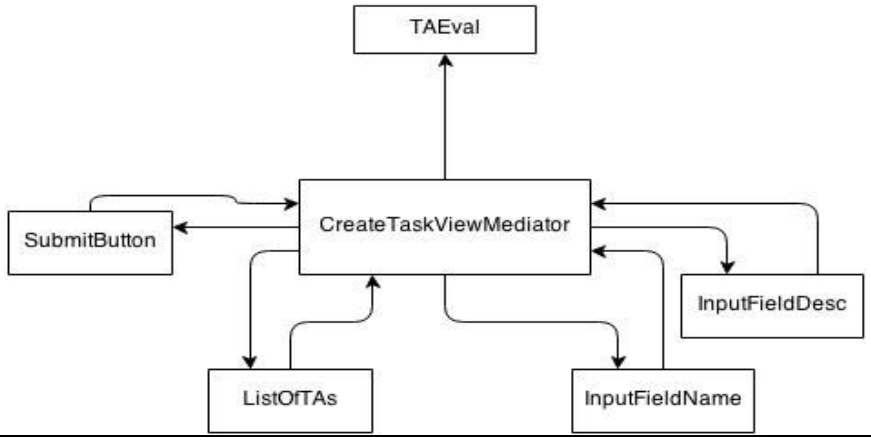


Figure 17: Mediator UML Diagram

This is a design pattern we would like in the ideal complete implementation of TAEval. This would be used in a control object that encapsulates the behavior of the set of all controllers that listen to a specific set of responses from the client interface.

Since the application is graphical in nature to the user, we are going to run into scenarios where, ideally, certain UI elements behave dependently upon the state of other UI elements on the same window or frame. An instructor user may commonly create tasks, and an example of such dependent behavior may be graying out the Submit button, and rendering it not clickable, until the necessary fields of information are filled out about the task.

It's undesirable to have the UI objects refer to and depend on each other in this form example. The mediator object assumes the responsibility of controlling and coordinating the exact dependencies between the UI elements, which promotes decoupling. In this way, each of the dependent objects would only need to refer to the mediator, reducing varying many-to-many relationships to as many one-to-many relationships as there are dependent classes. It's also cleaner to read when the interaction behavior is separated from each individual class and encapsulated into one sole class responsible for it.

The potential downfall of that is that the mediator class can become too convoluted and complex.

Memento:

"Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later."

This is another design pattern we would like in the ideal complete implementation of TAEval. Often, users with the ability to modify information by editing or deleting may want to undo undesired operations. An example could be an instructor who may accidentally delete an unresolved task, or, even worse, an administrator who deletes an unfinished course under the assumption that it's safe to do so. In these scenarios, we would want to revert back to the original state.

For those scenarios, we can implement the memento pattern to preserve the state of the object to be edited or deleted without exposing information private to the original class. The memento is an object of the original class with the state before modification. A controller class responsible for managing undo requests will request the memento upon initialization of a modification scenario, and return the memento object if it is necessary (ie: when the user requests to undo).

The downsides are that it might be a computational disadvantage to store each memento instance if it is expensive to do so. As well, there is a cost associated with the proper implementation of the class that will be responsible for deleting the mementos once they are no longer necessary.

4. Subsystem Services

Service Name	**AdminInput**
Subsystem Traceability	SB-01
Service Description	Users with admin privileges interact with *AdminInterface*. *CourseManagement* and *ListControl* are provided a user event service of knowing what and when admin events occur.
Traceability Code	SE-01

Service Name	**InstructorInput**
Subsystem Traceability	SB-02
Service Description	Users with instructor privileges interact with *InstructorInterface.* *TaskManagement* and *ListControl* are provided a user event service of knowing what and when instructor events occur.
Traceability Code	SE-02

Service Name	**TAInput**
Subsystem Traceability	SB-03
Service Description	Users with TA privileges interact with *TAInterface*. *ListControl* is provided a user event service of knowing what and when TA events occur.
Traceability Code	SE-03

Service Name	**CoursesManager**
Subsystem Traceability	SB-06
Service Description	Allows user with administrator privileges to add, delete, or edit courses from the user interface.
Traceability Code	SE-04

Service Name	**InstructorManager**
Subsystem Traceability	SB-06
Service Description	Allows user with administrator privileges to add, delete, or edit instructors from the user interface.
Traceability Code	SE-05

Service Name	**TAManager**
Subsystem Traceability	SB-06
Service Description	Allows user with administrator privileges to add, delete, or edit TAs from the user interface.
Traceability Code	SE-06

Service Name	**TaskManager**
Subsystem Traceability	SB-05
Service Description	Allows user with instructor privileges to add, delete, or edit tasks and task evaluations from the user interface.
Traceability Code	SE-07

Service Name	**ReportsProvider**
Subsystem Traceability	SB-06
Service Description	Allows user with administrator privileges to run reports on TA evaluation data from the user interface.
Traceability Code	SE-08

Service Name	**ListsProvider**
Subsystem Traceability	SB-04
Service Description	Allows user to view list of requested information, such as courses, instructors, TAs, or tasks.
Traceability Code	SE-09

Service Name	**RequestsSubmission**
Subsystem Traceability	SB-07
Service Description	Processes requests from the CourseManagement, ListControl, and TaskManagement subsystems, formats the data, and submits the request over to the server.
Traceability Code	SE-10

Service Name	**RequestsRetrieval**
Subsystem Traceability	SB-08
Service Description	Processes requests from the client, formats the data, and submits the data in the form of a request to DataAccess.
Traceability Code	SE-11

Service Name	**DataSubmission**
Subsystem Traceability	SB-09
Service Description	Gets parameters from RequestRetrieval, formats the data into a query, and completes requests that create or modify data in the database.
Traceability Code	SE-12

Service Name	**DataRetrieval**
Subsystem Traceability	SB-09
Service Description	Gets parameters from RequestRetrieval, formats the data into a query, and completes requests that get data from the database.
Traceability Code	SE-13

Table of operations within AdminInput

Operation	**Description**
void AdminUI::createCourseButtonClicked()	Initializes *CreateCourseControl* and invokes its sole function with the data on the form as parameters.
void AdminUI::createInstructorButtonClicked()	Initializes *CreateInstructorControl* and invokes its sole function with the data on the form as parameters.

void AdminUI:: createTeachingAssistantButtonClicked()	Initializes *CreateTeachingAssistantControl* and invokes its sole function with the data on the form as parameters.
void AdminUI::editCourseButtonClicked()	Initializes *EditCourseControl* and invokes its sole function with the data on the form as parameters.
void AdminUI::editInstructorButtonClicked()	Initializes *EditInstructorControl* and invokes its sole function with the data on the form as parameters.
void AdminUI:: editTeachingAssistantButtonClicked()	Initializes *EditTeachingAssistantControl* and invokes its sole function with the data on the form as parameters.
void AdminUI::deleteCourseButtonClicked()	Initializes *DeleteCourseControl* and invokes its sole function with the Course selected as its parameter.
void AdminUI::deleteInstructorButtonClicked()	Initializes *DeleteInstructorControl* and invokes its sole function with the Instructor selected as its parameter.
void AdminUI:: deleteTeachingAssistantButtonClicked()	Initializes *DeleteTeachingAssistantControl* and invokes its sole function with the TeachingAssistant selected as its parameter.
void AdminUI::reportEvaluationsForTAClicked()	Initializes *RunReportsControl* and invokes **RunReportsControl::viewEvaluationsForTA** with the selected TA as its parameter.
void AdminUI::reportEvaluationForTermClicked()	Initializes *RunReportsControl* and invokes **RunReportsControl::viewEvaluationsForTerm** with the specified term as its parameter.
void AdminUI::reportEvaluationForCourseClicked()	Initializes *RunReportsControl* and invokes **RunReportsControl::viewEvaluationsForCourse** with the selected Course as its parameter.
void AdminUI:: viewCoursesButtonClicked()	Initializes *ViewCoursesControl* and invokes its sole function with the data on the form as parameters to filter the list.
void AdminUI:: viewInstructorButtonClicked()	Initializes *ViewInstructorsControl* and invokes its sole function with the data on the form as parameters to filter the list.
void AdminUI:: viewTeachingAssistantButtonClicked()	Initializes *ViewTeachingAssistantControl* and invokes its sole function with the data on the form as parameters to filter the list.
void AdminUI:: viewTasksButtonClicked()	Initializes *ViewTasksControl* and invokes its sole function with the data on the form as parameters to filter the list.

Table of operations within InstructorInput

Operation	Description
void InstructorUI:: createTaskButtonClicked()	Initializes *CreateTaskControl* and invokes its sole function with the data on the form as parameters.
void InstructorUI::editTaskButtonClicked()	Initializes *EditTaskControl* and invokes its sole function with the data on the form as parameters.
void InstructorUI::deleteTaskButtonClicked()	Initializes *DeleteTaskControl* and invokes its sole function with the Task selected as its parameter.
void InstructorUI:: viewCoursesButtonClicked()	Initializes *ViewCoursesControl* and invokes its sole function with the data on the form as parameters to filter the list.
void InstructorUI:: viewInstructorButtonClicked()	Initializes *ViewInstructorsControl* and invokes its sole function with the data on the form as parameters to filter the list.
void InstructorUI:: viewTeachingAssistantButtonClicked()	Initializes *ViewTeachingAssistantControl* and invokes its sole function with the data on the form as parameters to filter the list.
void InstructorUI:: viewTasksButtonClicked()	Initializes *ViewTasksControl* and invokes its sole function with the data on the form as parameters to filter the list.

Table of operations within TAInput

Operation	Description
void TeachingAssistantUI:: viewCoursesButtonClicked()	Initializes *ViewCoursesControl* and invokes its sole function with the data on the form as parameters to filter the list.
void TeachingAssistantUI:: viewInstructorButtonClicked()	Initializes *ViewInstructorsControl* and invokes its sole function with the data on the form as parameters to filter the list.
void TeachingAssistantUI:: viewTeachingAssistantButtonClicked()	Initializes *ViewTeachingAssistantControl* and invokes its sole function with the data on the form as parameters to filter the list.
void TeachingAssistantUI:: viewTasksButtonClicked()	Initializes *ViewTasksControl* and invokes its sole function with the data on the form as parameters to filter the list.

Table of operations within CoursesManager

Operation	Description
void CreateCourseControl::invoke(TAEval* taEval, QString& name, QString& code, int year, QString& term)	Listens for the “Create Course” button to be clicked. If clicked, takes all data in form and uses the **TAEval::createCourse** operation to submit the request to the server.
void DeleteCourseControl::invoke(TAEval* taEval, Course& crs)	Listens for the “Delete Course” button to be clicked. If clicked, takes the selected Course from list and uses the **TAEval::deleteCourse** operation to submit the request to the server.
void EditCourseControl::invoke(TA Eval* taEval, QString& name, QString& code, int year, QString& term)	Listens for the “Edit Course” button to be clicked. If clicked, takes the selected Course from list, takes all data in form and edits the instance with the new inputs. It then uses the **TAEval::editCourse** operation to submit the request to the server.

Table of operations within InstructorManager

Operation	Description
void CreateInstructorControl:: invoke(TAEval* taEval, QString& f_name, QString& l_name, QString& dept)	Listens for the “Create Instructor” button to be clicked. If clicked, takes all data in form and uses the **TAEval::createInstructor** operation to submit the request to the server.
void DeleteInstructorControl:: invoke(TAEval* taEval, Instructor& instructor)	Listens for the “Delete Instructor” button to be clicked. If clicked, takes the selected Instructor from list and uses the **TAEval::deleteInstructor** operation to submit the request to the server.
void EditInstructorControl::inv oke(TAEval* taEval, QString& f_name, QString& l_name, QString& dept)	Listens for the “Edit Instructor” button to be clicked. If clicked, takes the selected Instructor from list, takes all data in form and edits the instance with the new inputs. It then uses the **TAEval::editInstructor** operation to submit the request to the server.

Table of operations within TAManager

Operation	Description
void CreateTeachingAssistantControl::invoke(TAEval* taEval, QString& f_name, QString& l_name, QString& degree, QString& major, int year)	Listens for the “Create Teaching Assistant” button to be clicked. If clicked, takes all data in form and uses the **TAEval::createTeachingAssistant** operation to submit the request to the server.
void DeleteTeachingAssistantControl::invoke(TAEval* taEval, TeachingAssistant& ta)	Listens for the “Delete Teaching Assistant” button to be clicked. If clicked, takes the selected TeachingAssistant from list and uses the **TAEval::deleteTeachingAssistant** operation to submit the request to the server.
void EditTeachingAssistantControl::invoke(TAEval* taEval, QString& f_name, QString& l_name, QString& degree, QString& major, int year)	Listens for the “Edit Teaching Assistant” button to be clicked. If clicked, takes the selected TeachingAssistant from list, takes all data in form and edits the instance with the new inputs. It then uses the **TAEval::editTeachingAssistant** operation to submit the request to the server.

Table of operations within TaskManager

Operation	Description
void CreateTaskControl::invoke(TAEval* taEval, Course& crs, TeachingAssistant& ta, QString& name, QString& desc)	Listens for the “Create Task” button to be clicked. If clicked, takes all data in form and uses the **TAEval::createTask** operation to submit the request to the server.
void DeleteTaskControl::invoke(TAEval* taEval, Task& task)	Listens for the “Delete Task” button to be clicked. If clicked, takes the selected Task from list and uses the **TAEval::deleteTask** operation to submit the request to the server.
void EditTaskControl::invoke(TAEval* taEval, QString& name, QString& desc, QString& comment, int rating)	Listens for the “Edit Task” button to be clicked. If clicked, takes the selected Task from list, takes all data in form and edits the instance with the new inputs. It then uses the **TAEval::editTask** operation to submit the request to the server.

Table of operations within ReportsProvider

Operation	Description
void RunReportsControl::viewEvaluationsForTA(TAEval* taEval, TeachingAssistant& ta)	Listens for the "View Evaluations for TA" button to be clicked. If clicked, takes the selected TA from the list and uses the **TAEval::viewEvaluationsForTA** operation to submit the request of information to the server.
void RunReportsControl::viewEvaluationsForTerm(TAEval* taEval, QString& term)	Listens for the "View Evaluations for Term" button to be clicked. If clicked, takes the term specified by the user and uses the **TAEval::viewEvaluationsForTerm** operation to submit the request of information to the server.
void RunReportsControl::viewEvaluationsForCourse(TAEval* taEval, Course& crs)	Listens for the "View Evaluations for Course" button to be clicked. If clicked, takes the Course selected from the list and uses the **TAEval::viewEvaluationsForCourse** operation to submit the request of information to the server.

Table of operations within ListProvider

Operation	Description
void ViewCoursesControl::invoke(TAEval* taEval, QString& term, int year)	Listens for the "View Course List" button to be clicked. If clicked, takes the term and year specified by the user and uses the **TAEval::requestCourseList** operation to submit the request of information to the server.
void ViewTeachingAssistantsControl::invoke(TAEval* taEval, Course& crs)	Listens for the "View Teaching Assistant List" button to be clicked. If clicked, takes the Course specified by the user and uses the **TAEval::requestTeachingAssistantList** operation to submit the request of information to the server.
void ViewTasksControl::invoke(TAEval* taEval, Course crs, TeachingAssistant& ta)	Listens for the "View Tasks List" button to be clicked. If clicked, takes the Course and TeachingAssistant specified by the user and uses the **TAEval::requestTaskList** operation to submit the request of information to the server.
void ViewInstructorsControl::invoke(TAEval* taEval, QStrimg& dept)	Listens for the "View Instructor List" button to be clicked. If clicked, takes the Course specified by the user and uses the **TAEval::requestInstructorList** operation to submit the request of information to the server.

Table of operations within RequestSubmission

Operation	Description
void TAEval::requestCourseList(QString& term, int year)	Invoked by **ViewCoursesControl::invoke**, it formats the query parameters into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.

void TAEval::requestTeachingAssistantList(Course& crs)	Invoked by **ViewTeachingAssistantsControl**::**invoke**, it formats the query parameter into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.
void TAEval::requestTaskList(Course& crs, TeachingAssistant ta)	Invoked by **ViewTasksControl**::**invoke**, it formats the query parameters into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.
void TAEval::requestInstructorList(Course& crs)	Invoked by **ViewInstructorsControl**::**invoke**, it formats the query parameter into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.
void TAEval::createTask(Course & crs, TeachingAssistant& ta, QString& name, QString& desc)	Invoked by **CreateTaskControl**::**invoke**, it formats the parameters desired to be attributes of the created task instance into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.
void TAEval::createCourse(QString& name, QString& code, int year, QString& term)	Invoked by **CreateCourseControl**::**invoke**, it formats the parameters desired to be attributes of the created course instance into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.
void TAEval::createInstructor(QString& f_name, QString& l_name, QString& dept)	Invoked by **CreateInstructorControl**::**invoke**, it formats the parameters desired to be attributes of the created instructor instance into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.
void TAEval::createTeachingAssistant(QString& f_name, QString& l_name, QString& degree, QString& major, int year)	Invoked by **CreateTeachingAssistantControl**::**invoke**, it formats the parameters desired to be attributes of the created TA instance into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.
void TAEval::deleteTask(Task& task)	Invoked by **DeleteTaskControl**::**invoke**, it formats the unique identifier of the task to be deleted into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.
void TAEval::deleteCourse(Course& crs)	Invoked by **DeleteCourseControl**::**invoke**, it formats the unique identifier of the course to be deleted into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.

void TAEval::deleteInstructor(Instructor& instructor)	Invoked by **DeleteInstructorControl::invoke**, it formats the unique identifier of the instructor to be deleted into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.
void TAEval::deleteTeachingAssistant(TeachingAssistant& ta)	Invoked by **DeleteTeachingAssistantControl::invoke**, it formats the unique identifier of the TA to be deleted into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.
void TAEval::editTask(Task& task)	Invoked by **EditTaskControl::invoke**, it formats the attributes of the edited task that is passed in into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.
void TAEval::editCourse(Course & crs)	Invoked by **EditCourseControl::invoke**, it formats the attributes of the edited course that is passed in into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.
void TAEval::editInstructor(Instructor& instructor)	Invoked by **EditInstructorControl::invoke**, it formats the attributes of the edited instructor that is passed in into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.
void TAEval::editTeachingAssistant(TeachingAssistant& ta)	Invoked by **EditTeachingAssistantControl::invoke**, it formats the attributes of the edited TA that is passed in into a packet, assigns the packet a mapped packetId and sends the packet to the server over the network.
void TAEval::viewEvaluationsForTA(TeachingAssistant& ta)	Invoked by **RunReportsControl::viewEvaluationsForTA,** it formats the ID of the TA into a packet, assigns the packet a mapped ID associated with this specific report and sends the packet to the server over the network.
void TAEval::viewEvaluationsForTerm(QString& term)	Invoked by **RunReportsControl::viewEvaluationsForTerm,** it formats the term into a packet, assigns the packet a mapped ID associated with this specific report and sends the packet to the server over the network.
void TAEval::viewEvaluationsForCourse(Course& crs)	Invoked by **RunReportsControl::viewEvaluationsForCourse,** it formats the ID of the Course into a packet, assigns the packet a mapped ID associated with this specific report and sends the packet to the server over the network.
void TAEval::courseListUpdated(std::vector<Course>& courseList)	Signal to let interested Interfaces know that the given list of Courses has been updated

void TAEval::teachingAssistantListUpdated(std::vector<TeachingAssistant>& taList)	Signal to let interested Interfaces know that the given list of TeachingAssistants has been updated
void TAEval::instructorListUpdated(std::vector<Instructor>& instructorList)	Signal to let interested Interfaces know that the given list of Instructors has been updated
void TAEval::taskListUpdated(std::vector<Task>& taskList)	Signal to let interested Interfaces know that the given list of Tasks has been updated
void TAEval::courseCreated(Course* crs)	Signal to let interested AdminInterfaces know that the requested Course creation was successful
void TAEval::teachingAssistantCreated(TeachingAssistant* ta)	Signal to let interested AdminInterfaces know that the requested TA creation was successful
void TAEval::instructorCreated(Instructor* instructor)	Signal to let interested AdminInterfaces know that the requested Instructor creation was successful
void TAEval::taskCreated(Task* task)	Signal to let interested InstructorInterfaces know that the requested Task creation was successful
void TAEval::courseEdited(Course* crs)	Signal to let interested AdminInterfaces know that the requested Course modification was successful
void TAEval::teachingAssistantEdited(TeachingAssistant* ta)	Signal to let interested AdminInterfaces know that the requested TA modification was successful
void TAEval::instructorEdited(Instructor* instructor)	Signal to let interested AdminInterfaces know that the requested Instructor modification was successful
void TAEval::taskEdited(Task* task)	Signal to let interested InstructorInterfaces know that the requested Task modification was successful
void TAEval::courseDeleted(bool success)	Signal to let interested AdminInterfaces know that the requested Course deletion was successful
void TAEval::teachingAssistantDeleted(bool success)	Signal to let interested AdminInterfaces know that the requested TA deletion was successful

void TAEval::instructorDeleted(bool success)	Signal to let interested AdminInterfaces know that the requested Instructor deletion was successful
void TAEval::taskDeleted(bool success)	Signal to let interested InstructorInterfaces know that the requested Task deletion was successful
void TAEval::requestTimedOut()	Signal to let interested Interfaces know that the request timed out
void TAEval::processCourseListRequest(QByteArray& packetData)	Processes server response to the request of viewing a list of Courses. Reads the packetData and formats the Course data into local instances to add to the client list of Courses.
void TAEval::processTeachingAssistantListRequest(QByteArray& packetData)	Processes server response to the request of viewing a list of TAs. Reads the packetData and formats the TA data into local instances to add to the client list of TAs.
void TAEval::processInstructorListRequest(QByteArray& packetData)	Processes server response to the request of viewing a list of Instructors. Reads the packetData and formats the Instructor data into local instances to add to the client list of Instructors.
void TAEval::processTaskListRequest(QByteArray& packetData)	Processes server response to the request of viewing a list of Tasks. Reads the packetData and formats the Task data into local instances to add to the client list of Tasks.
void TAEval::processCreateCourse(QByteArray& packetData)	Processes server response to the request of creating a new Course. Reads the packetData and, if the creation was successful on the server, formats the Course data into a local instance as an attribute of TAEval to monitor state.
void TAEval::processCreateTeachingAssistant(QByteArray& packetData)	Processes server response to the request of creating a new TA. Reads the packetData and, if the creation was successful on the server, formats the TA data into a local instance as an attribute of TAEval to monitor state.
void TAEval::processCreateInstructor(QByteArray& packetData)	Processes server response to the request of creating a new Instructor. Reads the packetData and, if the creation was successful on the server, formats the Instructor data into a local instance as an attribute of TAEval to monitor state.
void TAEval::processCreateTask(QByteArray& packetData)	Processes server response to the request of creating a new Task. Reads the packetData and, if the creation was successful on the server, formats the Task data into a local instance as an attribute of TAEval to monitor state.

void TAEval::processEditCourse(QByteArray& packetData)	Processes server response to the request of editing an existing Course. Reads the packetData and, if the modification was successful on the server, formats the Course data into a local instance as an attribute of TAEval to monitor state.
void TAEval::processEditTeachingAssistant(QByteArray& packetData)	Processes server response to the request of editing an existing TA. Reads the packetData and, if the modification was successful on the server, formats the TA data into a local instance as an attribute of TAEval to monitor state.
void TAEval::processEditInstructor(QByteArray& packetData)	Processes server response to the request of editing an existing Instructor. Reads the packetData and, if the modification was successful on the server, formats the Instructor data into a local instance as an attribute of TAEval to monitor state.
void TAEval::processEditTask(QByteArray& packetData)	Processes server response to the request of editing an existing Task. Reads the packetData and, if the modification was successful on the server, formats the Task data into a local instance as an attribute of TAEval to monitor state.
void TAEval::processDeleteCourse(QByteArray& packetData)	Processes server response to the request of deleting an existing Course. Reads the packetData to see whether the deletion was successful on the server.
void TAEval::processDeleteTeachingAssistant(QByteArray& packetData)	Processes server response to the request of deleting an existing TA. Reads the packetData to see whether the deletion was successful on the server.
void TAEval::processDeleteInstructor(QByteArray& packetData)	Processes server response to the request of deleting an existing Instructor. Reads the packetData to see whether the deletion was successful on the server.
void TAEval::processDeleteTask(QByteArray& packetData)	Processes server response to the request of deleting an existing Task. Reads the packetData to see whether the deletion was successful on the server.
void TAEval::processViewEvaluationsForTA(QByteArray& packetData)	Processes server response to the request of viewing a report of all evaluations for a given TA. Reads the packetData, which is a string text file generated on the server that emits a signal that the report is generated.
void TAEval::processViewEvaluationsForTerm(QByteArray& packetData)	Processes server response to the request of viewing a report of all evaluations for all TAs for a given term. Reads the packetData, which is a string text file generated on the server that emits a signal that the report is generated.
void TAEval::processViewEvaluationsForCourse(QByteArray& packetData)	Processes server response to the request of viewing a report of all evaluations for all TAs for a given Course. Reads the packetData, which is a string text file generated on the server that emits a signal that the report is generated.

void TAEval::processPacket(unsigned short packetId, QByteArray& packetData)	Gateway of processing the packet response from the server. Interprets the packetId and maps it to its appropriate function, passing it the packetData.
void TAEval::requestTimeout()	Alerts interested Interfaces that the request timed out.
void TAEval::initialize()	Initializes instance of TAEval by connecting the processPacket functions from the client's NetworkConnection instance to the server's, and connecting to the host.
void NetworkConnection::networkDataAvailable()	Checks for responses from the server's NetworkConnection instance. If one exists, it formats it into a QByteArray along with its packetId and directs the control flow to **TAEval::processPacket**
void NetworkConnection::sendPacket(unsigned short packetId, QByteArray& packetData)	Sends a message as a stream of bytes over the socket containing a unique ID for the message type, the number of bytes in the message and the message itself

Table of operations within RequestRetrieval

Operation	**Description**
void ClientConnection::sendCourseList()	Iterates through all of the Courses that were queried in **ClientConnection::processCourseListRequest** and formats the data into a packet to send over the network with its packetId, along with the number of Courses included in the header of the packet.
void ClientConnection::sendInstructorList()	Iterates through all of the Instructors that were queried in **ClientConnection::processInstructorListRequest** and formats the data into a packet to send over the network with its packetId, along with the number of Instructors included in the header of the packet.
void ClientConnection::sendTAList()	Iterates through all of the TA that were queried in **ClientConnection::processTAListRequest** and formats the data into a packet to send over the network with its packetId, along with the number of TAs included in the header of the packet.
void ClientConnection::sendTaskList()	Iterates through all of the Tasks that were queried in **ClientConnection::processTaskListRequest** and formats the data into a packet to send over the network with its packetId, along with the number of Tasks included in the header of the packet.

void ClientConnection::sendCourseCreatedSuccess(bool success)	Formats the Course that was created in the database into a packet to send over the network with its packetId if the creation was successful.
void ClientConnection::sendInstructorCreatedSuccess(bool success)	Formats the Instructor that was created in the database into a packet to send over the network with its packetId if the creation was successful.
void ClientConnection::sendTACreatedSuccess(bool success)	Formats the TA that was created in the database into a packet to send over the network with its packetId if the creation was successful.
void ClientConnection::sendTaskCreatedSuccess(bool success)	Formats the Task that was created in the database into a packet to send over the network with its packetId if the creation was successful.
void ClientConnection::sendCourseEditedSuccess(bool success)	Formats the Course that was modified in the database into a packet to send over the network with its packetId if the modification was successful.
void ClientConnection::sendInstructorEditedSuccess(bool success)	Formats the Instructor that was modified in the database into a packet to send over the network with its packetId if the modification was successful.
void ClientConnection::sendTAEditedSuccess(bool success)	Formats the TA that was modified in the database into a packet to send over the network with its packetId if the modification was successful.
void ClientConnection::sendTaskEditedSuccess(bool success)	Formats the Task that was modified in the database into a packet to send over the network with its packetId if the modification was successful.
void ClientConnection::sendCourseDeletedSuccess(bool success)	Formats the response from the deletion of the Course from the database into a packet to send over the network with its packetId.
void ClientConnection::sendInstructorDeletedSuccess(bool success)	Formats the response from the deletion of the Instructor from the database into a packet to send over the network with its packetId.

void ClientConnection::sendTADeletedSuccess(bool success)	Formats the response from the deletion of the TA from the database into a packet to send over the network with its packetId.
void ClientConnection::sendTaskDeletedSuccess(bool success)	Formats the response from the deletion of the Task from the database into a packet to send over the network with its packetId.
void ClientConnection::processEvaluationsForTA(QByteArray& packetData)	Reads the packetData for the specific TA who's evaluations are requested. Finds all tasks associated with the TA, averages the ratings per course, and sends back to the client a packet with the message consisting of the course code and associated average rating for all tasks as a string.
void ClientConnection::processEvaluationsForTerm(QByteArray& packetData)	Reads the packetData for the specific term's evaluations that are requested. Finds all courses assigned for that term, takes all TAs for those courses and sends back to the client a packet with the message consisting of the TA's full name and associated rating for all TAs as a string.
void ClientConnection::processEvaluationsForCourse(QByteArray& packetData)	Reads the packetData for the specific Course who's evaluations are requested. Finds all TAs assigned with the Course, averages their ratings, and sends back to the client a packet with the message consisting of the TA's full name and associated average rating for all TAs as a string.
void ClientConnection::processCourseListRequest(QByteArray& packetData)	Reads the packetData for the query parameters specified and queries the database for the specific Courses. Calls **ClientConnection::sendCourseList** to send the information back to the client.
void ClientConnection::processInstructorListRequest(QByteArray& packetData)	Reads the packetData for the query parameters specified and queries the database for the specific Instructors. Calls **ClientConnection::sendInstructorList** to send the information back to the client.
void ClientConnection::processTeachingAssistantListRequest(QByteArray& packetData)	Reads the packetData for the query parameters specified and queries the database for the specific TAs. Calls **ClientConnection::sendTAList** to send the information back to the client.
void ClientConnection::processTaskListRequest(QByteArray& packetData)	Reads the packetData for the query parameters specified and queries the database for the specific Tasks. Calls **ClientConnection::sendTaskList** to send the information back to the client.

void ClientConnection::processCreateCourseRequest(QByteArray& packetData)	Reads the packetData for the specified attributes for the new Course to be inserted into the database. Calls **ClientConnection::sendCourseCreatedSuccess** to send the information back to the client.
void ClientConnection::processCreateInstructorRequest(QByteArray& packetData)	Reads the packetData for the specified attributes for the new Instructor to be inserted into the database. Calls **ClientConnection::sendInstructorCreatedSuccess** to send the information back to the client.
void ClientConnection::processCreateTeachingAssistantRequest(QByteArray& packetData)	Reads the packetData for the specified attributes for the new TA to be inserted into the database. Calls **ClientConnection::sendTACreatedSuccess** to send the information back to the client.
void ClientConnection::processCreateTaskRequest(QByteArray& packetData)	Reads the packetData for the specified attributes for the new Task to be inserted into the database. Calls **ClientConnection::sendTaskCreatedSuccess** to send the information back to the client.
void ClientConnection::processEditCourseRequest(QByteArray& packetData)	Reads the packetData for the specified attributes for the existing Course to be modified in the database. Calls **ClientConnection::sendCourseEditedSuccess** to send the information back to the client.
void ClientConnection::processEditInstructorRequest(QByteArray& packetData)	Reads the packetData for the specified attributes for the existing Instructor to be modified in the database. Calls **ClientConnection::sendInstructorEditedSuccess** to send the information back to the client.
void ClientConnection::processEditTeachingAssistantRequest(QByteArray& packetData)	Reads the packetData for the specified attributes for the existing TA to be modified in the database. Calls **ClientConnection::sendTAEditedSuccess** to send the information back to the client.
void ClientConnection::processEditTaskRequest(QByteArray& packetData)	Reads the packetData for the specified attributes for the existing Task to be modified in the database. Calls **ClientConnection::sendTaskEditedSuccess** to send the information back to the client.
void ClientConnection::processDeleteCourseRequest(QByteArray& packetData)	Reads the packetData for the unique identifier for the existing Course to be deleted from the database. Calls **ClientConnection::sendCourseDeletedSuccess** to send the information back to the client.
void ClientConnection::processDeleteInstructorRequest(QByteArray& packetData)	Reads the packetData for the unique identifier for the existing Instructor to be deleted from the database. Calls **ClientConnection::sendInstructorDeletedSuccess** to send the information back to the client.

void ClientConnection::processDeleteTeachingAssistantRequest(QByteArray& packetData)	Reads the packetData for the unique identifier for the existing TA to be deleted from the database. Calls **ClientConnection::sendTADeletedSuccess** to send the information back to the client.
void ClientConnection::processDeleteTaskRequest(QByteArray& packetData)	Reads the packetData for the unique identifier for the existing Task to be deleted from the database. Calls **ClientConnection::sendTaskDeletedSuccess** to send the information back to the client.
void ClientConnection::startConnection()	Creates a socket to initialize *NetworkConnection* and connects the attribute *NetworkConnection*'s processPacket with the client's. Initializes the database.
void ClientConnection::processPacket(unsigned short packetId, QByteArray packetData)	Gateway of processing the packet response from the client. Interprets the packetId and maps it to its appropriate function, passing it the packetData.
void ClientConnection::connectionTimeout()	Alerts that the client connection timed out.
void Server::incomingConnection(int socketDescriptor)	Creates a new thread for every client connection. Initializes *ClientConnection*.
void NetworkConnection::networkDataAvailable()	Checks for responses from the client's NetworkConnection instance. If one exists, it formats it into a QByteArray along with its packetId and directs the control flow to **NetworkConnection::processPacket**
void NetworkConnection::sendPacket(unsigned short packetId, QByteArray& packetData)	Sends a message as a steram of bytes over the socket containing a unique ID for the message type, the number of bytes in the message and the message itself

Table of operations within DataSubmission

Operation	**Description**
int DBManager::createCourse(QString& name, QString& code, int year, QString& term, int instructorID)	Formats the input parameters into an SQL query, where the database creates the Course with the specified attributes and returns the ID

int DBManager::createInstructor(int instructorID, QString& f_name, QString& l_name, QString& dept)	Formats the input parameters into an SQL query, where the database creates the Instructor with the specified attributes and returns the ID
int DBManager::createTeachingAssistant(int studentID, QString& f_name, QString l_name, QString& degree, QString& major, int year)	Formats the input parameters into an SQL query, where the database creates the TA with the specified attributes and returns the ID
bool DBManager::createTask(QString& name, QString& desc, QString& comment, int rating, int studentID, int courseID)	Formats the input parameters into an SQL query, where the database creates the Task with the specified attributes and returns the ID
int DBManager::createCourseTA(int courseID, int studentID)	Formats the input parameters into an SQL query, where the database creates the TA assigned to the specific Course matching to the courseID and returns the ID
bool d DBManager::eleteCourse(int courseID)	Formats the input parameter into an SQL query, where the database deletes the Course with the specified ID. Returns true if the deletion was successful
bool DBManager::deleteInstructor(int instructorID)	Formats the input parameter into an SQL query, where the database deletes the Instructor with the specified ID. Returns true if the deletion was successful
bool DBManager::deleteTeachingAssistant(int studentID)	Formats the input parameter into an SQL query, where the database deletes the TA with the specified ID. Returns true if the deletion was successful
bool DBManager::deleteTask(int taskID)	Formats the input parameter into an SQL query, where the database deletes the Task with the specified ID. Returns true if the deletion was successful
bool DBManager::editCourse(int courseID, QString& name, QString& code, int year, QString& term)	Formats the input parameters into an SQL query, where the database edits the Course with the specified attributes and returns True if the modification was successful
bool DBManager::editInstructor(int instructorID, QString& f_name, QString& l_name, QString& dept)	Formats the input parameters into an SQL query, where the database edits the Instructor with the specified attributes and returns True if the modification was successful

bool DBManager::editTeachingAssistant(int studentID, QString& f_name, QString& l_name, QString& degree, QString& major, int year)	Formats the input parameters into an SQL query, where the database edits the TA with the specified attributes and returns True if the modification was successful
bool DBManager::editTask(int taskID, QString& name, QString& desc, QString& comment, QString& rating)	Formats the input parameters into an SQL query, where the database edits the Task with the specified attributes and returns True if the modification was successful

Table of operations within DataRetrieval

Operation	**Description**
void DBManager::getCourse(QString& term, int year)	Constructs an SQL query using the input parameters to find the specific Course in the database. Creates a local instance of the Course to add to the server's list of Courses.
void DBManager::getInstructorbyID(int instructorID)	Constructs an SQL query using the input parameter to find the specific Instructor in the database. Creates a local instance of the Instructor to add to the server's list of Instructors.
void DBManager::getTeachingAssistantbyID(int studentID)	Constructs an SQL query using the input parameter to find the specific TA in the database. Creates a local instance of the TA to add to the server's list of TAs.
void DBManager::getTask(int courseID, int taID)	Constructs an SQL query using the input parameters to find the specific Task in the database. Creates a local instance of the Task to add to the server's list of Tasks.
void DBManager::getCourseTeachingAssistant(int courseID)	Constructs an SQL query using the input parameter to find the specific TA in the database using **DBManager::getTeachingAssistantbyID**.
DBManager::getTaskbyID(int taskID)	Constructs an SQL query using the input parameter to find the specific Task in the database. Creates a local instance of the Task to add to the server's list of Tasks

5. Class Interfaces

Class Name	**AdminUI**
Services Traceability	SE-01
Class Description	Responsible for presenting the GUI for users with administrator privileges, who have the ability to create, edit, or delete courses, instructors, and TAs. They are also able to generate reports on TA evaluation data by either viewing all ratings for a given TA, all TA's ratings for a given term, or all TA's ratings for a given Course. They are also able to view all lists of data including Courses, Instructors, TeachingAssistants, and Tasks.

Class Name	**InstructorUI**
Services Traceability	SE-02
Class Description	Responsible for presenting the GUI for users with instructor privileges, who have the ability to create, edit, or delete tasks. They are also able to view all lists of data including Courses, Instructors, TeachingAssistants, and Tasks.

Class Name	**TeachingAssistantUI**
Services Traceability	SE-03
Class Description	Responsible for presenting the GUI for users with teaching assistant privileges, also able to view all lists of data including Courses, Instructors, and Tasks. Through the list of Tasks, they are able to view evaluations for completed Tasks.

Class Name	**CreateCourseControl**
Services Traceability	SE-04
Class Description	Responsible for listening to event of "Create Course" button being clicked from *AdminInterface* and providing the form data to **TAEval::createCourse**

Class Name	**EditCourseControl**
Services Traceability	SE-04
Class Description	Responsible for listening to event of "Edit Course" button being clicked from *AdminInterface*, using the form data to edit a local instance and pass that local instance to **TAEval::editCourse**

Class Name	**DeleteCourseControl**
Services Traceability	SE-04
Class Description	Responsible for listening to event of “Delete Course” button being clicked from *AdminInterface* and providing the Course to delete to **TAEval::deleteCourse**

Class Name	**CreateInstructorControl**
Services Traceability	SE-05
Class Description	Responsible for listening to event of “Create Instructor” button being clicked from *AdminInterface* and providing the form data to **TAEval::createInstructor**

Class Name	**EditInstructorControl**
Services Traceability	SE-05
Class Description	Responsible for listening to event of “Edit Instructor” button being clicked from *AdminInterface*, using the form data to edit a local instance and pass that local instance to **TAEval::editInstructor**

Class Name	**DeleteInstructorControl**
Services Traceability	SE-05
Class Description	Responsible for listening to event of “Delete Instructor” button being clicked from *AdminInterface* and providing the Instructor to delete to **TAEval::deleteInstructor**

Class Name	**CreateTeachingAssistantControl**
Services Traceability	SE-06
Class Description	Responsible for listening to event of “Create Teaching Assistant” button being clicked from *AdminInterface* and providing the form data to **TAEval::createTeachingAssistant**

Class Name	**EditTeachingAssistantControl**
Services Traceability	SE-06
Class Description	Responsible for listening to event of “Edit Teaching Assistant” button being clicked from *AdminInterface*, using the form data to edit a local instance and pass that local instance to **TAEval::editTeachingAssistant**

Class Name	**DeleteTeachingAssistantControl**
Services Traceability	SE-06
Class Description	Responsible for listening to event of “Delete Teaching Assistant” button being clicked from *AdminInterface* and providing the TeachingAssistant to delete to **TAEval::deleteTeachingAssistant**

Class Name	**CreateTaskControl**
Services Traceability	SE-07
Class Description	Responsible for listening to event of “Create Task” button being clicked from *InstructorInterface* and providing the form data to **TAEval::createTask**

Class Name	**EditTaskControl**
Services Traceability	SE-07
Class Description	Responsible for listening to event of “Edit Task” button being clicked from *InstructorInterface*, using the form data to edit a local instance and pass that local instance to **TAEval::editTask**

Class Name	**DeleteTaskControl**
Services Traceability	SE-07
Class Description	Responsible for listening to event of “Delete Task” button being clicked from *InstructorInterface* and providing the Course to delete to **TAEval::deleteTask**

Class Name	**RunReportsControl**
Services Traceability	SE-08
Class Description	Responsible for listening to events of either the “View Evaluations for TA”, “View Evaluations for Term”, or “View Evaluations for Course” buttons being clicked from *AdminInterface* and providing the corresponding correct call to *TAEval*, either **TAEval::viewEvaluationForTA, TAEval::viewEvaluationForTerm,** or **TAEval::viewEvaluationForCourse** respectively.

Class Name	**ViewCoursesControl**
Services Traceability	SE-09
Class Description	Responsible for listening to requests to view a list of Courses.

Class Name	**ViewInstructorsControl**
Services Traceability	SE-09
Class Description	Responsible for listening to requests to view a list of Instructors.

Class Name	**ViewTeachingAssistantControl**
Services Traceability	SE-09
Class Description	Responsible for listening to requests to view a list of TeachingAssistants.

Class Name	**ViewTasksControl**
Services Traceability	SE-09
Class Description	Responsible for listening to requests to view a list of Tasks.

Class Name	**TAEval**
Services Traceability	SE-10
Class Description	Façade object responsible for handling all requests from the client interface, formatting the data in a method that the network accepts and sending the data over the network to the server.

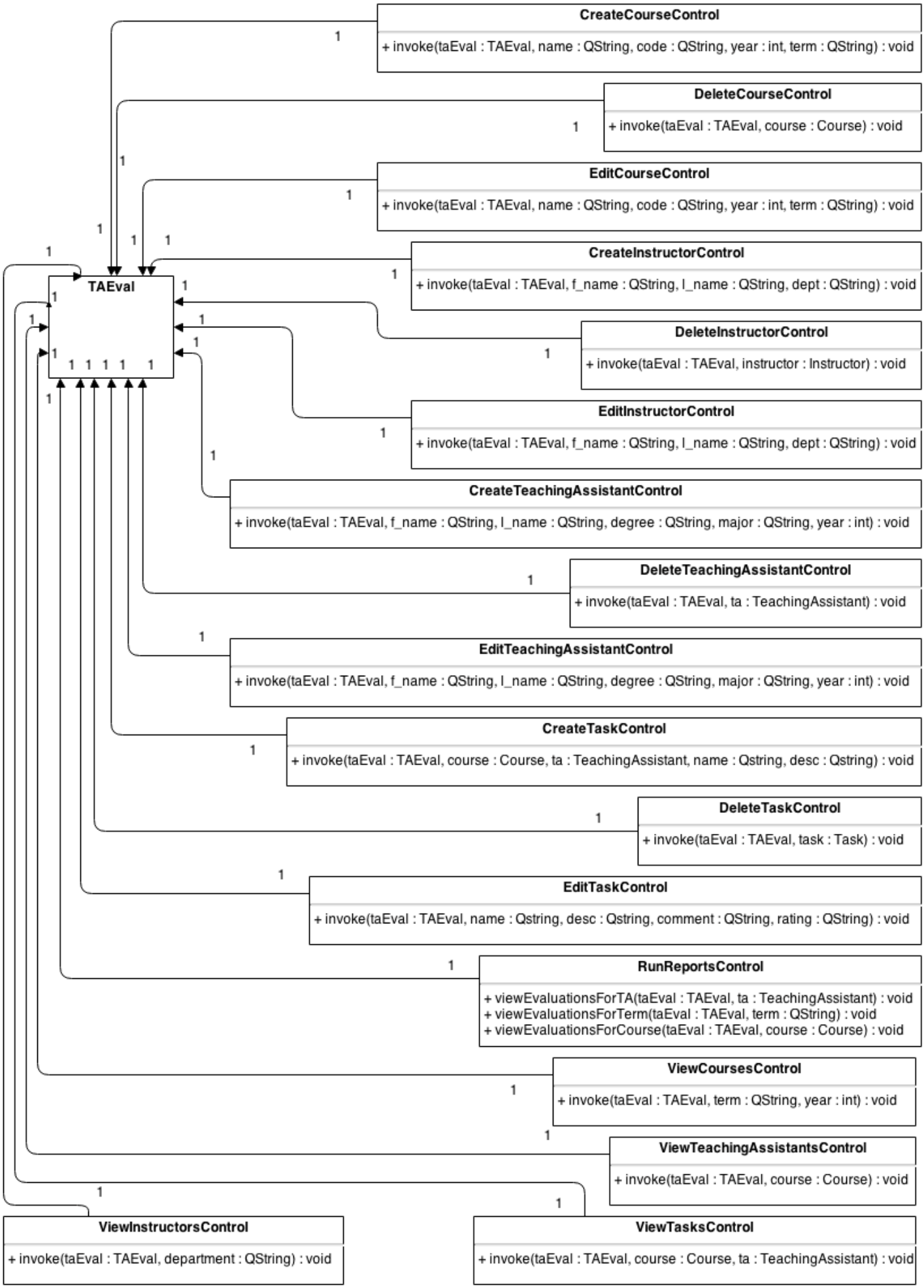
Class Name	**NetworkConnection**
Services Traceability	SE-10, SE-11
Class Description	Listens for data responses from the server and is responsible for sending packets (streams of bytes) over a socket between the client and the server.

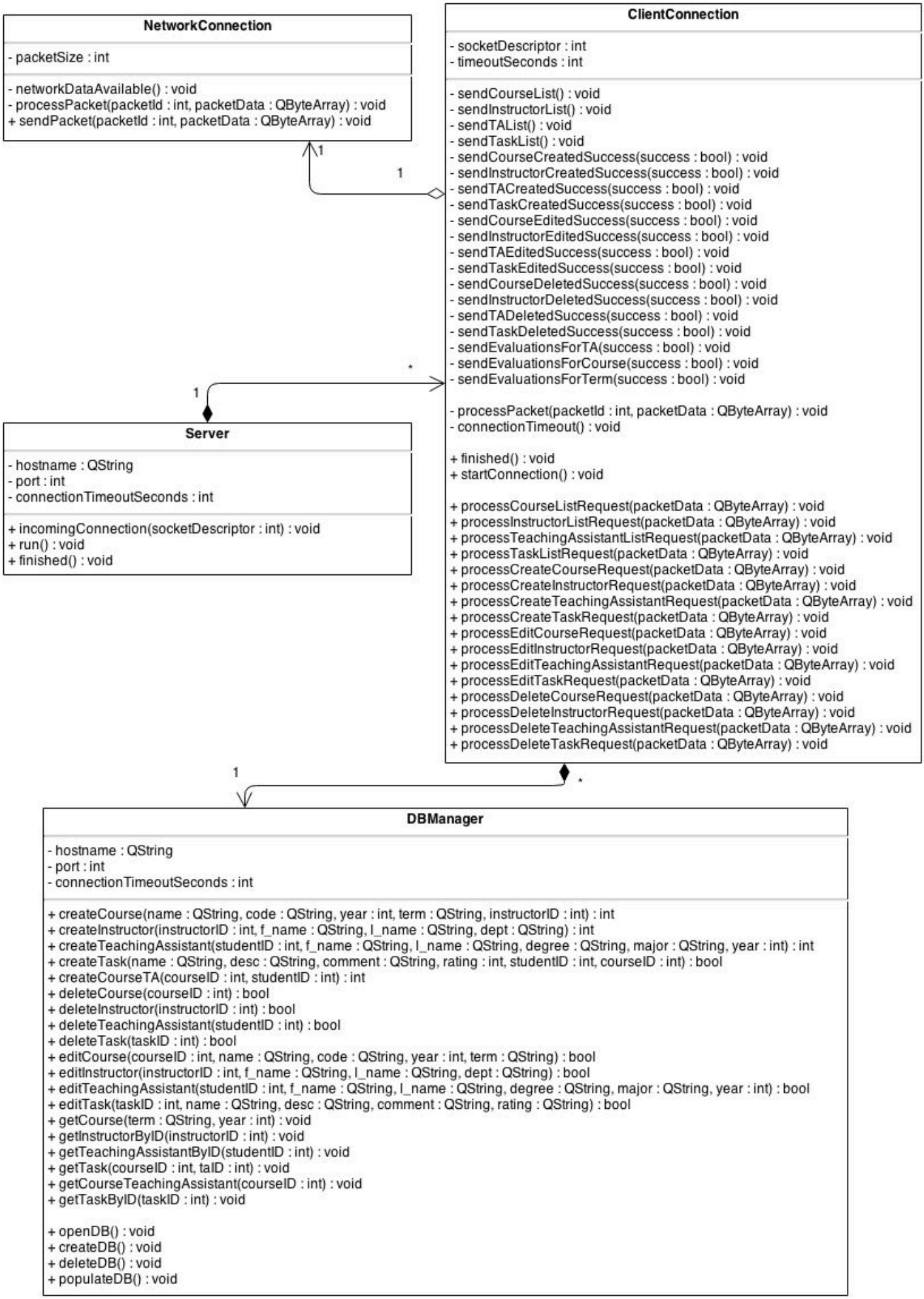
Class Name	**ClientConnection**
Services Traceability	SE-11
Class Description	Façade object responsible for handling all requests from the client's network, formatting the data in a method that is acceptable to make the appropriate server and/or database call. The result of the operation and pertinent information is sent over the network back to the client.

Class Name	**Server**
Services Traceability	SE-11
Class Description	Connects any clients attempting to connect to a server to the server instance.

Class Name	**DBManager**
Services Traceability	SE-12, SE-13
Class Description	Proxy object responsible for facilitating access to the database. Takes requests forwarded from the client by the server, and only creates queries when necessary to retrieve the requested data, or, to modify the database through creation, modification, or deletion.


TAEval
- hostname : QString
- port : int
- currentPacketId : int
- requestTimeoutSeconds : int
- clearClientState() : void
- processPacket(packetId : int, packetData : QByteArray) : void
- requestTimeout() : void
- processCourseListRequest(packetData : QByteArray) : void
- processTeachingAssistantListRequest(packetData : QByteArray) : void
- processInstructorListRequest(packetData : QByteArray) : void
- processTaskListRequest(packetData : QByteArray) : void
- processCreateCourse(packetData : QByteArray) : void
- processCreateTeachingAssistant(packetData : QByteArray) : void
- processCreateInstructor(packetData : QByteArray) : void
- processCreateTask(packetData : QByteArray) : void
- processEditCourse(packetData : QByteArray) : void
- processEditTeachingAssistant(packetData : QByteArray) : void
- processEditInstructor(packetData : QByteArray) : void
- processEditTask(packetData : QByteArray) : void
- processDeleteCourse(packetData : QByteArray) : void
- processDeleteTeachingAssistant(packetData : QByteArray) : void
- processDeleteInstructor(packetData : QByteArray) : void
- processDeleteTask(packetData : QByteArray) : void
- processViewEvaluationsForTA(packetData : QByteArray) : void
- processViewEvaluationsForTerm(packetData : QByteArray) : void
- processViewEvaluationsForCourse(packetData : QByteArray) : void
+ initialize() : void
+ requestCourseList(term : QString, year : int) : void
+ requestTeachingAssistantList(course : Course) : void
+ requestTaskList(course : Course, ta : TeachingAssistant) : void
+ requestInstructorList(department : QString) : void
+ createTask(course : Course, ta : TeachingAssistant, name : QString, desc : QString) : void
+ createCourse(name : QString, code : QString, year : int, term : QString) : void
+ createInstructor(f_name : QString, l_name : QString, dept : QString) : void
+ createTeachingAssistant(f_name : QString, l_name : QString, degree : QString, major : QString, year : int) : void
+ deleteTask(task : Task) : void
+ deleteCourse(course : Course) : void
+ deleteInstructor(instructor : Instructor) : void
+ deleteTeachingAssistant(ta : TeachingAssistant) : void
+ editTask(task : Task) : void
+ editCourse(course : Course) : void
+ editInstructor(instructor : Instructor) : void
+ editTeachingAssistant(ta : TeachingAssistant) : void
+ viewEvaluationsForTA(ta : TeachingAssistant) : void
+ viewEvaluationsForTerm(term : QString) : void
+ viewEvaluationsForCourse(course : Course) : void
+ courseListUpdated(courseList : vector<Course>) : void
+ teachingAssistantListUpdated(courseList : vector<TeachingAssistant>) : void
+ instructorListUpdated(courseList : vector<Instructor>) : void
+ taskListUpdated(courseList : vector<Task>) : void
+ courseCreated(course : Course) : void
+ teachingAssistantCreated(ta : TeachingAssistant) : void
+ instructorCreated(instructor : Instructor) : void
+ taskCreated(task : Task) : void
+ courseEdited(course : Course) : void
+ teachingAssistantEdited(ta : TeachingAssistant) : void
+ instructorEdited(instructor : Instructor) : void
+ taskEdited(task : Task) : void
+ courseDeleted(success : bool) : void
+ teachingAssistantDeleted(success : bool) : void
+ instructorDeleted(success : bool) : void
+ taskDeleted(success : bool) : void
1
1
NetworkConnection
- packetSize : int
- networkDataAvailable() : void
- processPacket(packetId : int, packetData : QByteArray) : void
+ sendPacket(packetId : int, packetData : QByteArray) : void


CreateCourseControl
+ invoke(taEval : TAEval, name : QString, code : QString, year : int, term : QString) : void
DeleteCourseControl
+ invoke(taEval : TAEval, course : Course) : void
EditCourseControl
+ invoke(taEval : TAEval, name : QString, code : QString, year : int, term : QString) : void
CreateInstructorControl
+ invoke(taEval : TAEval, f_name : QString, l_name : QString, dept : QString) : void
TAEval
DeleteInstructorControl
+ invoke(taEval : TAEval, instructor : Instructor) : void
EditInstructorControl
+ invoke(taEval : TAEval, f_name : QString, l_name : QString, dept : QString) : void
CreateTeachingAssistantControl
+ invoke(taEval : TAEval, f_name : QString, l_name : QString, degree : QString, major : QString, year : int) : void
DeleteTeachingAssistantControl
+ invoke(taEval : TAEval, ta : TeachingAssistant) : void
EditTeachingAssistantControl
+ invoke(taEval : TAEval, f_name : QString, l_name : QString, degree : QString, major : QString, year : int) : void
CreateTaskControl
+ invoke(taEval : TAEval, course : Course, ta : TeachingAssistant, name : Qstring, desc : Qstring) : void
DeleteTaskControl
+ invoke(taEval : TAEval, task : Task) : void
EditTaskControl
+ invoke(taEval : TAEval, name : Qstring, desc : Qstring, comment : QString, rating : QString) : void
RunReportsControl
+ viewEvaluationsForTA(taEval : TAEval, ta : TeachingAssistant) : void
+ viewEvaluationsForTerm(taEval : TAEval, term : QString) : void
+ viewEvaluationsForCourse(taEval : TAEval, course : Course) : void
ViewCoursesControl
+ invoke(taEval : TAEval, term : QString, year : int) : void
ViewTeachingAssistantsControl
+ invoke(taEval : TAEval, course : Course) : void
ViewInstructorsControl
+ invoke(taEval : TAEval, department : QString) : void
ViewTasksControl
+ invoke(taEval : TAEval, course : Course, ta : TeachingAssistant) : void


NetworkConnection
- packetSize : int
- networkDataAvailable() : void
- processPacket(packetId : int, packetData : QByteArray) : void
+ sendPacket(packetId : int, packetData : QByteArray) : void
1
1
ClientConnection
- socketDescriptor : int
- timeoutSeconds : int
- sendCourseList() : void
- sendInstructorList() : void
- sendTAList() : void
- sendTaskList() : void
- sendCourseCreatedSuccess(success : bool) : void
- sendInstructorCreatedSuccess(success : bool) : void
- sendTACreatedSuccess(success : bool) : void
- sendTaskCreatedSuccess(success : bool) : void
- sendCourseEditedSuccess(success : bool) : void
- sendInstructorEditedSuccess(success : bool) : void
- sendTAEditedSuccess(success : bool) : void
- sendTaskEditedSuccess(success : bool) : void
- sendCourseDeletedSuccess(success : bool) : void
- sendInstructorDeletedSuccess(success : bool) : void
- sendTADeletedSuccess(success : bool) : void
- sendTaskDeletedSuccess(success : bool) : void
- sendEvaluationsForTA(success : bool) : void
- sendEvaluationsForCourse(success : bool) : void
- sendEvaluationsForTerm(success : bool) : void
- processPacket(packetId : int, packetData : QByteArray) : void
- connectionTimeout() : void
+ finished() : void
+ startConnection() : void
+ processCourseListRequest(packetData : QByteArray) : void
+ processInstructorListRequest(packetData : QByteArray) : void
+ processTeachingAssistantListRequest(packetData : QByteArray) : void
+ processTaskListRequest(packetData : QByteArray) : void
+ processCreateCourseRequest(packetData : QByteArray) : void
+ processCreateInstructorRequest(packetData : QByteArray) : void
+ processCreateTeachingAssistantRequest(packetData : QByteArray) : void
+ processCreateTaskRequest(packetData : QByteArray) : void
+ processEditCourseRequest(packetData : QByteArray) : void
+ processEditInstructorRequest(packetData : QByteArray) : void
+ processEditTeachingAssistantRequest(packetData : QByteArray) : void
+ processEditTaskRequest(packetData : QByteArray) : void
+ processDeleteCourseRequest(packetData : QByteArray) : void
+ processDeleteInstructorRequest(packetData : QByteArray) : void
+ processDeleteTeachingAssistantRequest(packetData : QByteArray) : void
+ processDeleteTaskRequest(packetData : QByteArray) : void
*
1
Server
- hostname : QString
- port : int
- connectionTimeoutSeconds : int
+ incomingConnection(socketDescriptor : int) : void
+ run() : void
+ finished() : void
1
*
DBManager
- hostname : QString
- port : int
- connectionTimeoutSeconds : int
+ createCourse(name : QString, code : QString, year : int, term : QString, instructorID : int) : int
+ createInstructor(instructorID : int, f_name : QString, l_name : QString, dept : QString) : int
+ createTeachingAssistant(studentID : int, f_name : QString, l_name : QString, degree : QString, major : QString, year : int) : int
+ createTask(name : QString, desc : QString, comment : QString, rating : int, studentID : int, courseID : int) : bool
+ createCourseTA(courseID : int, studentID : int) : int
+ deleteCourse(courseID : int) : bool
+ deleteInstructor(instructorID : int) : bool
+ deleteTeachingAssistant(studentID : int) : bool
+ deleteTask(taskID : int) : bool
+ editCourse(courseID : int, name : QString, code : QString, year : int, term : QString) : bool
+ editInstructor(instructorID : int, f_name : QString, l_name : QString, dept : QString) : bool
+ editTeachingAssistant(studentID : int, f_name : QString, l_name : QString, degree : QString, major : QString, year : int) : bool
+ editTask(taskID : int, name : QString, desc : QString, comment : QString, rating : QString) : bool
+ getCourse(term : QString, year : int) : void
+ getInstructorByID(instructorID : int) : void
+ getTeachingAssistantByID(studentID : int) : void
+ getTask(courseID : int, taID : int) : void
+ getCourseTeachingAssistant(courseID : int) : void
+ getTaskByID(taskID : int) : void
+ openDB() : void
+ createDB() : void
+ deleteDB() : void
+ populateDB() : void