Bilkent University

Department of Computer Engineering

CS319- OBJECT ORIENTED SOFTWARE ENGINEERING

ANALYSIS AND DESIGN REPORT

Grader++

Group Members: Ali Burak Erdoğan, Süleyman Can Özülkü, Ömer Eren,

Yusuf Said Canbaz

Contents

1. Introduction

Grader++ is a code evaluation system we plan to develop in aid to instructors and teaching assistants. Most of the time, instructors' method of grading students' codes involves steps that can be automated by a software system. For example, an instructor firstly compiles student's code and checks whether there is a compilation error, then he/she proceeds by checking if the code has memory leaks, and tries a few test cases and compares output with student's solution. Steps like these can be easily automated by a software and reduces the time it takes for instructors to grade code submissions.
There are problems automating these kinds of tasks. Students might submit malicious codes which can be caught by instructor by scrutinizing the code. Automating this process and handling dangerous codes is difficult. We plan to make grader++ secure by using technologies like containers. Also we will make it extendible, since every course, and even individual homeworks, needs different evaluation steps. Another planned capability for our system is working fast even under a huge work load, because (for example) a course could be taken by a lot of students, and it is possible that a significant amount of students will submit their code at the last minute and it will create a lot of load on the system. In this kind of situation, system should be able to handle evaluating process very efficiently so that students can see the analysis of their code.

Grader++ has two parts: Core part and UI part. Core part will be an extendible system composed of evaluators, evaluating submissions concurrently. UI part will give ability to instructors to add homeworks to the system and students to submit their code. Core part will be a restful api server, accepting task descriptions and submissions. UI part will interact core part through http requests and will have a web frontend in which instructors and students could interact core part with ease. Separating core part from UI will enable us to easily extend our system and will enable us to use more than one server, so that UI part will be accessible under heavy load.

2. Requirement Analysis

2.1 Overview

As the group #9 we decided Grader++ for our project's name . We aim to develop an automated grading service. This service can be useful for a lot of different purposes, however we decided to develop Grader++ for helping with grading programming homeworks, and for helping students with the development of their programming skills by providing them with analysis tools. We want to make a service which is somewhat similar to hackerrank and codeforces. We decided to use JAVA for implementation. There is a brief information about Grader++ in "Introduction" part of this report. After short description of the game, there is an explanation of why we have chosen to work on Grader++. In this report, requirements of the project will be given under two headings: Functional Requirements and Nonfunctional Requirements. System models of the service will also be described in detail. System models section includes domain analysis, use case analysis, sequence diagrams, state and activity diagrams. After System models section, detailed description of Grader++ features will be provided.

2.2 Functional Requirements

Register

- TA and student will register in standard way with generating a unique username and password
- Instructor must contact with admin to create an instructor account.

Login

- User name and password required log-in mechanism.
- Login system is common to for all types of users. (TA, instructor, student)

After Login

- Student should get new tasks from the courses that he subscribed.
- Student should be able to generate a new submission for a selected task
- Student should be able to view old submissions for a particular task
- Student should be able to subscribe or unsubscribe from a course.
- Instructor should be able add or delete course
- Instructor should be able to add or delete a task
- Instructor should be able to assign a TA for a course.
- Instructor should be able see old submissions
- TA should be able to edit a task that he assigned
- TA should be able to see old submissions.

2.3 Non-Functional Requirements

User-Friendly UI

- Basic menu with a single page application
- No recursive paths in application

Performance

- Make a queue of submissions to divide load
- Lightweight single page application UI to prevent unnecessary loads

Security

- Disable to see others code and submissions
- Disable to see closed tasks

- To create instructor account admin is required

Robustness

- Check submitted files for malware
- Check submitted code for malware

2.4 Constraints

- The user interface should be in the web platform. Thus, the user can interact with the system through a web browser.
- The web-based user interface should be built with Java and by following the MVC (Model-View-Controller) principle in order to handle the future modifications quickly.
- The back-end mechanism, the essential part of the code evaluating system should also be implemented in Java, and if needed C++.
- The live server machine which will serve to code compiling, evaluating and grading, should use Ubuntu 12 or above.

2.5 Scenarios

In this section we provide some specific scenarios which will help us derive use case model of the overall system.

2.5.1 Scenario Name: addCourseToSystem

Participating Actors: ozgur:Instructor

Flow of events:

1) Ozgur selects “Instructor” as the user type on the login page
2) Ozgur types username and password, and clicks login button.
3) Web client provides the Instructor Home Page for viewing.
4) Ozgur clicks “add course” button.
5) Client provides the add course page and waits for ozgur to confirm the course addition.

6) Ozgur fills the forms for “course name” with “cs202”, adds a course description, and confirms the creation of coruse by clicking “create course” button.

7) Client provides ozgur with insctructor home page.

2.5.2 Scenario Name: deleteCourseCS202

Participating Actors: ozgur:Instructor

Flow of events:

1) ozgur selects “Instructor” as the user type on the login page

2) ozgur types username and password, and clicks login button.

3) Web client provides the Instructor Home Page for viewing.

4) ozgur selects the course cs202 to delete it from the system.

5) ozgur clicks “delete course” button.

6) Web client provides a pop-up alert asking ozgur to confirm the deletion.

7) ozgur confirms the deletion by clicking the “confirm” button.

8) Client provides the Instructor Home Page for ozgur to view.

2.5.3 Scenario Name: addTaskForCS202

Participating Actors: ozgur:Instructor

Flow of events:

1) ozgur selects “Instructor” as the user type on the login page

2) ozgur types username and password, and clicks login button.

3) Web client provides the Instructor Home Page for viewing.

4) ozgur selects the course CS202 to add a task to, and clicks “add task” button..

5) Client shows the form page for adding a new task.

6) ozgur fills the required information such as “information”, “test case”, “due date”, and then clicks the “add” button.

7) Client provides the Instuctor Home Page to ozgur.

2.5.4 Scenario Name: submitCodeToCS202

Participating Actors: omer:Student

Flow of events:

1) omer selects “Student” as the user type on the login page

2) omer types username and password, and clicks login button.

3) Web client provides the Student Home Page for viewing.

4) omer selects the hw1 of CS202 to make submission to, and clicks “submit” button.

5) Client shows the form page for submitting code.

6) omer selects the code file he wants to submit from his computer, and then clicks “submit” button.

7) Client provides omer the Student Home Page

2.5.5 Scenario Name: viewSubmissionsForCS202HW1

Participating Actors: fatma:TeachingAssistant

Flow of events:

1) fatma logs in to the system as a TA.

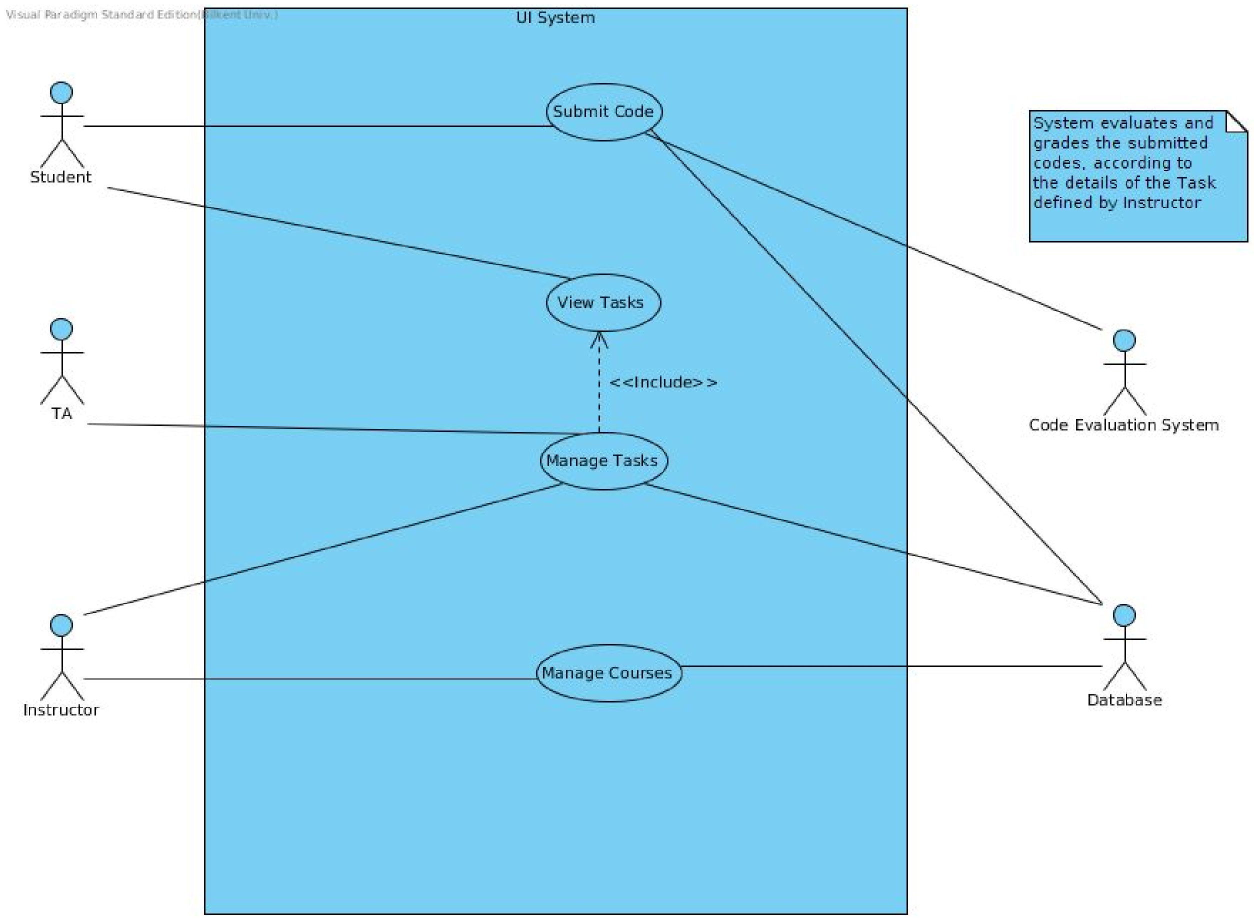
2) Web client provides fatma with TA Home Page.

3) fatma selects the course cs202 and hw1 to view the status of every submission on hw1.

4) Client provides fatma with “view submissions” page.

2.6 Use Case Models

This sections explains the main use cases of Grader++ in detail.



Visual Paradigm Standard Edition(Bilkent Univ.)
UI System
Student
TA
Instructor
Submit Code
View Tasks
<<Include>>
Manage Tasks
Manage Courses
System evaluates and grades the submitted codes, according to the details of the Task defined by Instructor
Code Evaluation System
Database

Figure: Use Case Diagram for Grader++

2.6.1 Submit Code

Use Case Name: Submit Code
Primary Actor: Student
Stakeholders and Interests:

- Student aims to make his/her submission to the related Task of his/her Course.
- Submission must be received by System and put into progress of evaluation and grading.

Pre-condition: Student must be logged in. Also the Course that Student is taking must have a Task.

Post-condition: The submission must be obtained by System and saved. System should evaluate the submission in terms of requirements criteria.

Entry Condition: Student presses the "Quick Submission" link from homepage.

Exit Condition: Student presses the "Submit" button of the submission form.

Event Flow:

1. Student selects a Task from the list.
2. Student selects the ".zip" file which contains all the required files from "Browse" button.
3. Student presses the "Submit" button.

Alternative Flows:

1. If Student wants to cancel submission:
    a. Presses the "Cancel Submission" button of the form.
    b. He/she returns to the Main Page.

2.6.2 View Tasks

Use Case Name: View Tasks

Primary Actor: Student, TA, Instructor

Stakeholders and Interests:

1. Student should see the details and evaluation state of the submission he/she made.
2. Instructor and TA should see the details and evaluation state of the submissions made by students on the selected Task.
3. Student, Instructor, TA should see all the details of a specific Task.

Pre-condition: User must be logged in.

Post-condition: -

Entry Condition: Instructor, TA or Student presses a particular Task's "View Submissions & Details" link which is located on a specific Course's page.

Exit Condition: User presses the "Return to Main Page" button.

Event Flow:

1. User should see a data table which consists of rows that each represents a submission, and columns such as submitter name, submission date, evaluation grade, the name of the Task that is submitted to etc.

Alternative Flows:

1. If User wants to see the details of the submission:
    a. Presses the button at the left corner of the row of Submission.
    b. User should see the details in a new page.
2. If User wants to reorder the results by a column in the table:
    a. Presses the head of the column.
    b. All the rows should be reordered by the column (ascending or descending)

2.6.3 Manage Tasks

Use Case Name: Manage Tasks
Primary Actor: Instructor, TA

Stakeholders and Interests:

- User should be able to define a new Task and determine its qualifications. Also, User can assign a TA as a manager.
- User should be able to modify a specific Task's qualifications such as task description, grading policy, due date, required files, makefiles, test files.

Pre-condition: User must be logged in. User should be authorized to manage Tasks.
Post-condition: The overall changes like adding or editing a Task, should be saved to the System.

Entry Condition: User presses the "Task Manager" button from a particular Course's page.
Exit Condition: User presses the "Return to Main Page" button.

Event Flow:

A. If User wants to add a task:
    a. Chooses the "Add Task" tab from panel.
    b. Fills the required fields
    c. Clicks to the "Add" button
B. If User wants to edit a task:
    a. Clicks to a task's "Edit" button
    b. Makes necessary changes like changing the name, description, grading policy, due date, compiling options, adding test files.
    c. Clicks to the "Save" button.
C. If User wants to view a Task's submissions
    a. Clicks to the "Submissions" button.

2.6.4 Manage Courses

Use Case Name: Manage Courses
Primary Actor: Instructor

Stakeholders and Interests:

- User should be able to define a new Course and determine its qualifications.
- User should be able to modify a specific Course's qualifications such as course description, course name.

Pre-condition: User must be logged in. User should be authorized to manage Courses.
Post-condition: The overall changes like adding or editing a Course, should be saved to the System.

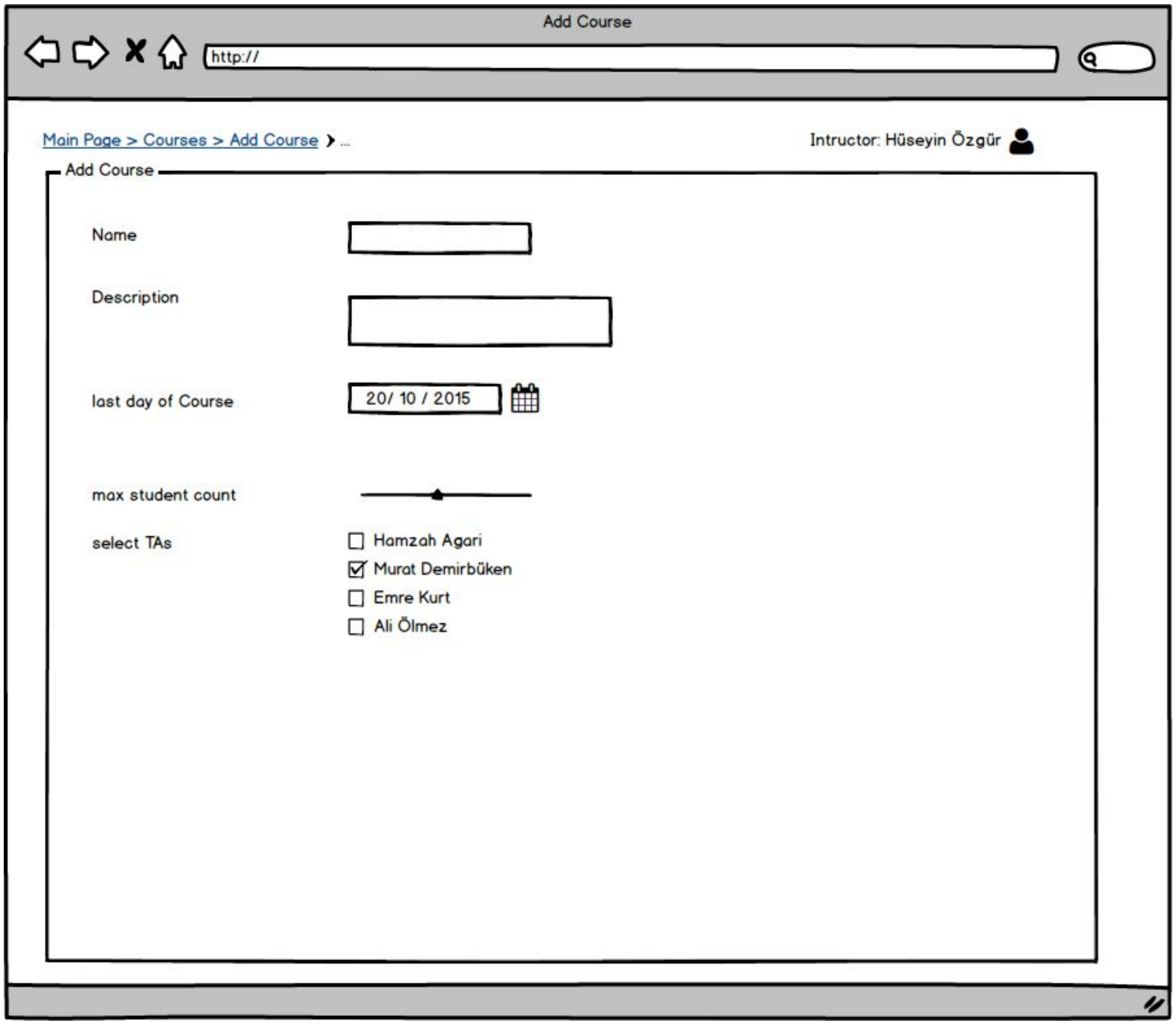
Entry Condition: User presses the "Course Manager" button from a particular Course's page.
Exit Condition: User presses the "Return to Main Page" button.

Event Flow:

A. If User wants to add a course:
   a. Chooses the "Add Course" tab from panel.
   b. Fills the required fields
   c. Clicks to the "Add" button
B. If User wants to edit a course:
   a. Clicks to a task's "Edit" button
   b. Makes necessary changes like changing the name, description, grading policy, due date, compiling options, adding test files.
   c. Clicks to the "Save" button.
C. If User wants to view a Course's tasks
   a. Clicks to the "Tasks" button.

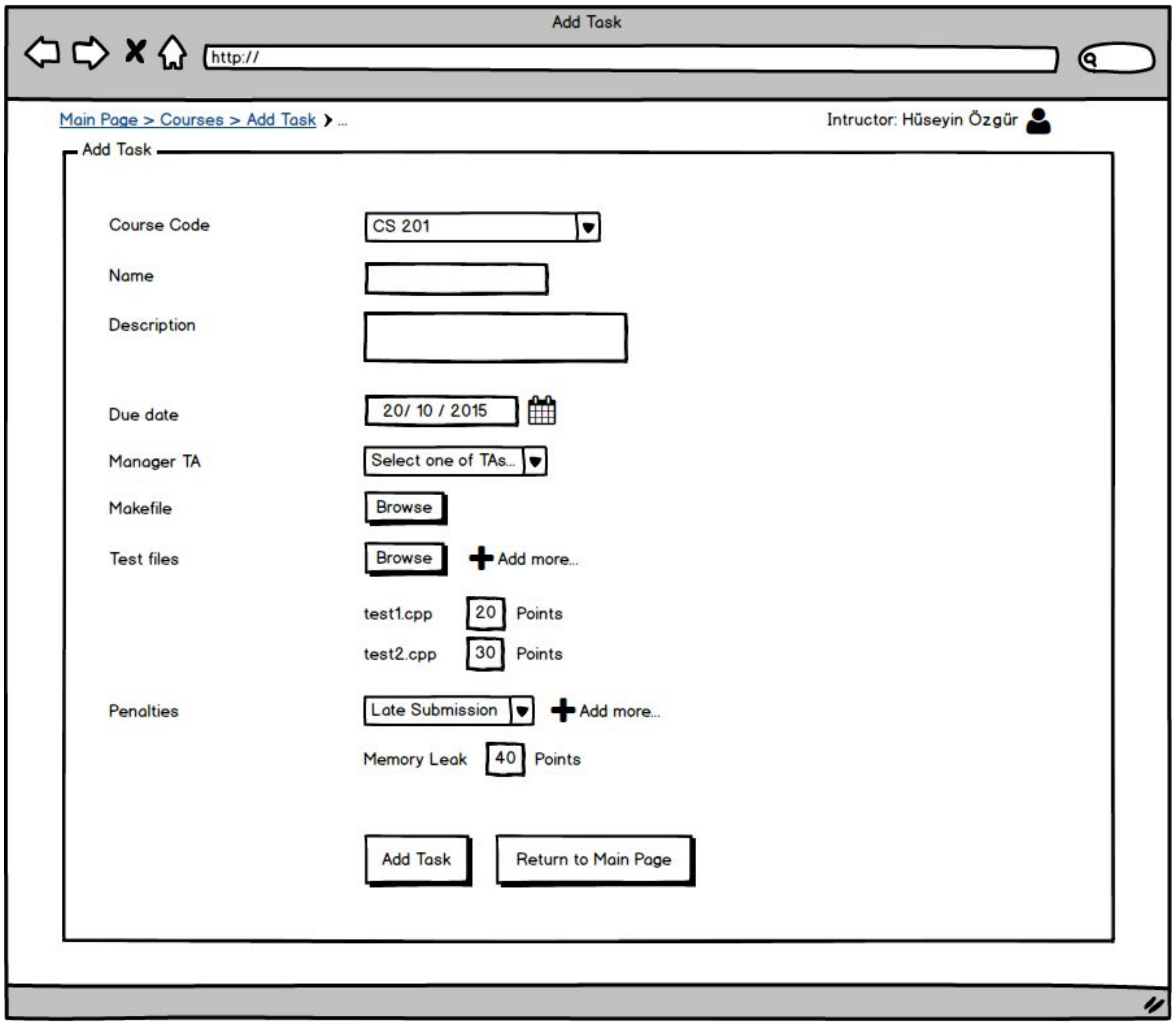
2.7 User Interface

2.7.1 Instructor Add Course Screen



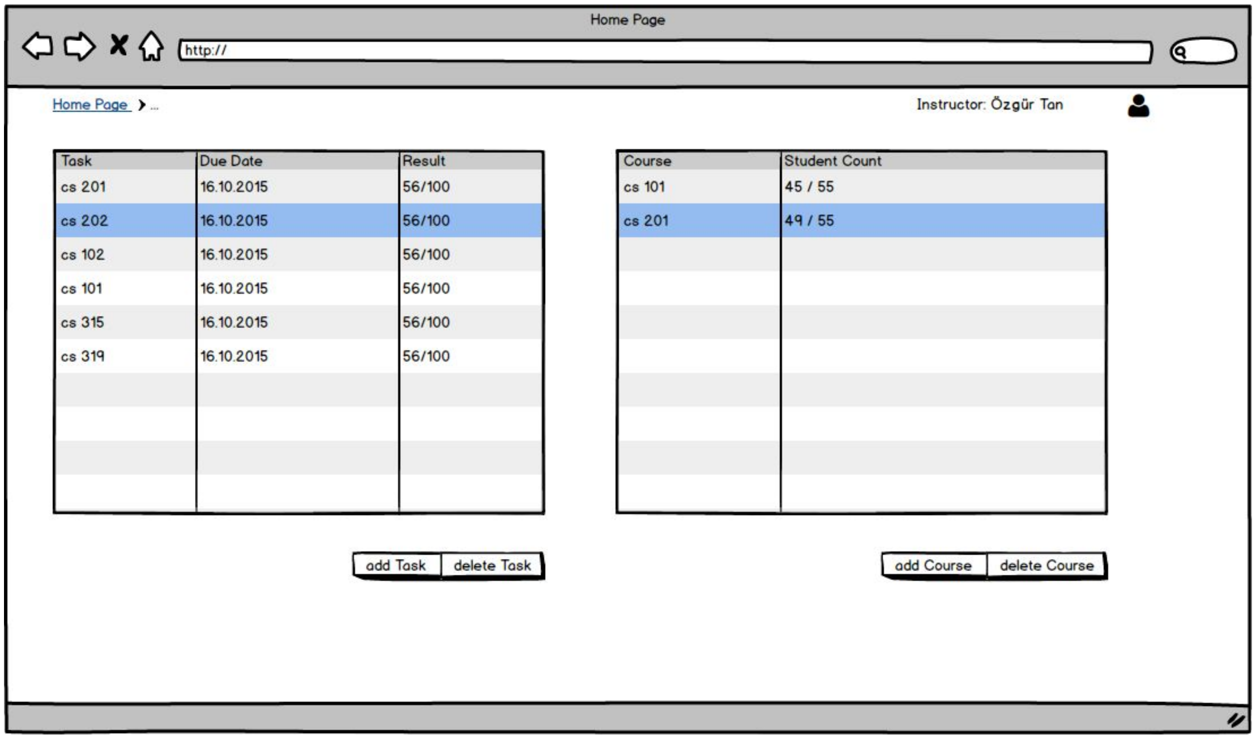
Name	
Description	
last day of Course	20/ 10 / 2015
max student count	
select TAs	☐ Hamzah Agari ☑ Murat Demirbüken ☐ Emre Kurt ☐ Ali Ölmez

2.7.2 Instructor Add Task



Course Code	CS 201
Name	
Description	
Due date	20/ 10 / 2015
Manager TA	Select one of TAs...
Makefile	Browse
Test files	Browse + Add more...
	test1.cpp 20 Points
	test2.cpp 30 Points
Penalties	Late Submission + Add more...
	Memory Leak 40 Points

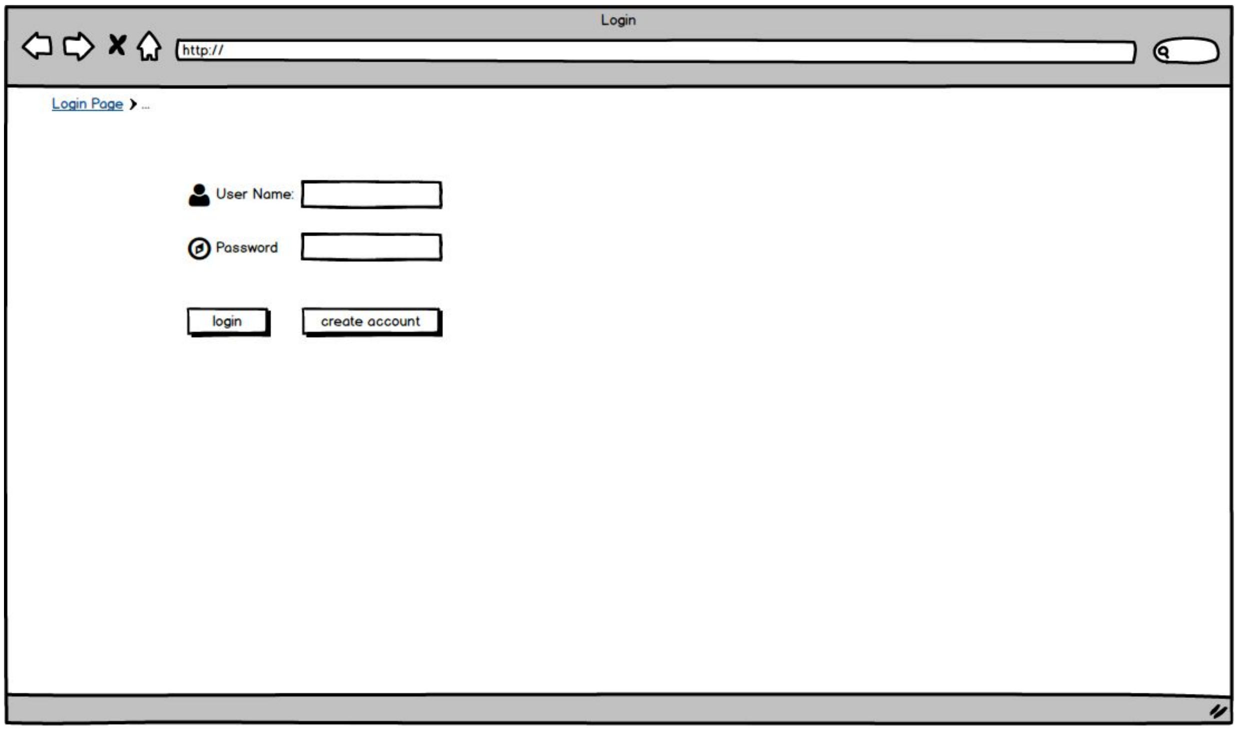
2.7.3 Instructor Home Page



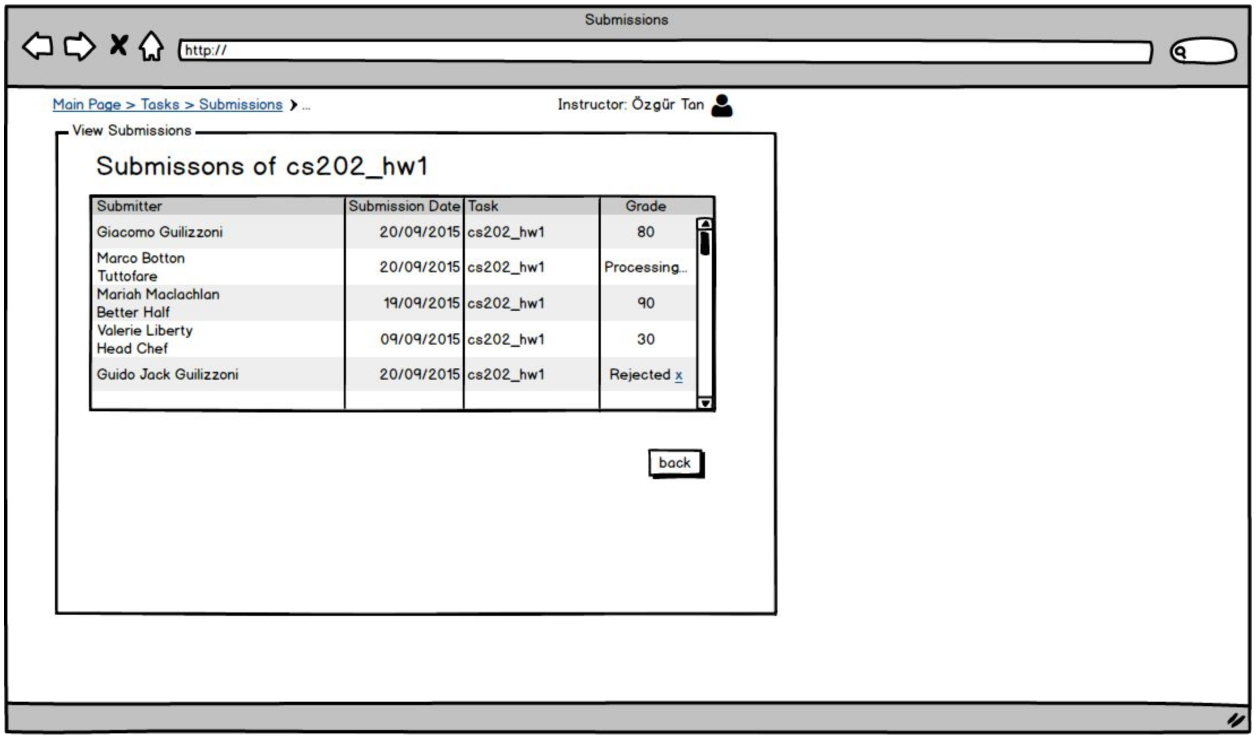
Task	Due Date	Result
cs 201	16.10.2015	56/100
cs 202	16.10.2015	56/100
cs 102	16.10.2015	56/100
cs 101	16.10.2015	56/100
cs 315	16.10.2015	56/100
cs 319	16.10.2015	56/100

Course	Student Count
cs 101	45 / 55
cs 201	49 / 55

2.7.4 Login Page



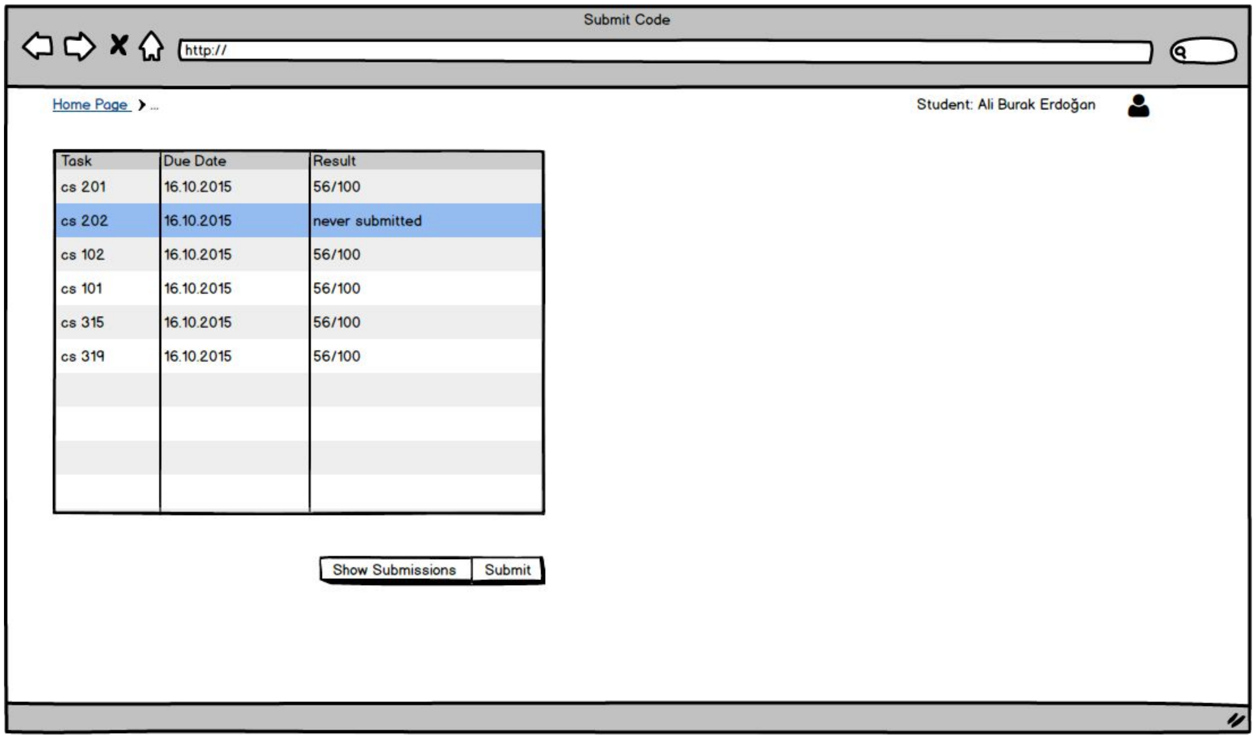
2.7.5 Instructor View Submissions Page



Submissons of cs202_hw1

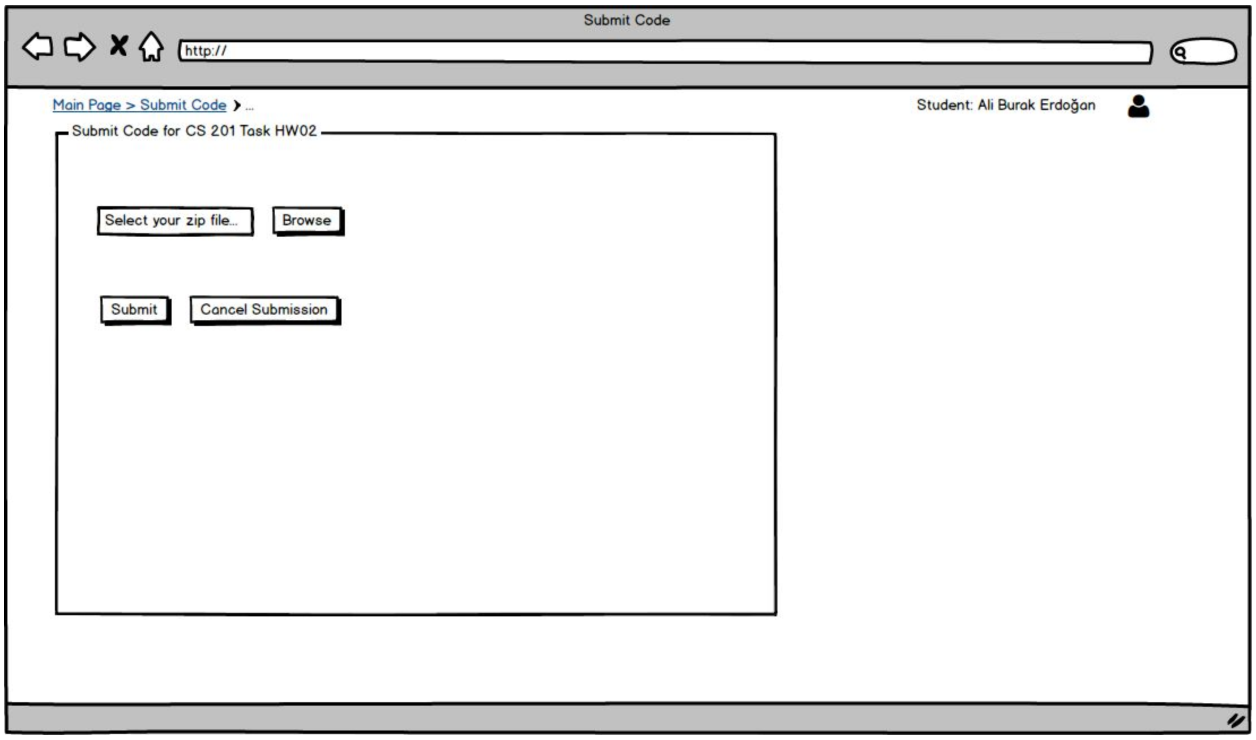
Submitter	Submission Date	Task	Grade
Giacomo Guilizzoni	20/09/2015	cs202_hw1	80
Marco Botton Tuttofare	20/09/2015	cs202_hw1	Processing...
Mariah Maclachlan Better Half	19/09/2015	cs202_hw1	90
Valerie Liberty Head Chef	09/09/2015	cs202_hw1	30
Guido Jack Guilizzoni	20/09/2015	cs202_hw1	Rejected x

2.7.6 Student Home Page

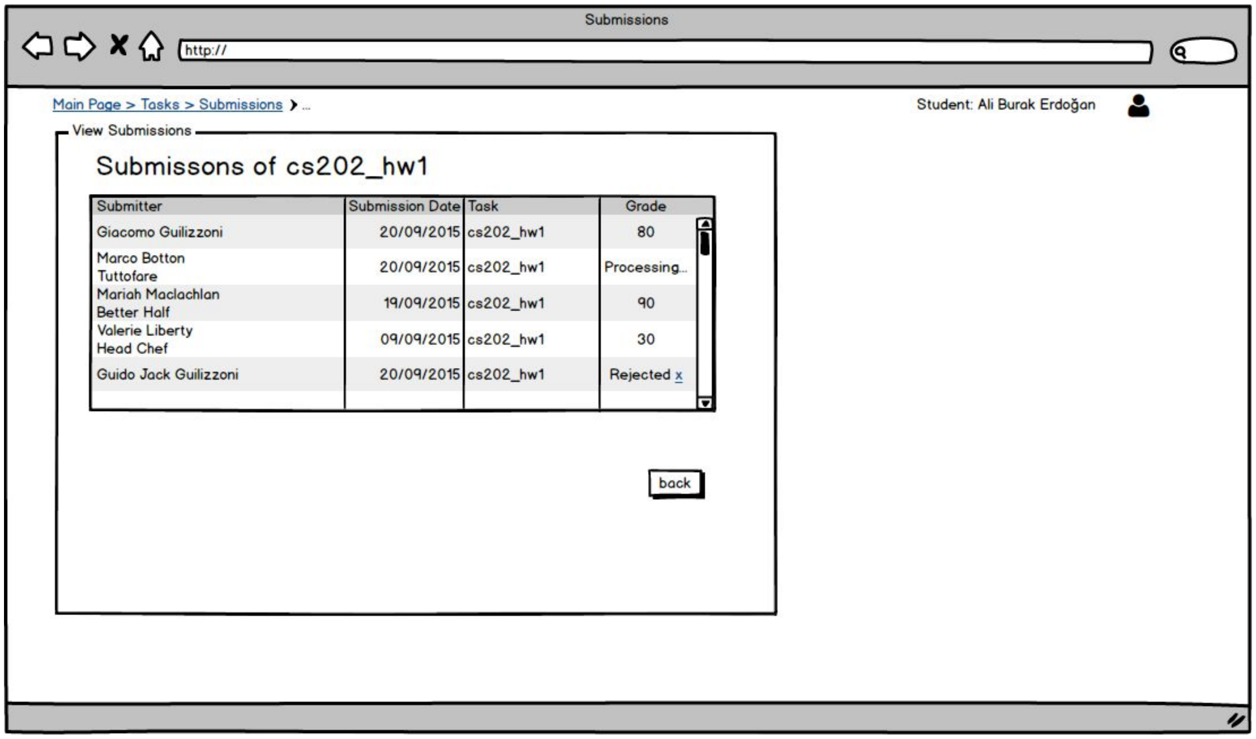


Task	Due Date	Result
cs 201	16.10.2015	56/100
cs 202	16.10.2015	never submitted
cs 102	16.10.2015	56/100
cs 101	16.10.2015	56/100
cs 315	16.10.2015	56/100
cs 319	16.10.2015	56/100

2.7.7 Student Submit Code Page


Submit Code
http://
Main Page > Submit Code > ...
Student: Ali Burak Erdoğan
Submit Code for CS 201 Task HW02
Select your zip file...
Browse
Submit
Cancel Submission

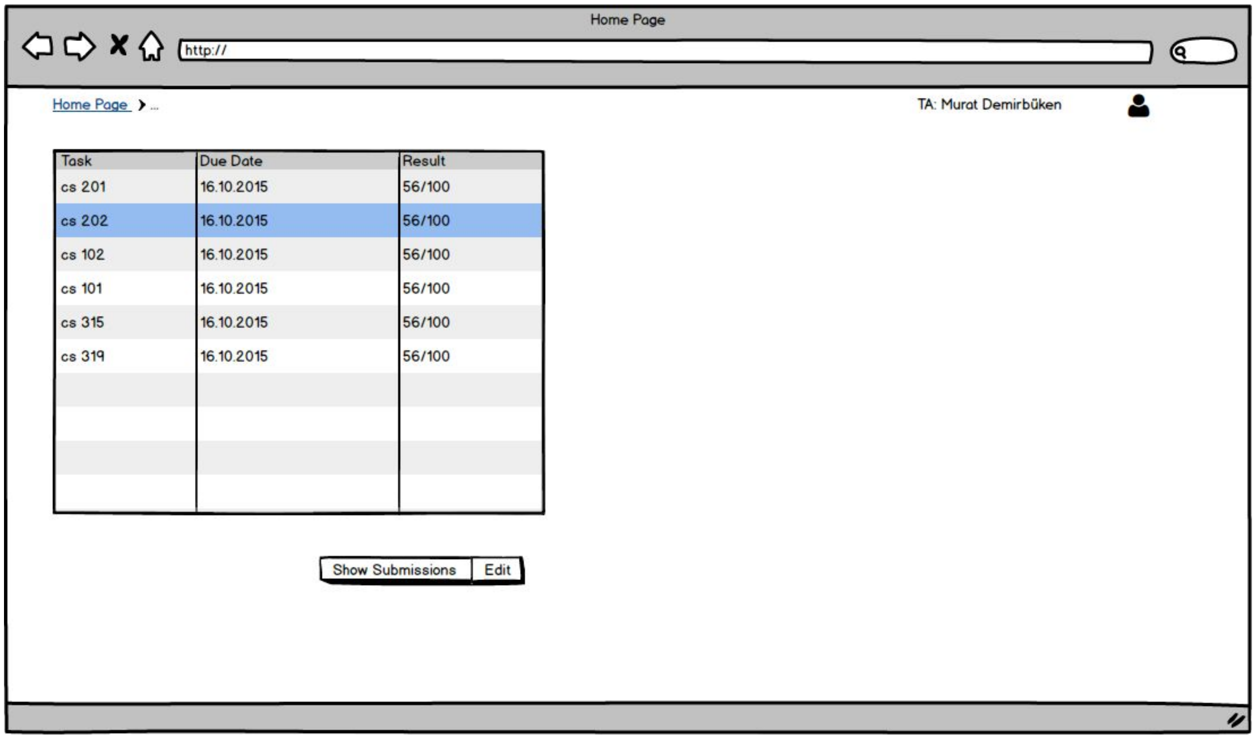
2.7.8 Student View Submissions Page


Submissions
http://
Main Page > Tasks > Submissions > ...
Student: Ali Burak Erdoğan
View Submissions
back

Submissons of cs202_hw1

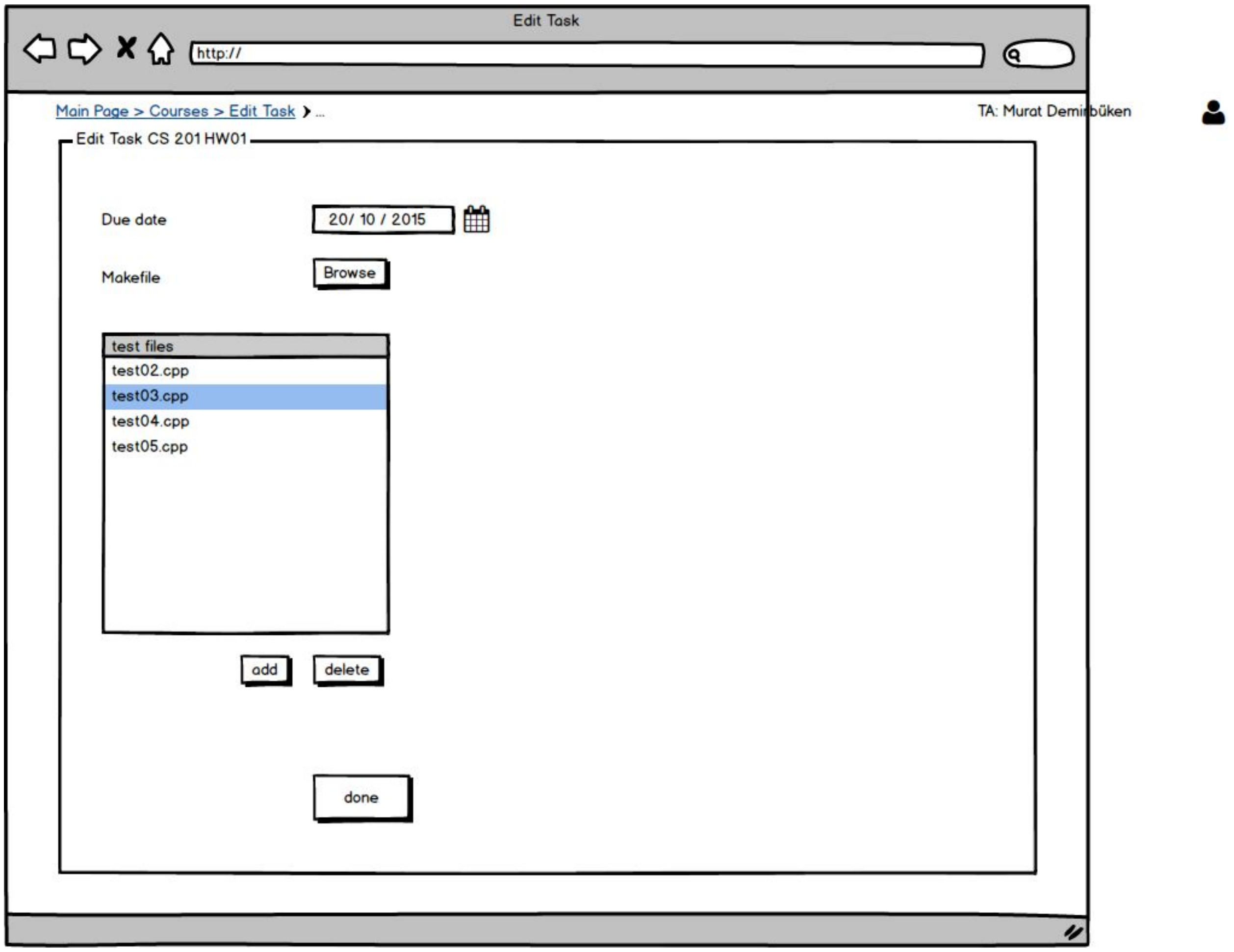
Submitter	Submission Date	Task	Grade
Giacomo Guilizzoni	20/09/2015	cs202_hw1	80
Marco Botton Tuttofare	20/09/2015	cs202_hw1	Processing...
Mariah Maclachlan Better Half	19/09/2015	cs202_hw1	90
Valerie Liberty Head Chef	09/09/2015	cs202_hw1	30
Guido Jack Guilizzoni	20/09/2015	cs202_hw1	Rejected x

2.7.9 TA Home Page

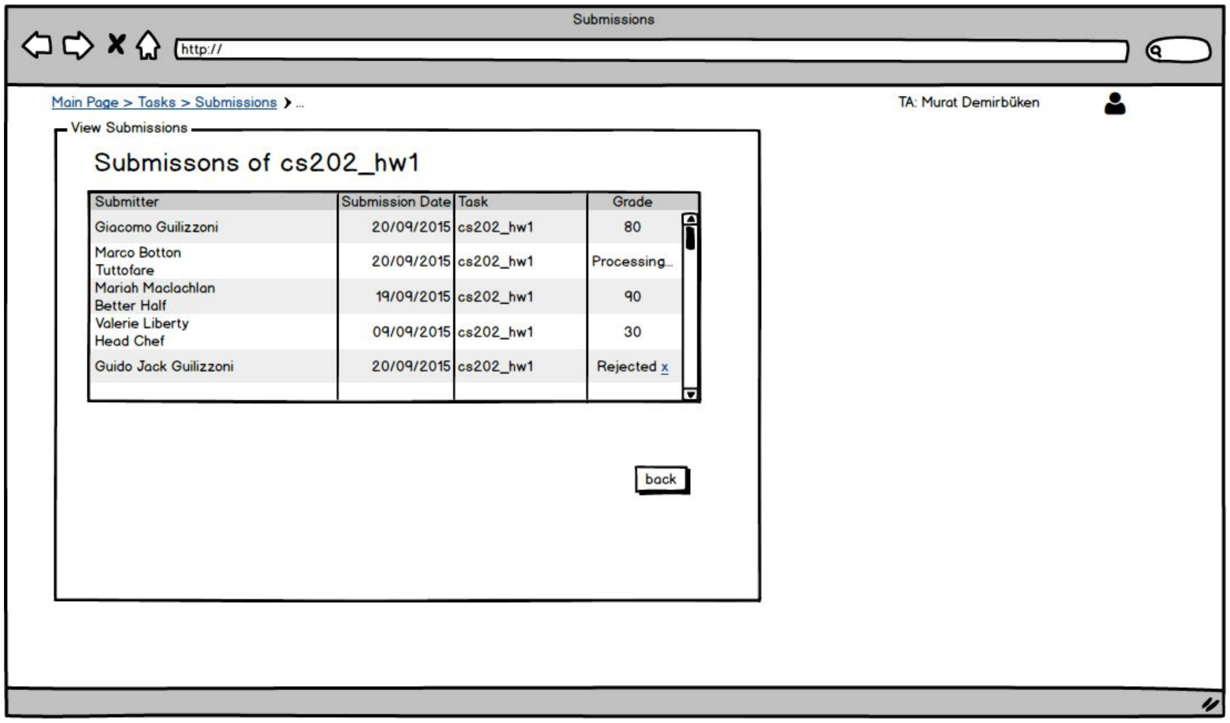


Task	Due Date	Result
cs 201	16.10.2015	56/100
cs 202	16.10.2015	56/100
cs 102	16.10.2015	56/100
cs 101	16.10.2015	56/100
cs 315	16.10.2015	56/100
cs 319	16.10.2015	56/100

2.7.10 TA Edit Task Page



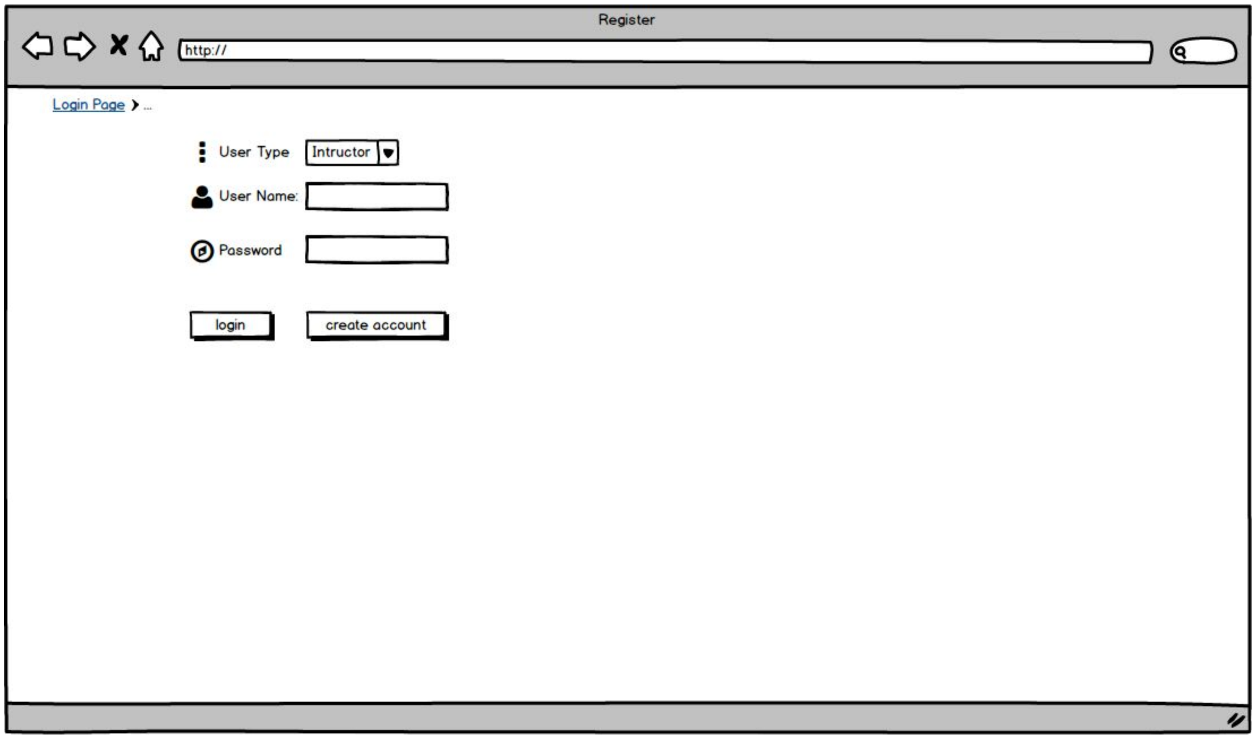
2.7.11 TA View Submissions



Submissons of cs202_hw1

Submitter	Submission Date	Task	Grade
Giacomo Guilizzoni	20/09/2015	cs202_hw1	80
Marco Botton Tuttofare	20/09/2015	cs202_hw1	Processing...
Mariah Maclachlan Better Half	19/09/2015	cs202_hw1	90
Valerie Liberty Head Chef	09/09/2015	cs202_hw1	30
Guido Jack Guilizzoni	20/09/2015	cs202_hw1	Rejected x

2.7.12 Register Page



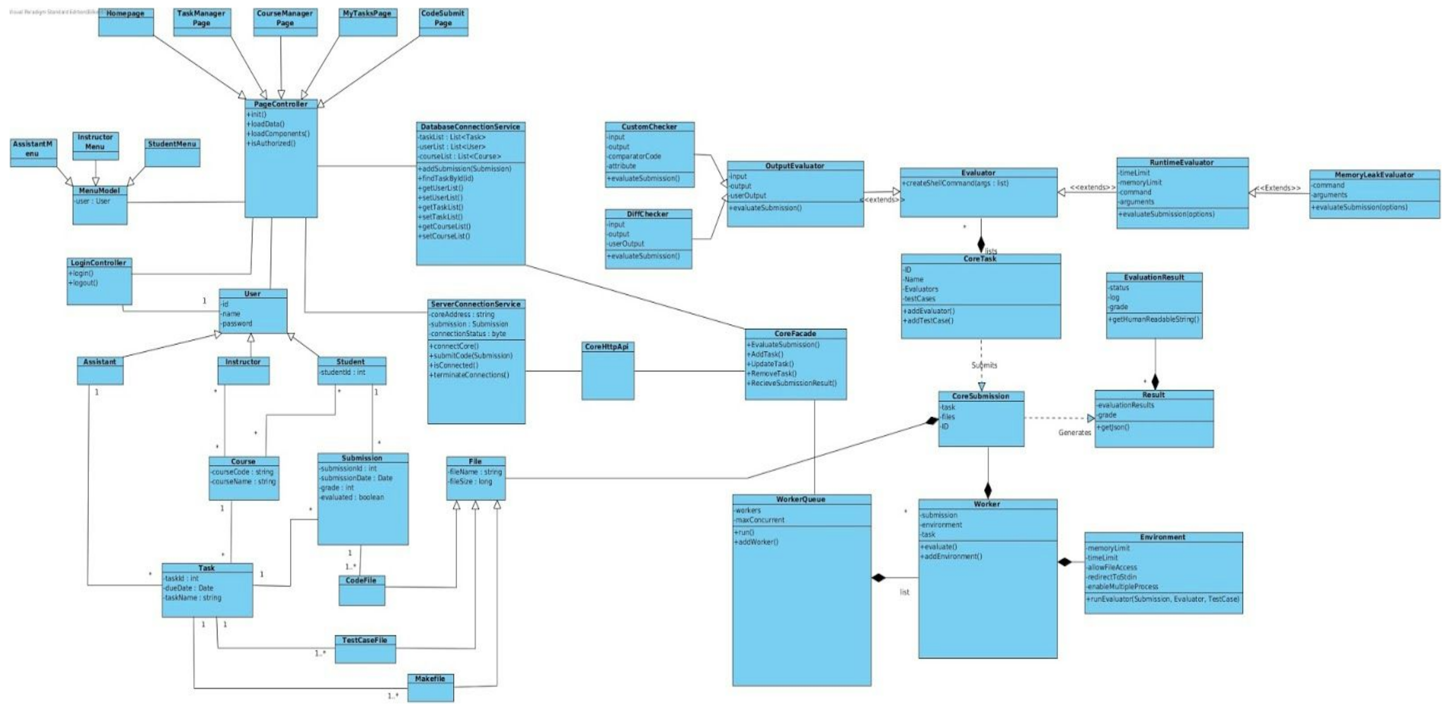
User Type: Intructor
User Name:
Password

3. Analysis

3.1 Object Model

3.1.1 Class Diagrams

Note: Class diagram is small and cannot readable here. We uploaded the bigger version of image to github.


Homepage
TaskManager Page
CourseManager Page
MyTasksPage
CodeSubmit Page
PageController
+init()
+loadData()
+loadComponents()
+isAuthorized()
AssistantMenu
InstructorMenu
StudentMenu
MenuModel
-user : User
LoginController
+login()
+logout()
User
-id
-name
-password
Assistant
Instructor
Student
-studentId : int
Course
-courseCode : string
-courseName : string
Submission
-submissionId : int
-submissionDate : Date
-grade : int
-evaluated : boolean
Task
-taskId : int
-dueDate : Date
-taskName : string
File
-fileName : string
-fileSize : long
CodeFile
TestCaseFile
Makefile
DatabaseConnectionService
-taskList : List<Task>
-userList : List<User>
-courseList : List<Course>
+addSubmission(Submission)
+findTaskById(id)
+getUserList()
+setUserList()
+getTaskList()
+setTaskList()
+getCourseList()
+setCourseList()
ServerConnectionService
-coreAddress : string
-submission : Submission
-connectionStatus : byte
+connectCore()
+submitCode(Submission)
+isConnected()
+terminateConnections()
CoreHttpApi
CoreFacade
+EvaluateSubmission()
+AddTask()
+UpdateTask()
+RemoveTask()
+RecieveSubmissionResult()
CustomChecker
-input
-output
-comparatorCode
-attribute
+evaluateSubmission()
DiffChecker
-input
-output
-userOutput
+evaluateSubmission()
OutputEvaluator
-input
-output
-userOutput
+evaluateSubmission()
<<extends>>
Evaluator
+createShellCommand(args : list)
<<extends>>
RuntimeEvaluator
-timeLimit
-memoryLimit
-command
-arguments
+evaluateSubmission(options)
<<Extends>>
MemoryLeakEvaluator
-command
-arguments
+evaluateSubmission(options)
lists
CoreTask
-ID
-Name
-Evaluators
-testCases
+addEvaluator()
+addTestCase()
Submits
EvaluationResult
-status
-log
-grade
+getHumanReadableString()
CoreSubmission
-task
-files
-ID
Generates
Result
-evaluationResults
-grade
+getJson()
WorkerQueue
-workers
-maxConcurrent
+run()
+addWorker()
list
Worker
-submission
-environment
-task
+evaluate()
+addEnvironment()
Environment
-memoryLimit
-timeLimit
-allowFileAccess
-redirectToStdin
-enableMultipleProcess
+runEvaluator(Submission, Evaluator, TestCase)
1
*
1..*

3.1.1.a User Interface Classes

1.Models:

- User: This is a model class which represents a single user of UI. When a person is signed in to the system, according to the role of the actor, an instance of a child class of User (Student, Instructor or Assistant) is instantiated.
- Task: It represents a single task which is assigned by an Instructor. Holds necessary information and references to the other objects.
- Submission: Instantiated when a Student makes a submission to a specific Task. Holds the code file and necessary informations.
- Course: Stands for representing a Course. It can hold multiple Tasks.
- File: Represents submission files, test case files and makefiles.

2. Views

- MenuModel: This is a class for generating user-specific menus in the interface. This class is inherited by other menu models which are specific to Students, Instructors, and Assistants.
- XHTML View Files: These are not Java classes, but a mix of XML and HTML files. We have separate .xhtml files for each page in the UI. These files contain all the required elements in the views such as buttons, tables, input fields and etc.

3. Controllers

- LoginController: It is a controller class which will be used to handle the authorization management. It holds a reference to the User object which has signed in. This class is also used by other controllers in order to handle authorization.
- PageController: This is an abstract class and inherited by other controller classes which are particular to each view (CodeSubmitPageController, MyTasksPageController, HomepageController, TaskManagerController, CourseManagerController). PageController functions as a bridge between views and models, and is able to manipulate all the models. It has references to the services so that it can communicate with our Database and Remote Server.
- DatabaseConnectionService: User interface makes all the database interaction via this service. Creating, reading, deleting, updating objects in the database is done by this class.
- ServerConnectionService: UI connects to our remote server through this service. This class is used for establishing the connection between User Interface and Remote Server which can receive submitted codes and compile it, then evaluate it according to the preferences. This CoreDispatcher class can make all the required network requests such as connectCore, submitCode etc.

3.1.1.b Core Classes

Core Facade class and rest of the classes, which will together live in a standalone web server, will be accepting https requests of task submissions and will only know enough about tasks that can be used in grading the submissions. Core server will not know about users, and it will be stateless (It will not hold the submission results, it will just return it and forget it). Stateless core standlone web server gives us the opportunity to distribute it on many servers behind a load balancer and adding more servers will be very easy because of our stateless architecture. However it will store task definitions in the same database as UI part. There several classes which will fulfill the core's task description. Evaluators will be runned on the submission according to given topology in task description, their log will be returned in submission object, which will be consist of evaluatorResults. Here is the description of core classes:

- Core Facade: Core facade will be a class which will expose functionality related to evaluation of submissions to the rest of the classes in a structured manner. It will have access to database and worker queue, and provide syncronization between them.
- WorkerQueue: Every submission on the system will be described as Worker class and WorkerQueue class will orchestrate those Worker objects. It will allow us to assign our resources to the tasks in a well defined behaviour and prevent overloading and system crashes.

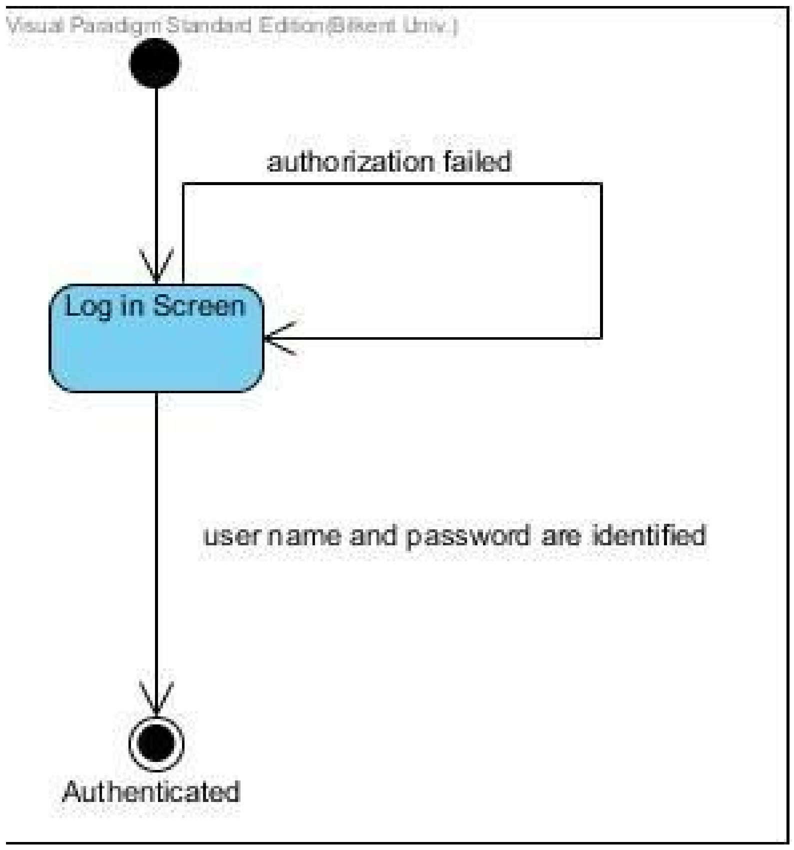
- Worker: Worker class will consist of a CoreSubmission object and an Environment object. When a Worker object’s evaluate method is called, it will evaluate submission in given Environment.
- Environment: Environment class will hold resources and access to an isolated system, in which submissions can be compiled, tested and graded Environment class will provide a secure sandbox in which user submissions can be evaluated without interfering the system. Also not accessing system directly, but by a shield class will give us an opportunity to improve security features quickly without changing other parts of the code
- CoreSubmission: CoreSubmission Class is similar to Submission class. In a way it is a proxy of the Submission class. However it will only hold the necessary information about evaluating the code, it will not hold any information about the student who submitted the code, or deadlines.
- CoreTask: Similar to CoreSubmission, CoreTask is a proxy of Task Class. Again, similar to CoreSubmission, it will not hold any other information about task other than how to grade codes that submitted to given task. We used this two proxy classes in order to have a clean, understandable core package.
- Evaluator: An Evaluator is a class that will grade the submission for a given task and return the result. Evaluator will work on given Environment. There are two types of evaluators: RuntimeEvaluator and OutputEvaluator.
- RuntimeEvaluator: RuntimeEvaluator will receive shell command which will be executable over given input and output format and will produce result on

them. There is one child of RuntimeEvaluator currently, MemoryLeakEvaluator which will tell if users submission has a memory leak given a test case.

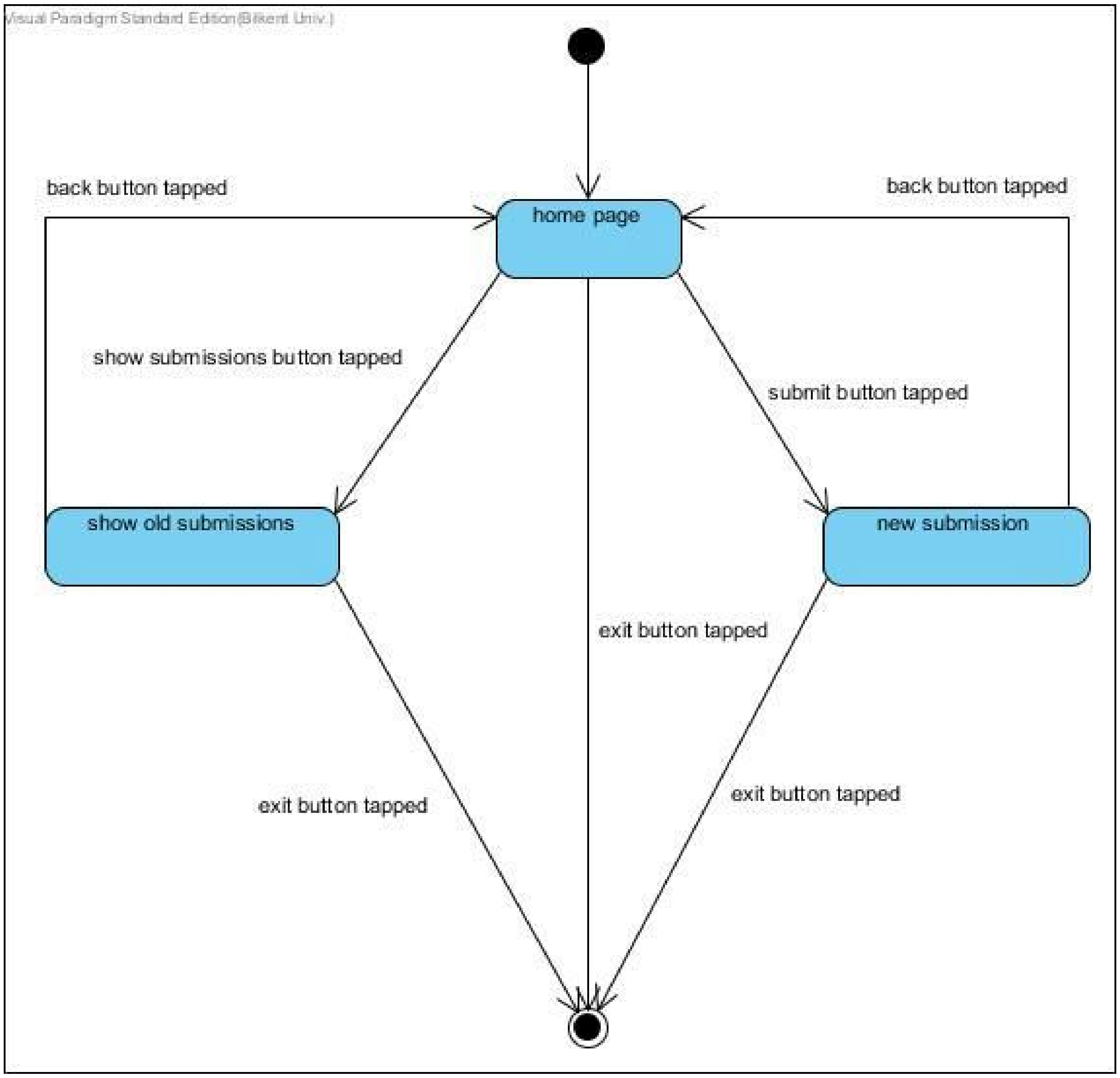
- Result: Result Class will hold the overall evaluation result for a submission and a task. It will have necessary transformation functions which will help getting a structured response, whether to save or display it to the user. It will hold the list of EvaluationResult objects.
- EvaluationResult: Every Evaluator class will return an EvaluationResult object which will hold status about evaluation, whether evaluation is completed without any errors or not, grading and logging. It will also return an human readable string function.

3.2 Dynamic Model

3.2.1 Statechart Diagram


Visual Paradigm Standard Edition(Bilkent Univ.)
authorization failed
Log in Screen
user name and password are identified
Authenticated

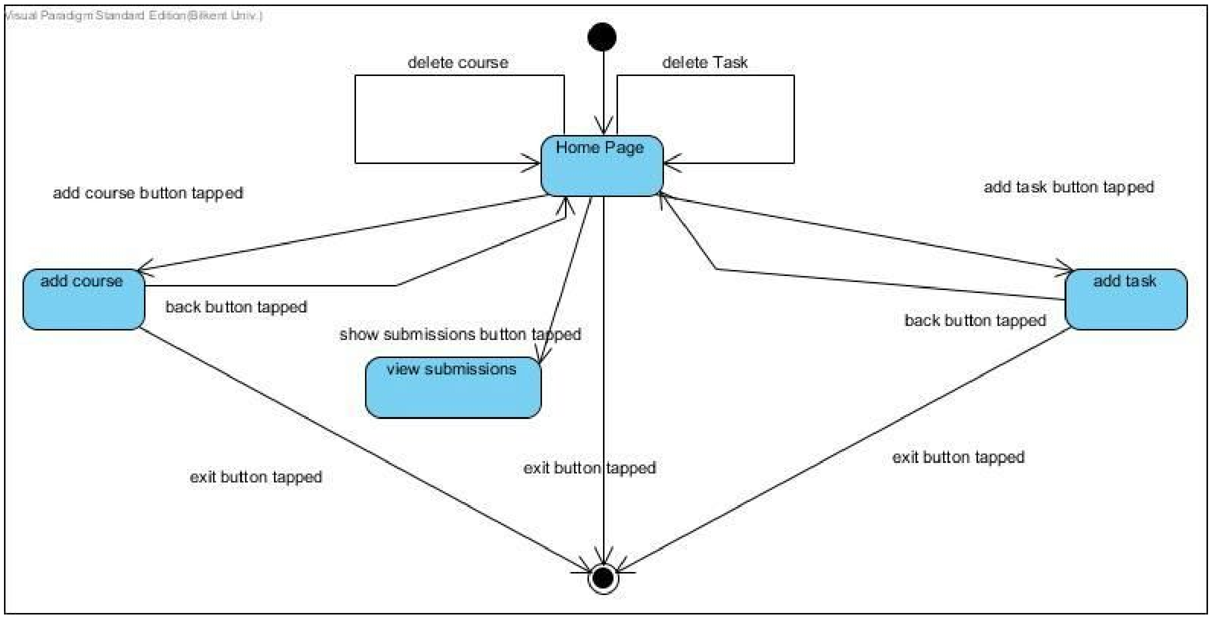
This diagram shows the standard procedure for logging in to the system. This is common to for all types of users. (TA, instructor, student)



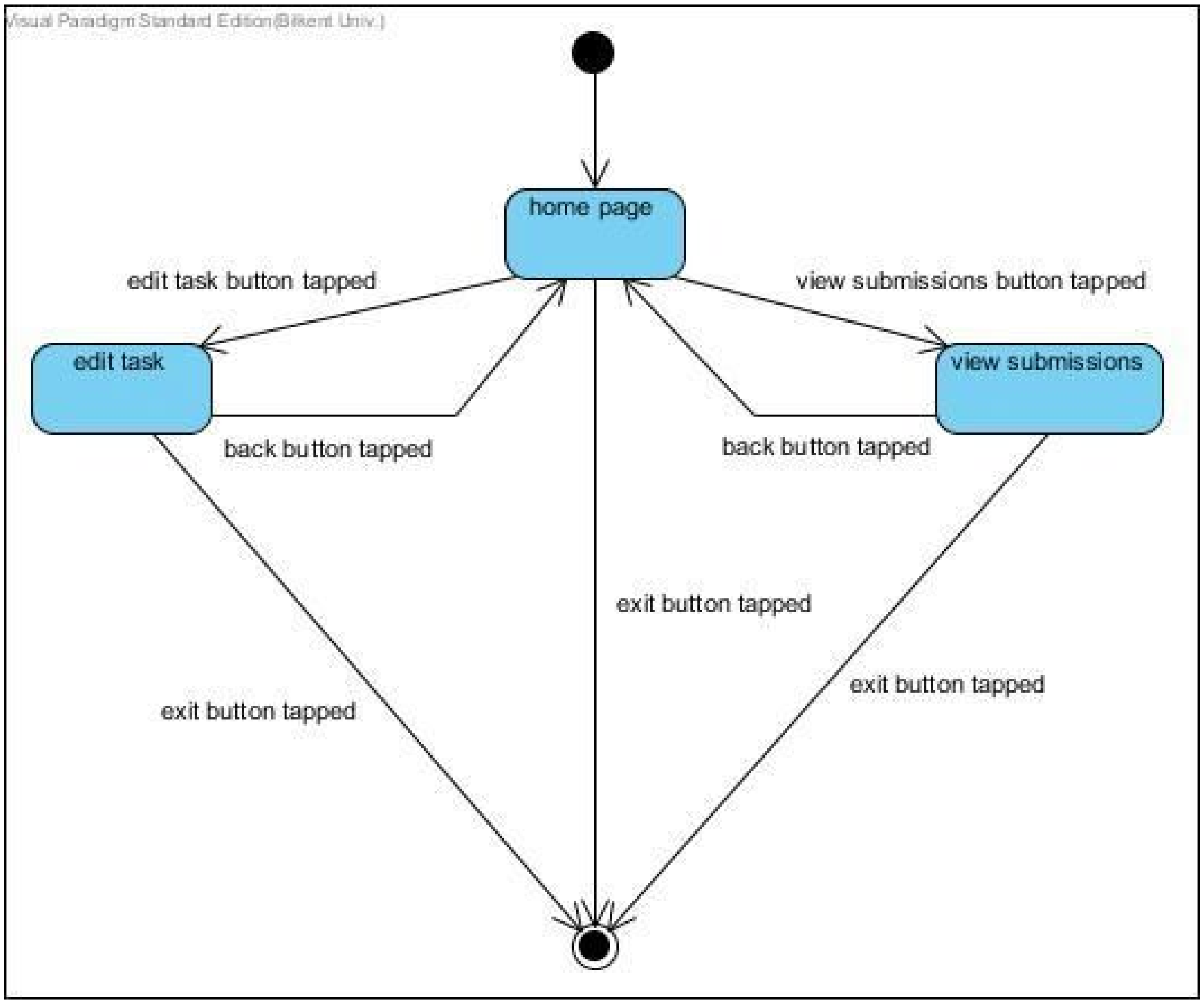
This diagram shows the interactions of the student in Grader++. Basicaly student can see his old submissions and can generate a new submission.

To create a new submission student must have selected a task. After selecting a task student could make a new submissions for the task. After tapping submit button, new submission screen will pop up.

To see the old submissions student must selected a task. After selecting a task student could see the old submissions for the task. After tapping view button, old submissions screen will pop up.



This diagram shows the interactions of the instructor in Grader++. Basicly instructor can add course, delete course, add task, delete task and view submissions. To delete a task instructor must be selected a task. To delete a course instructor must be selected a course. Add task and add course options will pop up appropriate screens. Instructor able to see old submissions



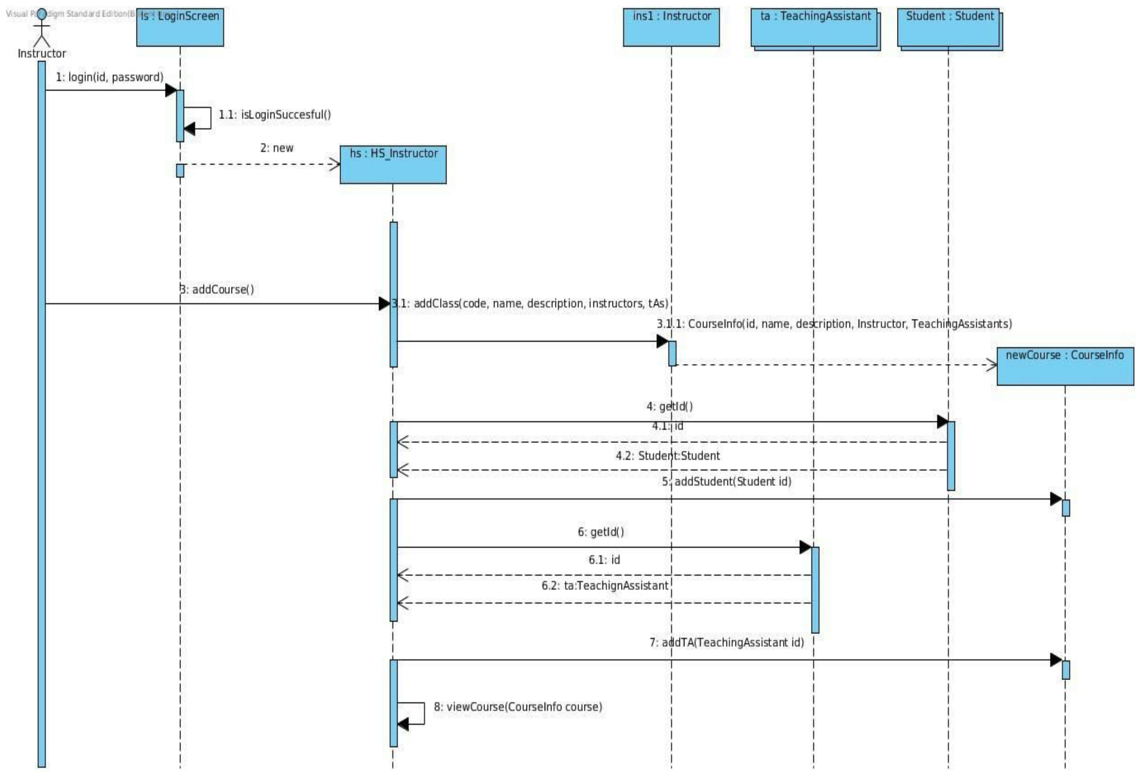
This diagram shows the interactions of TA in Grader++. Basicly TA can edit task and view submissions. To edit a task TA must be selected a task. To view submissions TA must be selected a task. After selecting task TA could see the submissions for that task.

3.2.2 Sequence Diagrams

After user accesses the login screen and successfully enters his/hers id and password the system will provide the appropriate homepage depending on the type of user (instructor, teachign assistant or student). Each user is allowed to do spesific operations on client, for example instructors can add courses to system, add tasks(homeworks) for a course, and students can submit their homeworks through client. Among these operations, significant ones' sequence diagrams are provided below.

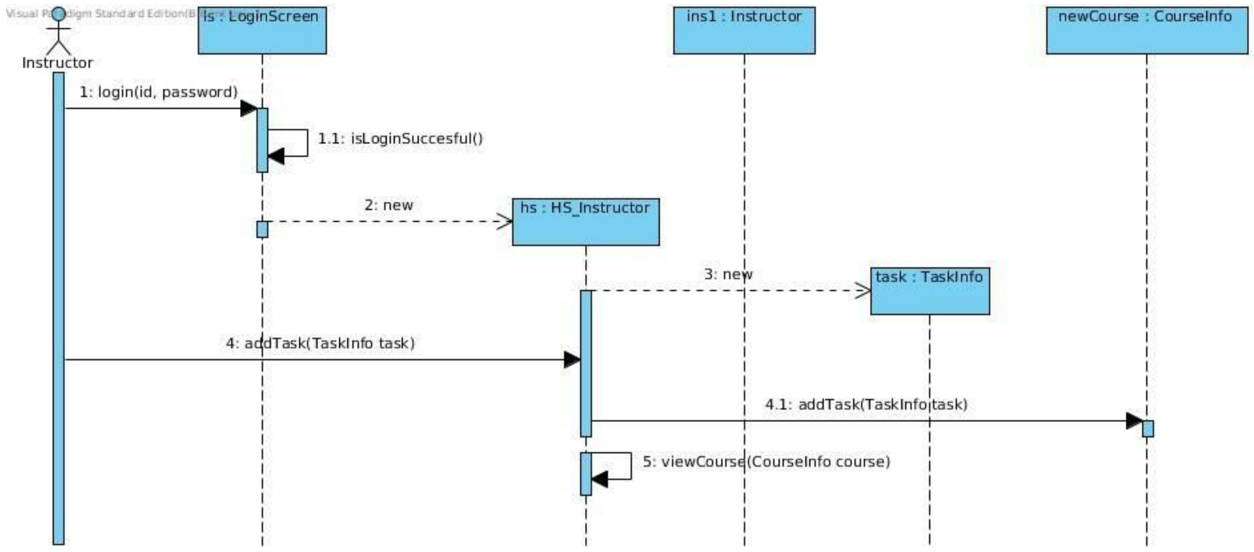
3.2.2.a Add Course

Instructors are the only ones allowed to add courses to system. So, the user should login to the system, and after it is confirmed that user is of type "Instructor" client will provide user with homescreen for instructors. User is provided with add course button in the homescreen, and when he/she clicks on it Instructor class will construct a new CourseInfo object that with the properties indicated by user. After this process user will be able to get the information of students and teaching assistants so that they can be subscribed to the course.
Once these processes are over client will provide user with the viewCourse screen as default.



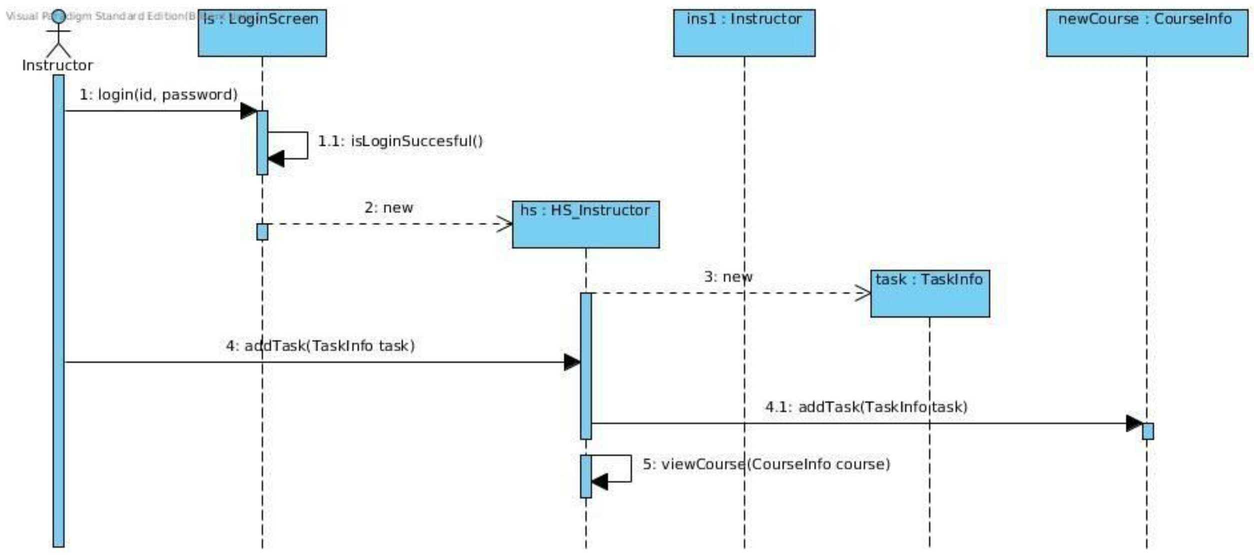
3.2.2.b Add Task

Instructors can add tasks (or homeworks) as part of a course. After the login process is completed and the appropriate homescreen is provided to instructor, the button for adding task will be visible to user. Following the click on the button there will be a new TaskInfo object constructed according to the specifications indicated by instructor. Then, this newly created task will be added to the appropriate CourseInfo instance.



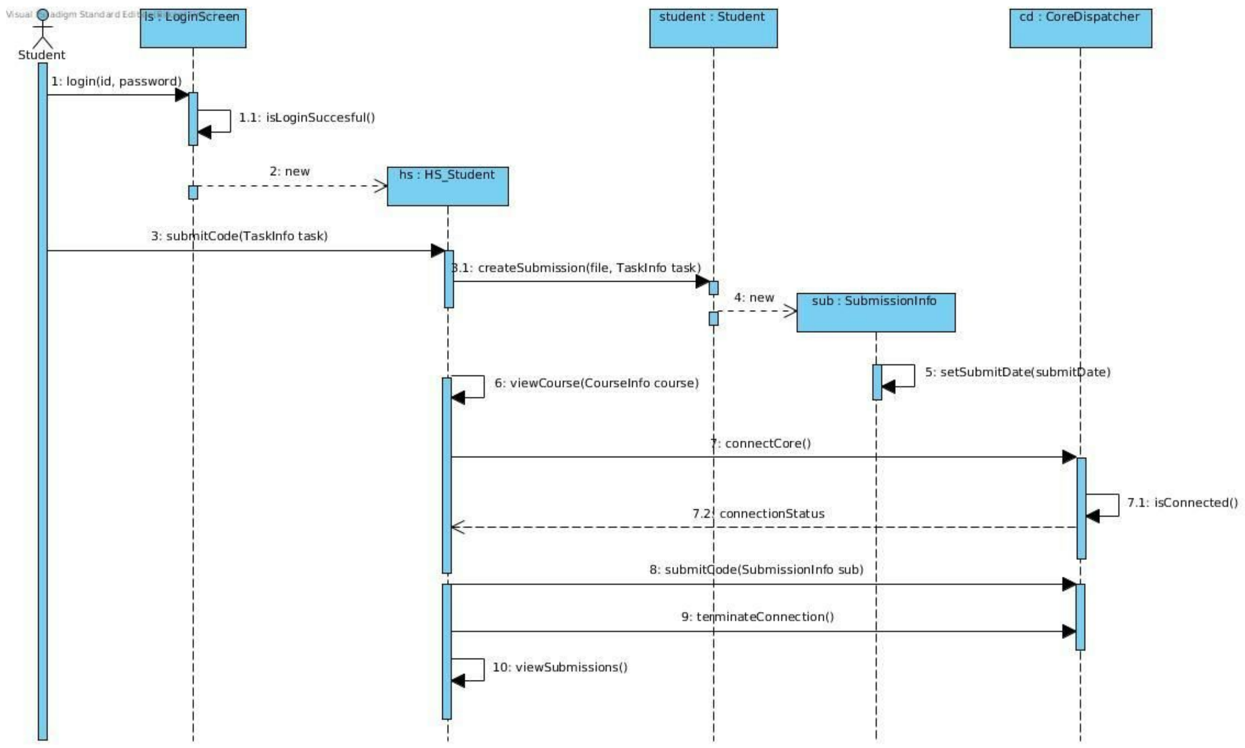
3.2.2.c Edit Task

In some cases instructor may need to edit task info, for example the test inputs may had been initially wrong and instructor would want to change the wrong inputs with the correct ones, or the responsible TA might want to try his own test inputs. In order to do this user should login, and once the homescreen is ready he/she should select a particular task and click on Edit Task button. After the button is clicked there will be a new TaskInfo object constructed with respect to the attributes indicated by user, and then the copy constructor of the original TaskInfo object will take the new TaskInfo as it’s parameter. Hence, the task will be edited.



3.2.2.d Submit Code

In order to submit code to the system students should first login, and then click the submit code button. This will trigger the createSubmission() method of Student object and it will eventually create a SubmissionInfo object and deliver it to the core (evaluator) for analysis and grading.



4. Design

4.1 Design Goals

Before we start the design and implementation phase, it is important to decide main design goals for the system according to the mentioned non-functional requirements. In this section we provide the quality features that we decided to which system design should focus on.

4.1.1 End User Criterias

-Ease of use: Web client that grader++ should provide to users must be designed in a way that there will not be any unnecessary pages that users should visit in order to execute simple tasks. For example, students will not be needing to visit a new page for viewing the list of courses they are registred to, the list will be provided on the homepage. This approach will lead the UI to be simple and end-user friendly.

-Ease of learning: Web client should not include any terms or concepts other than users already know from their academical life, so that there won't be need for any extra effort on users' side to get familiar with using web client.

4.1.2 Maintenance Criterias

-Portability: It is important that users can access and interact with grader++ easily. Because of this fact, we decided to implement the user interface as web site, so that users can access grader++ from their computers, tablets, and smart phones.

-Extendability: Grader++ will be easily extandable, since we are planning our system to be used by not only for one course. New courses with new needs of different evaluator systems should be easily added, since we decided to implement a modular system for evaluators and their work flow in core server.

4.1.3 Performance Criterias

-Response time: Grader++ should have minimium response times. It is important that users can see the results of evaluation progresses for their submissions, so that they can get the feedback on their submissions and can be aware of any errors they made, and eventually have enough time to do modifications before due dates of homeworks.

4.1.4 Dependability Criterias

-Availability: It is high likely that there will be significant amount of loads near homework due dates. In order to overcome this work load we divided the system and do evaluation on a seperate server named "core". Core should evaluate submissions as concurrently as possible, so that system can respond to as many people as possible at the same time.

-Security: There will be need of security measurements for core server, since one of it’s possible job is to compile codes and execute it there might be malicious attempts to hack the system. In order to prevent these situations, there should be container systems such as Docker.

4.1.5 Trade-offs

-Ease of use and learning vs. Functionality: In order to achieve friendly user interface system should include only necessary ui components to provide users with. Our aim should be on the side of ease of use, however we should make every necessary action on the application domain available for users.

-Response times and Availability vs. Deployment cost: In order to have a high percentage of available time there might be need for relatively expensive servers. In order to solve this issue, core should be coded as optimized as possible in implementation phase.

4.2 Subsystem Decomposition

There are 2 distinguishable main parts of Grader++ subsystem: The client part, which will generate output which will be consumable by browsers(html, css, javascript), and is responsible for coordinating Users when adding tasks, submitting code and viewing results, and core server part which will be responsible of evaluating submitted codes in an uniform and secure sandbox environment, and lastly. Also client and core server part will share storage in order to communicate submission results.

For the client part, we are using MVC pattern in which GraderppWeb subsystem provides a front end for users to initiate all use cases (e.g. EditTask, SubmitCode, ViewSubmissions) as a View part in MVC. The GraderppServer subsystem is responsible for access control and initiates the Models, hence the Controller.

Different subsystems are dedicated to the models of users, tasks, submissions and synchronizing submission files and test case files to the core server and will update Views as the Model part. Also Model subsystem will be persisted by GraderppStorage subsystem, responsible for storing any persistent objects, it will use filesystem for storing submissions and test cases and MongoDB for storing everything else. Fig 4.2.1 shows client part component diagram.

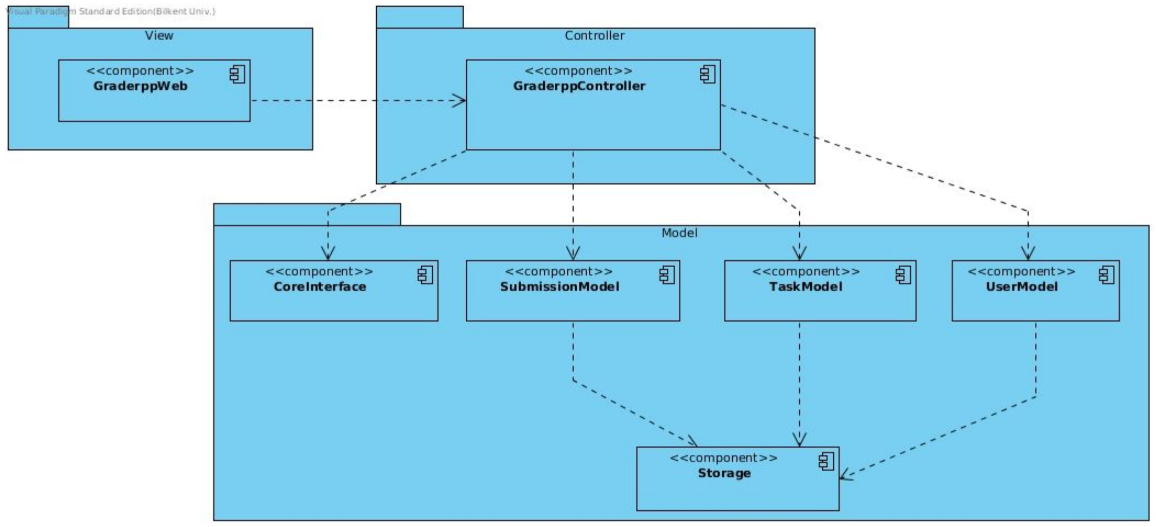


Fig 4.2.1

For the core server part, we select three-tier architectural style in which an CoreClient subsystem will hide all the complexity behind core server and offer a clean api for the client part as a front-end which provides functionality for submitting code, task descriptions and recieving feedback. The CoreWorker subsystem will orchestrate the grading operations in a queue manner, Evaluator Subsystem will determine how the tasks will be graded, and together they will form Application Logic layer. Bottom tier will be realized by CoreStorage subsystem, responsible for storing submitted codes and test cases and submission results, and Environment subsystem which will provide a secure environment for Evaluator to assess the submissions. Fig 4.2.2 shows core server component diagram

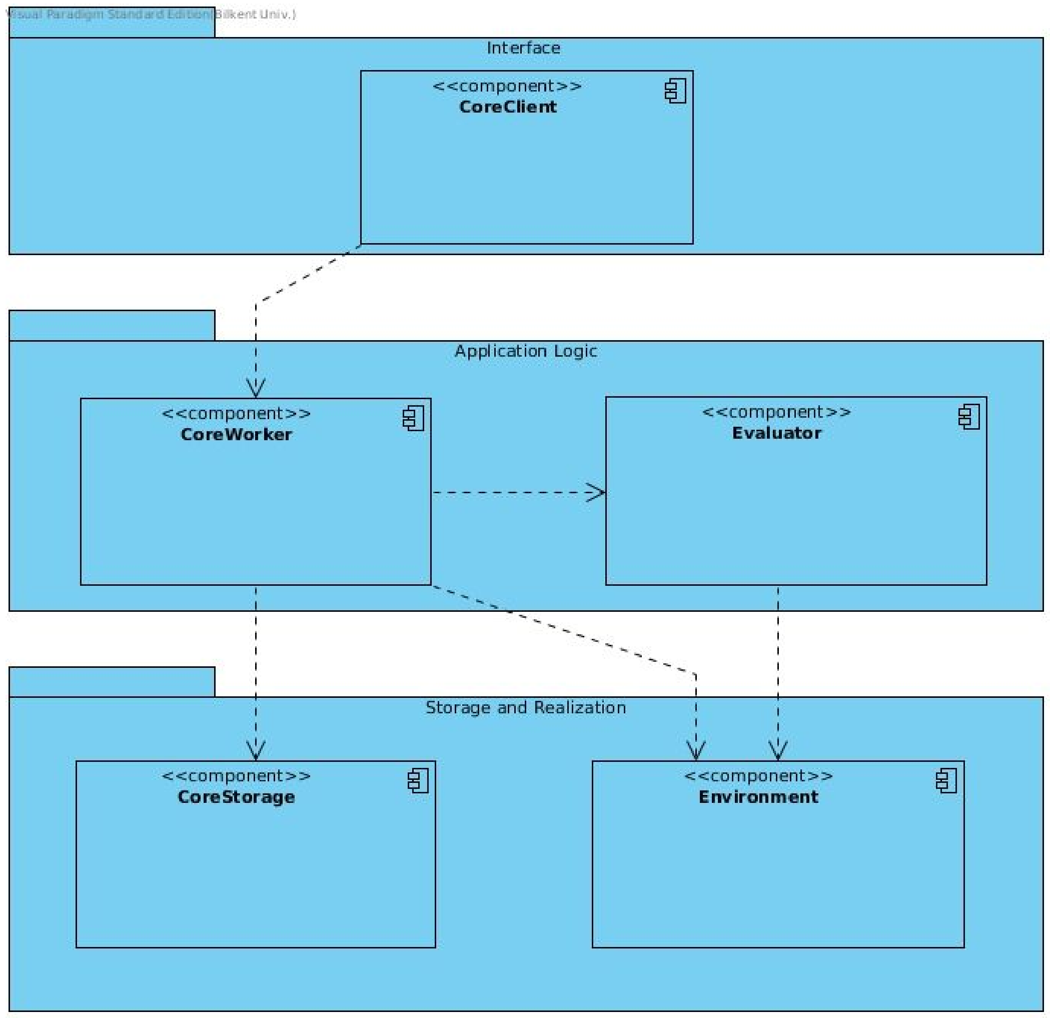


Fig 4.2.2

4.3 Architectural Patterns

4.3.1 MVC Pattern

MVC (Model-View-Controller) pattern is one of the most fundamental design patterns in our project. Its main goal is to make a clear separation between Models as plain data, Views as visualization of the data and Controllers which makes the control flow between models and their views. In our client subsystem we used MVC pattern.

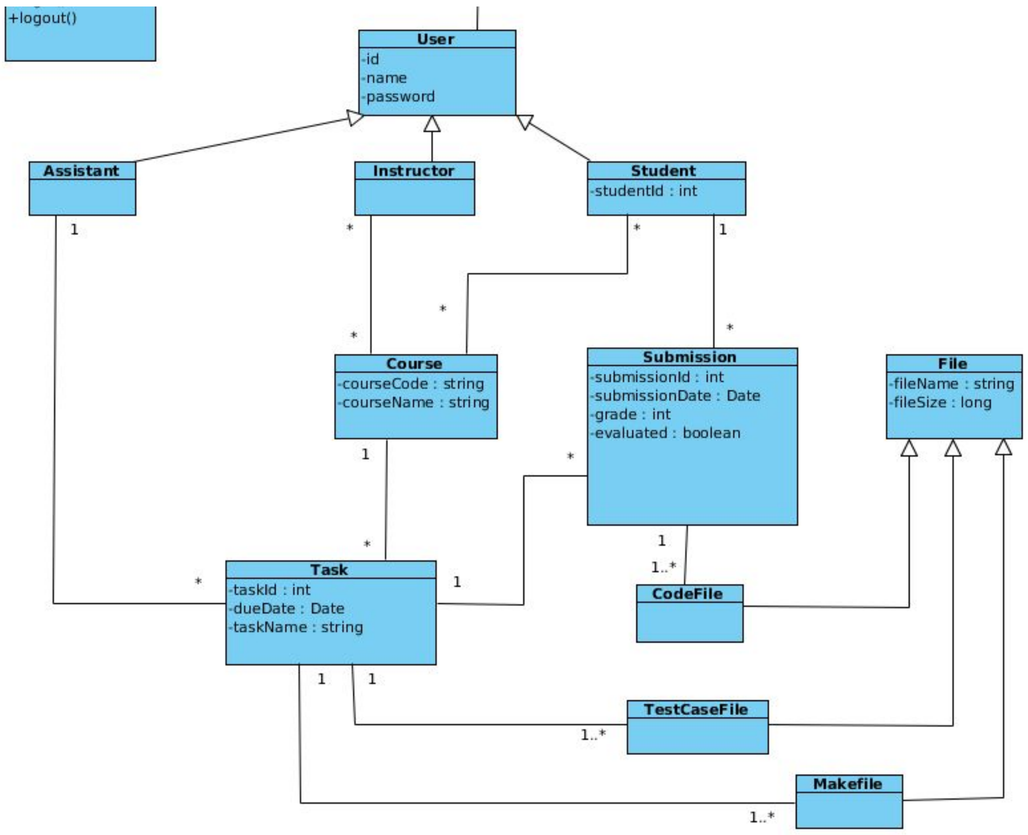


Figure 4.3.1.1. Model classes

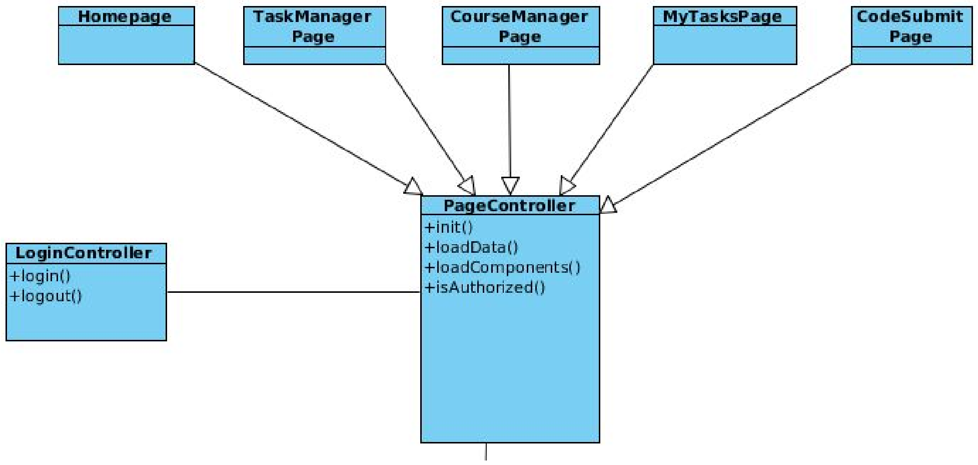


Figure 4.3.1.2. Controller classes

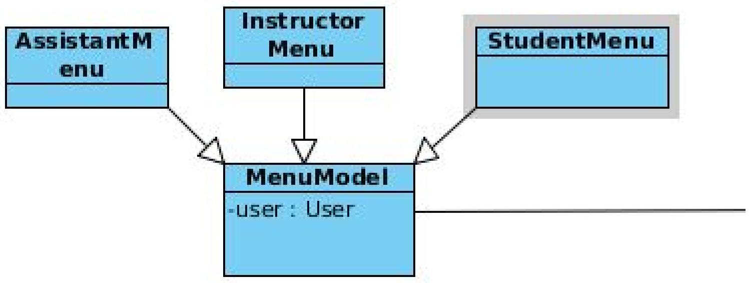


Figure 4.3.1.3. Some of View classes

In the client side of our project we use many View objects (MenuModel, XHTML files etc.) for providing visible web pages to user; Model objects (Course, Submission, User, Task etc.) to represent data; Controller objects (Homepage, CodeSubmitPage, TaskManagerPage etc.) to make all the operations between those two.

4.3.3 Data Access Object (DAO) Pattern

Data Access Object Pattern is used to make operations on database with the POJO model classes (Plain-old Java Objects). Data operations can be simplified by separating low level details from high-level uses. We have DAO classes for each of our models and we can make CRUD operations (Create, Read, Update, Delete) on the database with the help of them.

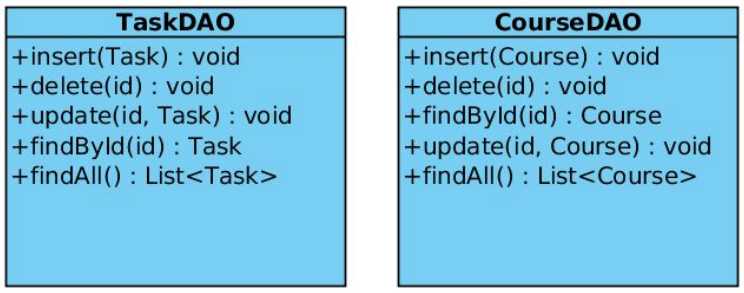


Figure 6. Data Access Objects (DAOs) for some of our model classes

4.3.3 Three-tier Architectural Style

In core part of our system decomposition we used three layered architecture. Interface layer is provided by CoreFacade class, Application logic layer have two classes: CoreWorker and Evaluator. CoreWorker orchestractes evaluations and Evaluator will do the work on Environment given by CoreWorker. Bottom tier will be realized by CoreStorage which will persist task data and Environment subsystem will use system resources to realize evaluations. Following figures, 4.3.7-10 will show which class is belonging to which layer in our design.

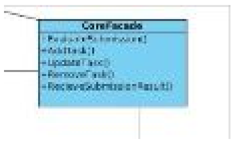


Fig 4.3.7 Core Facade Class (Interface Layer)

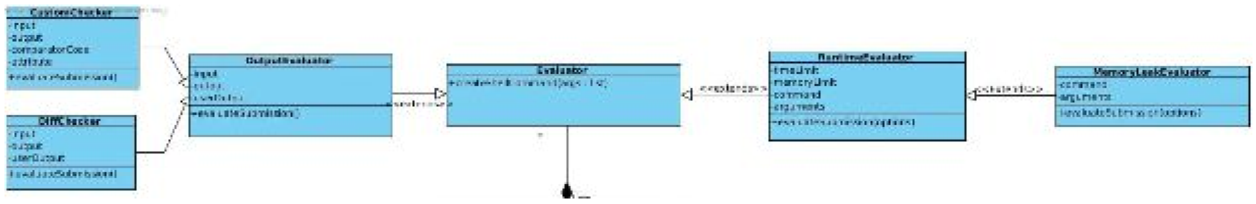


Fig 4.3.8 Evaluator Subsystem (Application Logic)

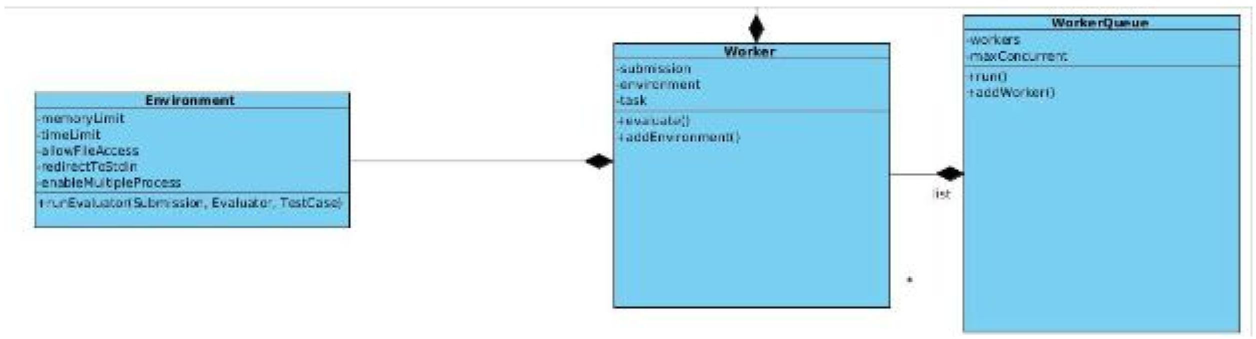


Fig. 4.3.10 Worker Subsystem classes (Application Logic)

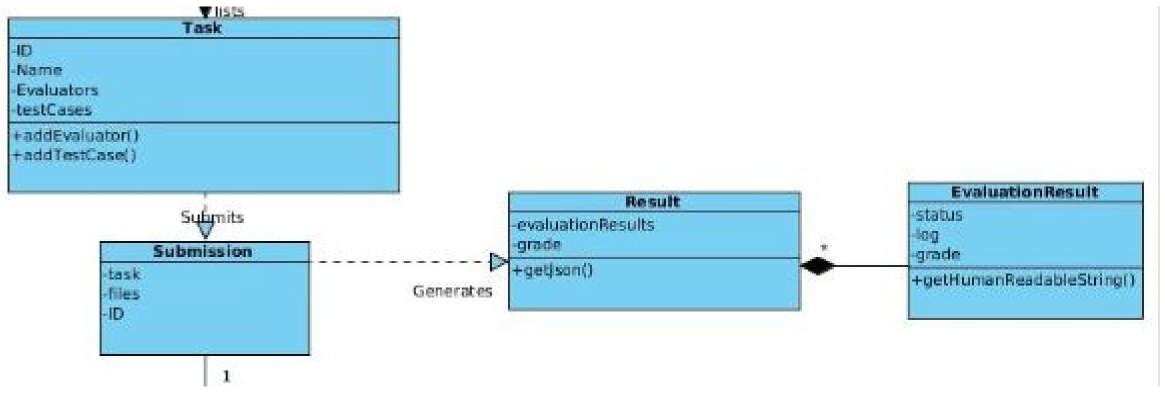


Fig. 4.3.9 Storage Classes (Bottom Tier)

4.4 Hardware/Software Mapping

Mapping subsystems to processors and components enables us to identify potential concurrency among subsystems and to address performance and reliability goals. In Grader++ project, we use three servers, one for ClientController subsystem and one for core part and one for both storage parts of client part and core part. For the Environment subsystem we plan to use Docker (https://www.docker.com/), in order to provide security design goal. All of the submissions will be run in different docker environments and since docker is very lightweight, it won't make us compromise our performance goal. In core part, CoreClient will providehttp interface through an undecided web server container like Tomcat or Glassfish. Fig 4.4.1 shows our deployment diagram.

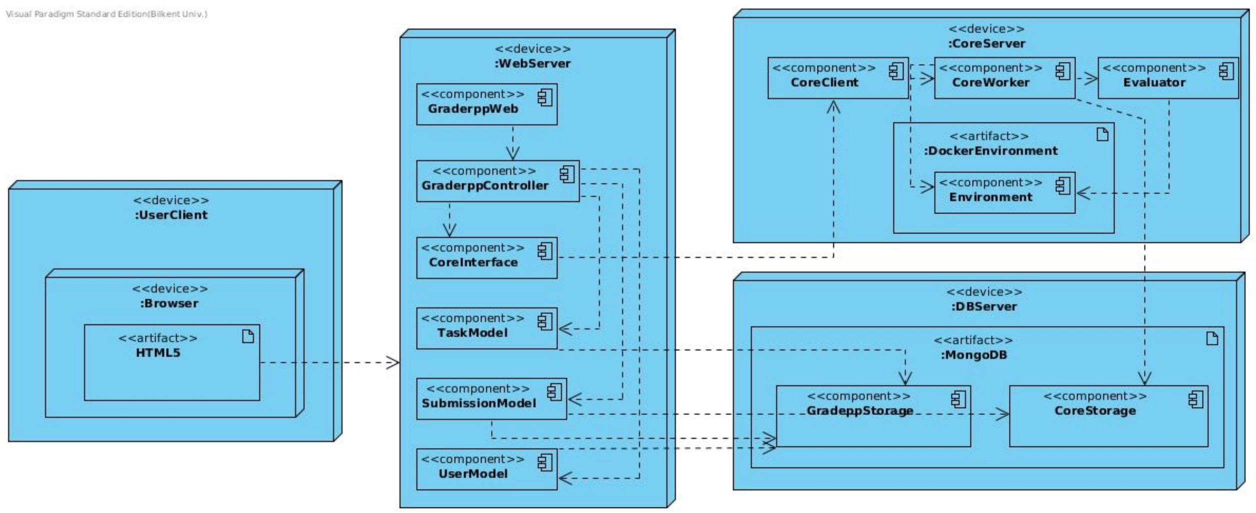

Visual Paradigm Standard Edition(Bilkent Univ.)
<<device>> :UserClient
<<device>> :Browser
<<artifact>> HTML5
<<device>> :WebServer
<<component>> GraderppWeb
<<component>> GraderppController
<<component>> CoreInterface
<<component>> TaskModel
<<component>> SubmissionModel
<<component>> UserModel
<<device>> :CoreServer
<<component>> CoreClient
<<component>> CoreWorker
<<component>> Evaluator
<<artifact>> :DockerEnvironment
<<component>> Environment
<<device>> :DBServer
<<artifact>> :MongoDB
<<component>> GradeppStorage
<<component>> CoreStorage

Fig 4.4.1 Deployment diagram

4.5 Addressing Key Concerns

4.5.1 Persistent Data Management

Data management is one of the significant parts of the system. Grader++ requires storing lots of data such as user information, tasks, courses and submissions. These data must stay persistent, and for this purpose we chose to use database systems instead of using file system. Using a database system can make system maintainable and easy to operate. Data should be accessible by multiple users such as students, TAs, and instructors. We decided to use MongoDB database system because of it's ease of the fact that it is a lightweight system. In our database, many types of persistence data are used. First one is for storing personal information of users. In order to use Grader++, users must log in. Therefore, their personal information such as user name and password are stored in database. Database should also store courses, tasks and submissions. For submissions, students will be allowed to submit only specified amount of submissions for a certain task.

4.5.2 Access Control and Security

There are 3 different types of actors anticipated for using Grader++: students, TAs and instructors. Accessibility permissions of Grader++ is different for each actor. Firstly, students can only see the tasks that are scheduled for the courses that they are taking. Students can only see the active tasks. Student can see their previous submissions. Other than these actions, students cannot do anything.

TA can only see the tasks that are assigned to the courses that they subscribed. TAs can see both active tasks and closed tasks. TA can list previous submissions. TAs can edit tasks.

Individual instructors can see the tasks created by any other instructor of the same course. Instructors are only allowed to see the courses that he generated. Instructor can add or delete tasks. Instructors can add or delete courses.

Actors/Class	SubmissionCtrl	AuthCtrl	TaskCtrl	CourseCtrl

student	showStudentSubmissions()
	submit()
		login()	showStudentTasks()	subscribe()
				unsubscribe()

TA	showSubmissions()	login()	showTasks()
			editTask()
				subscribe()
				unsubscribe()

Instructor	showSubmissions()	login()	showTasks()
			addTask()
			deleteTask()
				addCourse()
				deleteCourse
				()

4.5.3 Global Software Control

For every part of Grader++ using event driven control systems would be reasonable. Because of the nature of Grader++ that always waits for user actions, and tries to respond accordingly. Even though it is generally event driven, different parts use various patterns and systems.

For client part (UI) of Grader++ we plan to use Model-View-Controller (MVC) architecture. Although MVC is not the fastest, it is the appropriate choice for developing a maintainable and scalable event driven system. Views waits for action. If action occurs, view assigns it to controller. Controller goes to model if it is needed. To get data client calls database. To submit a code, it calls methods from core. Core evaluates codes and responds when it is ready.

Between core and UI there is facade pattern. Facade pattern mainly operates when student submits a code. Interface in this pattern will provide necessary connections between UI and core.

4.5.4 Boundary Conditions

Initialization: Simply, client side of Grader++ is a web-based classical server-client system. An Internet connection is enough to interact with system. Grader++ uses Java to evaluate codes and to form user-interface. To use Grader++ actually, user needs to have an account. Student and TA can create an account through website. For instructors, they should contact system administrator to create an account. When user logs in to Grader++, home page will be shaped specifically according to the user type. For all of the users, fetching some data from database is necessary. The reason is all the data will be stored in servers. Data is not going to be big because there is no any image or media data, but will be text. Hence, start-up of system should also be fast.

Termination: Single subsystems are not allowed to terminate individually, since they are all needed for the system's continuity. If a subsystem terminates because of a fatal error, the whole system needs to be rebooted. Since system is light-weighted it is not a big deal. We are going to be using MongoDB behind the scene, in order to implement the database.

Failure: All the data we need to keep are tasks, submissions, courses and user information. System will be designed to keep these data in the database all the time and fetch the data whenever it is needed. Therefore, during fetching data if any communication link or a router fails nothing will be lost. System needs to be refreshed. If there is a problem during writing data to database, there might be problems. MongoDB is a powerful database system. We think that most of the time MongoDB will handle these problematic situations in the background.

5. Object Design

5.1 Design Patterns

In this section we provide the description for adaptation of the patterns we used, as well as the modified version of the class diagram with these patterns applied. For the specific parts of client and core subsystems we used strategy pattern, singleton pattern, and facade pattern with different motives explained below.

5.1.1 Singleton Pattern

Singleton pattern is about instantiating a single object of a class and using it during the life-period of the application. At client side, we use this pattern with LoginController, DatabaseConnectionService, ServerConnectionService and WorkerQueue classes. They are all singleton classes, instantiated once and injected into required classes at the start of the application life-cycle.

- When User logs into the system, we instantiate a LoginController object and inject it on each page controller class since we need to check whether authorization process is done or not before showing the page to a user.
- DatabaseConnectionService is used for making all the required DB tasks, and ServerConnectionService is used to communicate with remote server. We inject their singleton objects to required controller classes and use them.

- In core system, we have a singleton class named WorkerQueue. This class is instantiated once at the start-up of the server, and used all along the life-period of server. This class is responsible for orchestrating Worker objects. It will allow us to assign our resources to the tasks in a well defined behaviour and prevent overloading and system crashes.

5.1.2 Strategy Pattern

In the web client subsystem of graderpp we provide different pages with specialized content according to the user type of end user who is logged in to the client. Web pages provided for browser consumption are created based on different java classes. For example homepage differs in content for instructors and students. In order to control this process each page object (such as homepage, course manager, task manager) extends the abstract class PageController, and uses it's functions to determine how to behave.

5.1.3 Façade Pattern

Façade Pattern is about enclosing the complex processes via interfaces and simplifying the procedure of complicated and big operations. We use Façade classes to access to the remote server and database. Database has many operations such as getting the tasks of a specific course or finding a submission with given id, and our System is to connect to our RemoteServer and send the code files to it. However, those operations is being done at the deep of our system with huge and complicated code, but we want to make these frequent tasks easier in our page controller classes. Therefore, we designed Façade classes to be an intermediate between our external systems and client side application.

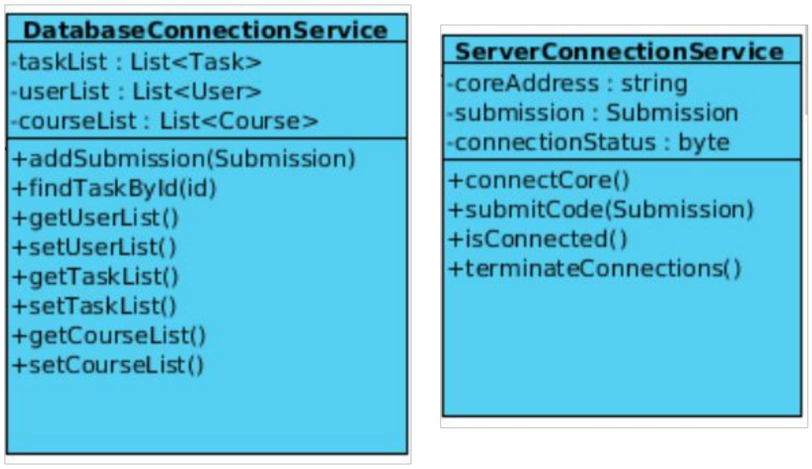


Figure 4. Façade classes for external system connection

In client side, We use DatabaseConnectionService and ServerConnectionService to make those operations much more easily with a couple lines of code. DatabaseConnectionService provides us methods like findTaskById(), setCourseList() to retrieve and manipulate data easily in page controller classes. ServerConnectionService make the file submission process simpler as well via HTTP requests.

In our core system, another Façade class named as CoreFacade exists. It provides an interface to make complex operations just by calling a single function like evaluateSubmission() or addTask(). Also core part of the subsystem composition, uses three layered architecture and CoreFacade is a part of CoreClient subsystem.

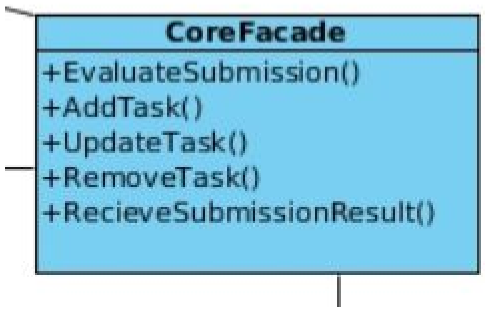
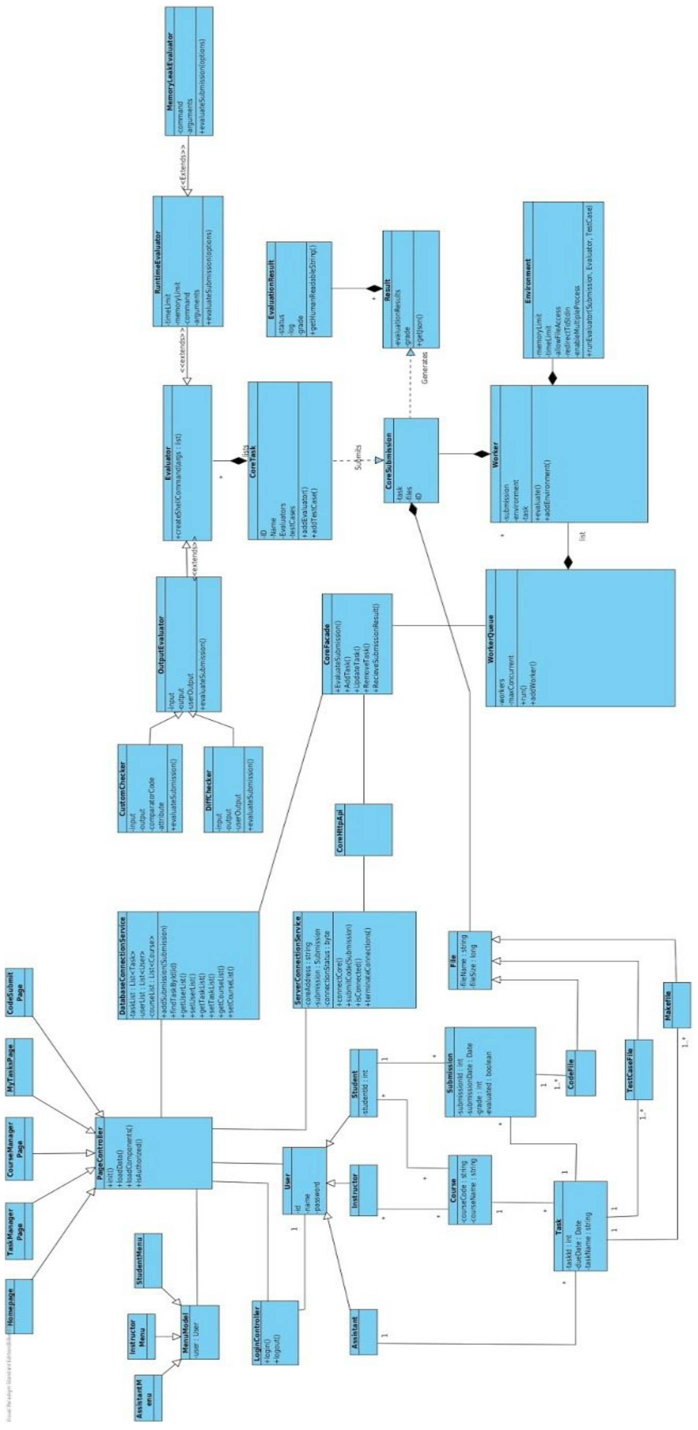


Figure 5. Façade class for Core system operations

5.1.3 Pattern Applied Class Diagram



5.2 Class Interfaces

In this section we provide descriptions for our classes in terms of their interfaces, methods, visibility. For brevity we have divided this part into implementation packages.

Models package

User

This is the model for each account in the system. This class is inherited by Student, Assistant and Instructor classes. It has properties such as username, password, fullName and userId (all of them Strings) and corresponding getter and setter methods.

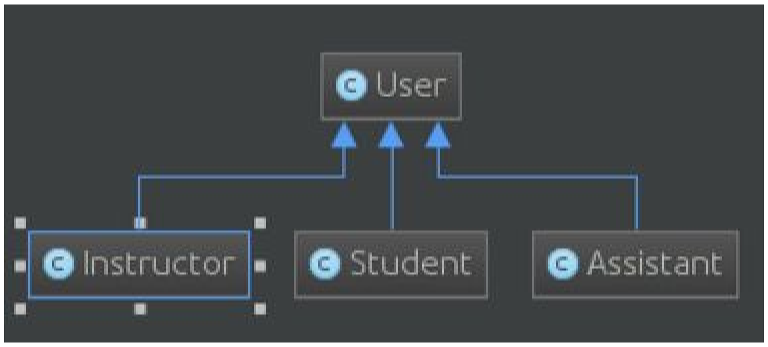


Figure: The inheritance of User classes

- **Assistant**

  This class inherits methods and properties from User. Additionally, it has a list of tasks for storing the Tasks which the Assistant is responsible for.

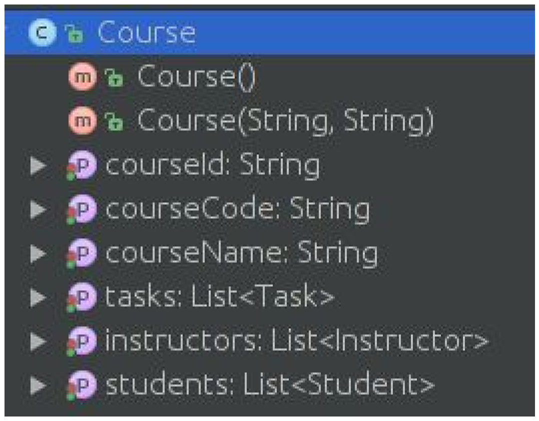
- **Student**

  This class inherits methods and properties from User. Additionally, it has a list of tasks for storing the Tasks assigned and a list of Submissions which holds the submissions made to those tasks by this Student.

- **Instructor**

  This class inherits methods and properties from User. Additionally, it has a list of courses for storing the Courses which the Instructor is responsible for.

Course



```
Course
    Course()
    Course(String, String)
  ▶ courseId: String
  ▶ courseCode: String
  ▶ courseName: String
  ▶ tasks: List<Task>
  ▶ instructors: List<Instructor>
  ▶ students: List<Student>
```

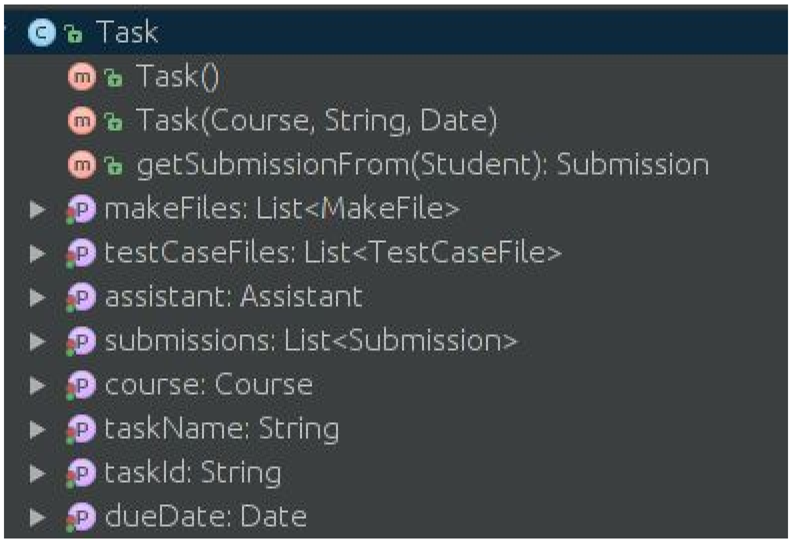
Figure: The properties and methods of Course class

This class represents each Course in the system. It has the properties such as courseId, courseCode, courseName (all of them are Strings). And also:

- tasks: The list of Tasks which belongs to this Course.
- instructors: The Instructors of this Course
- students: The Students registered to this Course.

There are corresponding getter and setter methods.

Task



```
Task
    Task()
    Task(Course, String, Date)
    getSubmissionFrom(Student): Submission
  ▶ makeFiles: List<MakeFile>
  ▶ testCaseFiles: List<TestCaseFile>
  ▶ assistant: Assistant
  ▶ submissions: List<Submission>
  ▶ course: Course
  ▶ taskName: String
  ▶ taskId: String
  ▶ dueDate: Date
```

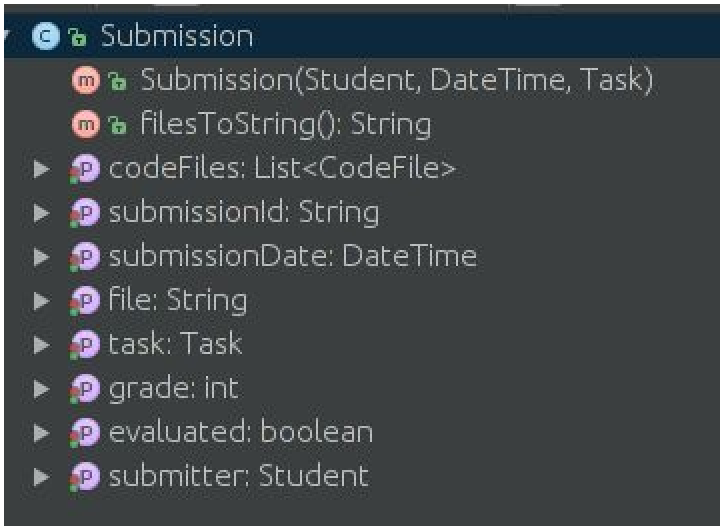
Figure: The properties and methods of Task class

This class represents each task in the system. It has the properties such as taskId (String),taskName (String), dueDate (Date), assistant (Assistant), course (Course). And also:

- submissions: The list of Submissions which belongs to this Task.
- makeFiles: The list of MakeFiles of this Task.
- testCaseFiles: The TestCaseFiles of this Task.

There are corresponding getter and setter methods. And a method named as getSubmissionFrom(Student student) for picking up the submission of a specific student from the list of submissions.

Submission

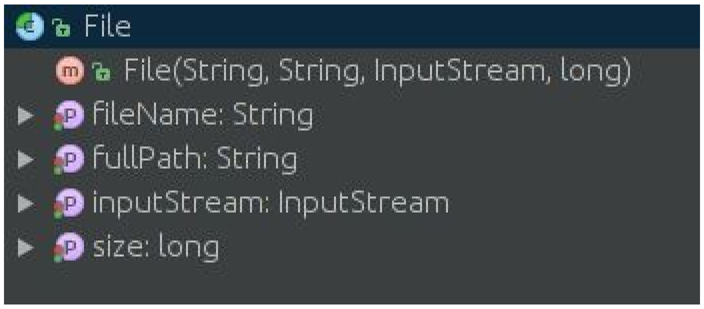


```
Submission
    Submission(Student, DateTime, Task)
    FilesToString(): String
  codeFiles: List<CodeFile>
  submissionId: String
  submissionDate: DateTime
  file: String
  task: Task
  grade: int
  evaluated: boolean
  submitter: Student
```

Figure: The properties and methods of Submission class

This class represents each submission made in the system. It has the properties such as submissionId, submissionDate, grade, evaluated, task, submitter. And also codeFiles as the list of CodeFiles which belongs to this Submission.
There are corresponding getter and setter methods.

File



```
File
    File(String, String, InputStream, long)
▶ fileName: String
▶ fullPath: String
▶ inputStream: InputStream
▶ size: long
```

Figure: The properties of File class

This classes is for representing each File possessed by Submissions or Tasks. It holds the fileName, fullPath, size and inputStream object as the byte-wise data itself.

- **CodeFile:**

  This kind of files are owned by Submissions. Each file uploaded for a submission belongs to this class.

- **MakeFile**

  This class is for holding the specific makefiles for Tasks. They are defined by the Instructor who assigned the Task.

- **TestCaseFile**

  This class is for holding the test case files for evaluation process. Objects of this class are generated in the period of the definiton of the Task by the Instructor.

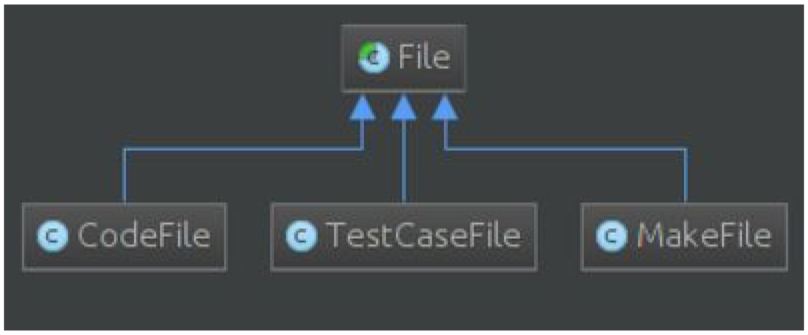


Figure: Inheritance of File classes

Service package

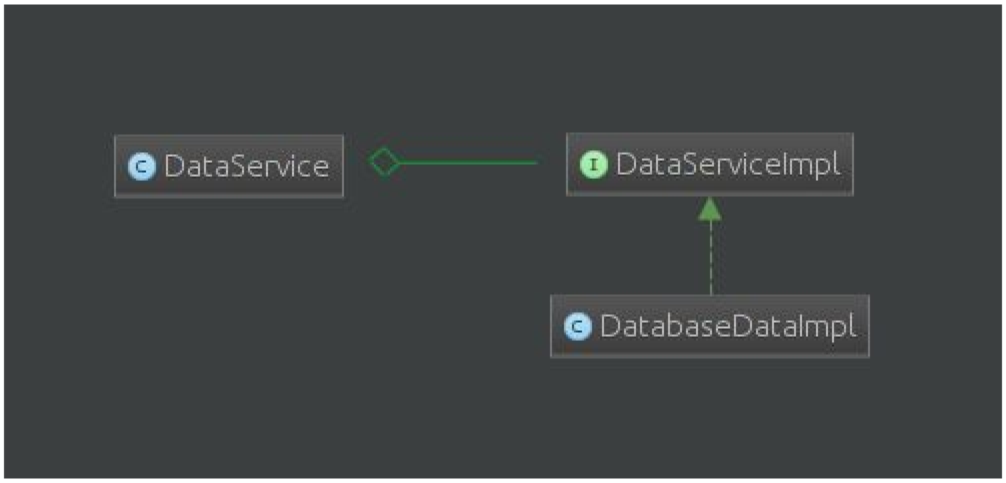


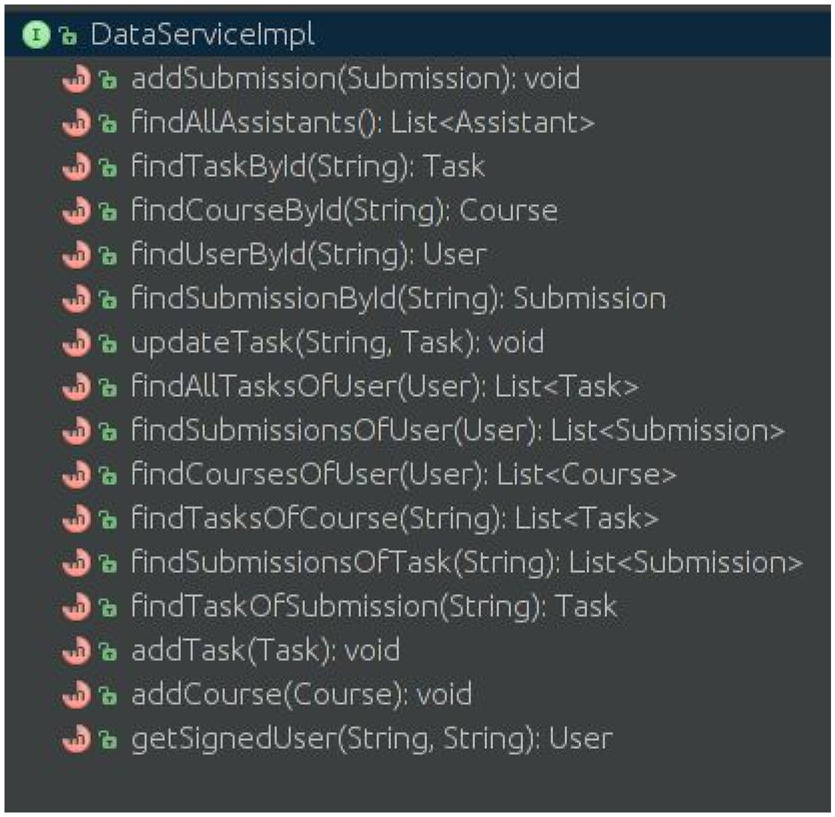
Figure: Bridge Pattern of Service package

DataService

It is actually a bridge class to access to the real data service implementation. It has a single property named as realDataService (which is a DataServiceImpl). It's constructor initializes this property. It has also getter and setter method for this property.

DataServiceImpl

This is an implementation interface for data operations. It holds the signatures of the methods for retrieving, deleting, adding and updating data. No matter which implementation is used (database, XML etc.), this interface should be implemented.


```
DataServiceImpl
    addSubmission(Submission): void
    findAllAssistants(): List<Assistant>
    findTaskById(String): Task
    findCourseById(String): Course
    findUserById(String): User
    findSubmissionById(String): Submission
    updateTask(String, Task): void
    findAllTasksOfUser(User): List<Task>
    findSubmissionsOfUser(User): List<Submission>
    findCoursesOfUser(User): List<Course>
    findTasksOfCourse(String): List<Task>
    findSubmissionsOfTask(String): List<Submission>
    findTaskOfSubmission(String): Task
    addTask(Task): void
    addCourse(Course): void
    getSignedUser(String, String): User
```
Figure: Methods of DataServiceImpl interface

Methods:

- addSubmission(Submission submission): It is used for saving a submission entry to the database.
- addTask(Task task): It is used for saving a Task entry to the database.
- addCourse(Course course): It is used for saving a Course entry to the database.
- updateTask(String taskId, Task task): It is used to update the Task in the database with a new Task object.
- updateCourse(String courseId, Course course): It is used to update the Course in the database with a new Course object.
- getSignedUser(String username, String password): This method is called in the procedure of sign-in. If username and password matches with an entry in database, corresponding User object is returned.
- findAllAssistants(): All Assistants are returned as a list from the database with this method.
- findTaskById(String taskId): Given an ID number, corresponding Task object is returned.
- findCourseById(String courseId): Given an ID number, corresponding Course object is returned.
- findUserById(String userId): Given an ID number, corresponding User object is returned.

- findSubmissionById(String submissionId): Given an ID number, corresponding Submission object is returned.
- findAllTasksOfUser(User user): Given a User object, all Tasks of it are returned as a list.
- findCoursesOfUser(User user): Given a User object, all Courses of it are returned as a list.
- findSubmissionsOfUser(User user): Given a User object, all Submissions of it are returned as a list.
- findTasksOfCourse(String courseId): Given a Course object, all Tasks of it are returned as a list.
- findSubmissionsOfTask(String taskId): Given a Task object, all Submissions of it are returned as a list.
- findTaskOfSubmission(String submissionId): Given a Submission object, associated Task of it returned as an object.

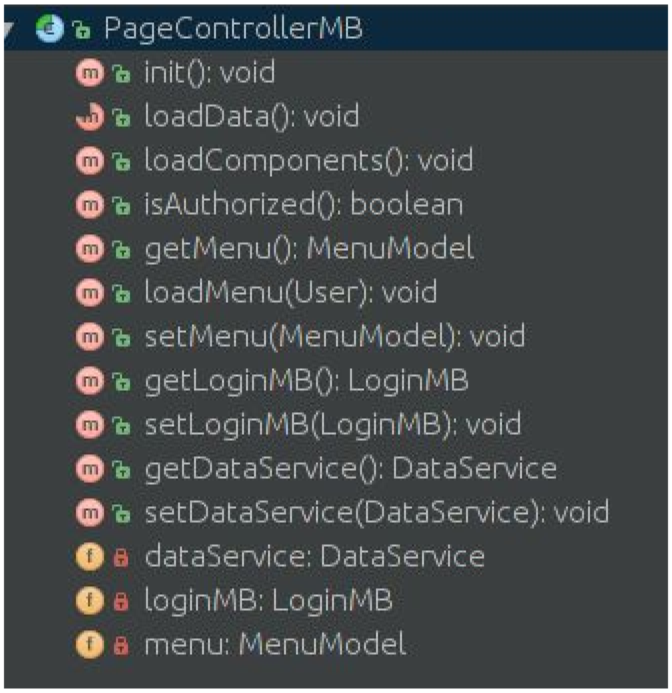
DatabaseDataImpl

This is an actual implementation class for data operations. It is written for MongoDB databases. This class implements all the methods defined in the DataServiceImpl interface. The necessary explanation about those methods is given on the interface section.

ManagedBeans package

PageControllerMB

This is an abstract class for the Controller classes of the user interface. All the other controller classes are childs of this abstract class. It has several abstract methods to handle the common controller logic. The method init() is called on the construction of the view, and within it calls isAuthorized() for handling the access control, loadData() and loadComponents() for the initializing view components and necessary data to the page. As properties, it stores DataService and LoginMB singleton objects for database and authorization operations, and a MenuModel object specific to the user signed in.

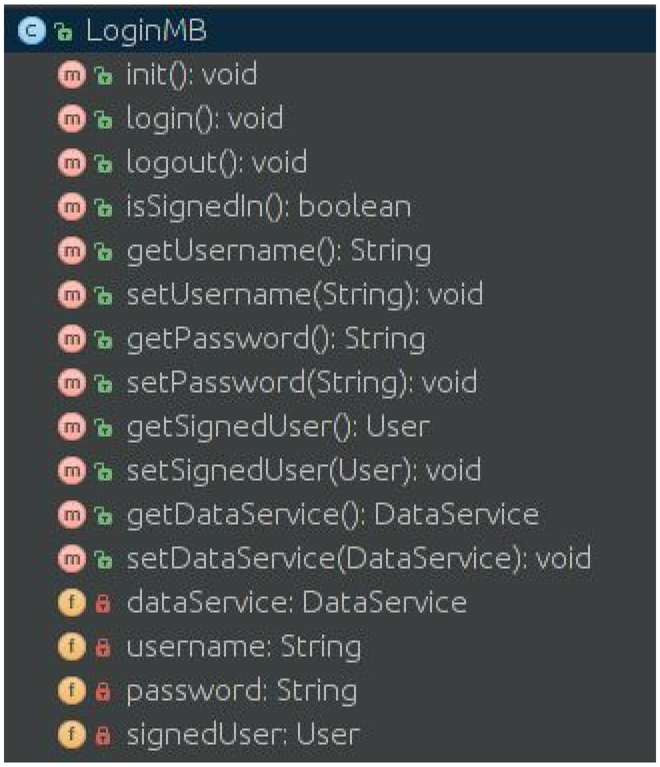


```
PageControllerMB
    init(): void
    loadData(): void
    loadComponents(): void
    isAuthorized(): boolean
    getMenu(): MenuModel
    loadMenu(User): void
    setMenu(MenuModel): void
    getLoginMB(): LoginMB
    setLoginMB(LoginMB): void
    getDataService(): DataService
    setDataService(DataService): void
    dataService: DataService
    loginMB: LoginMB
    menu: MenuModel
```

Figure: The properties and methods of PageControllerMB class

LoginMB

This controller class handles all the sign-in and sign-out processes, also the singleton instance of this class is used by all the other controller classes for dealing with authorization. It inherits methods and properties from PageControllerMB, and has methods such as login() and logout(). Aside from those, it has getter and setter methods for the properties: username (String), password (String), signedUser (User). This class stores the currently signed User, and it is used by many other classes.

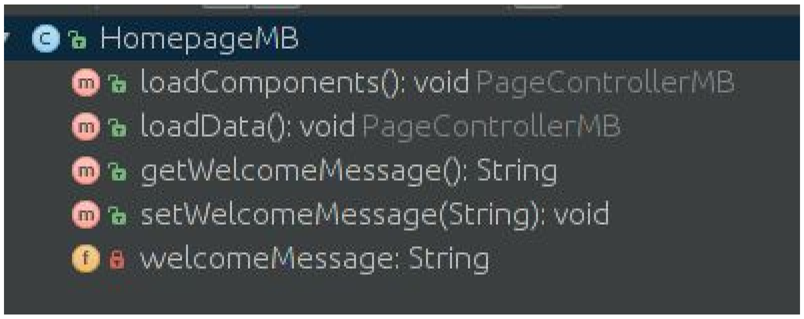


```
LoginMB
    init(): void
    login(): void
    logout(): void
    isSignedIn(): boolean
    getUsername(): String
    setUsername(String): void
    getPassword(): String
    setPassword(String): void
    getSignedUser(): User
    setSignedUser(User): void
    getDataService(): DataService
    setDataService(DataService): void
    dataService: DataService
    username: String
    password: String
    signedUser: User
```

Figure: The properties and methods of LoginMB class

HomepageMB

This is the controller class for the homepage. It holds a welcome message and inherits properties and methods from PageControllerMB.

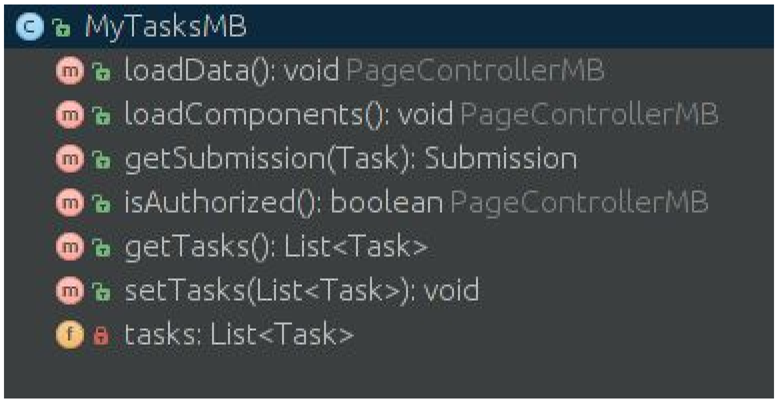


```
HomepageMB
    loadComponents(): void PageControllerMB
    loadData(): void PageControllerMB
    getWelcomeMessage(): String
    setWelcomeMessage(String): void
    welcomeMessage: String
```

Figure: The properties and methods of HomepageMB class

MyTasksMB

This is the controller for the “My Tasks” page. This page can only be viewed by Students. It has a getSubmission(Task) method for retrieving the submission made to a specific Task by the Student. As a property, this class stores a list of Tasks which includes only the tasks assigned to that Student.

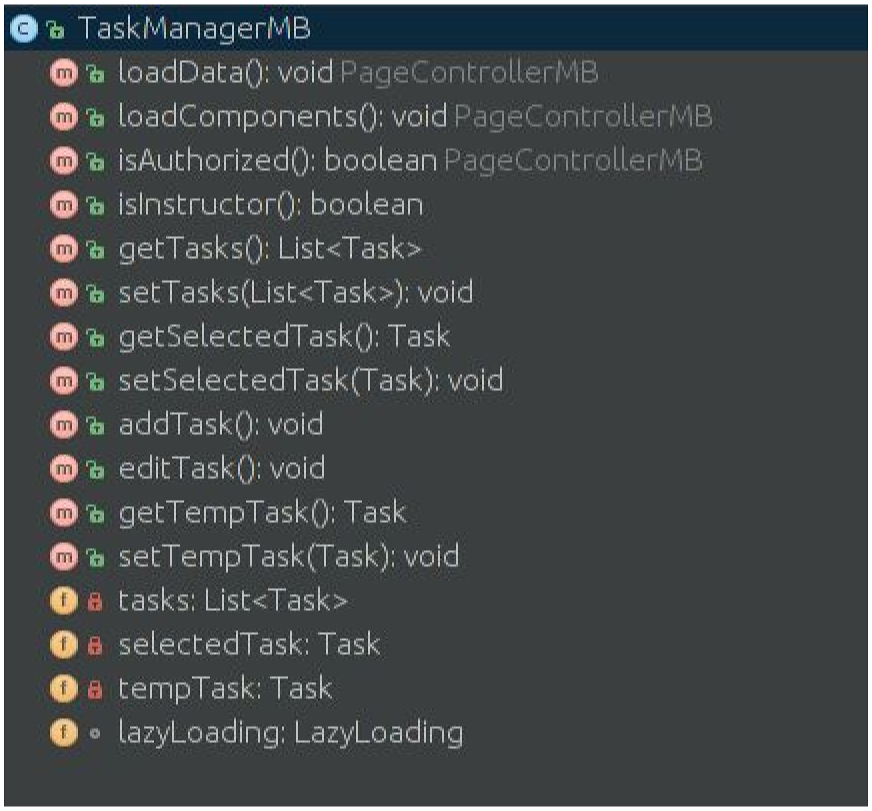


```
MyTasksMB
    loadData(): void PageControllerMB
    loadComponents(): void PageControllerMB
    getSubmission(Task): Submission
    isAuthorized(): boolean PageControllerMB
    getTasks(): List<Task>
    setTasks(List<Task>): void
    tasks: List<Task>
```

Figure: The properties and methods of MyTasksMB class

TaskManagerMB

This is controller class for the page named as “Task Manager”. This page can be viewed by Instructors and Assistants. This controller class is responsible for all the manipulations done on Tasks.



```
TaskManagerMB
    loadData(): void PageControllerMB
    loadComponents(): void PageControllerMB
    isAuthorized(): boolean PageControllerMB
    isInstructor(): boolean
    getTasks(): List<Task>
    setTasks(List<Task>): void
    getSelectedTask(): Task
    setSelectedTask(Task): void
    addTask(): void
    editTask(): void
    getTempTask(): Task
    setTempTask(Task): void
    tasks: List<Task>
    selectedTask: Task
    tempTask: Task
    lazyLoading: LazyLoading
```

Figure: The properties and methods of TaskManagerMB class

Properties are:

- tasks: It is a List<Task> object. This holds all the Task objects which can User (Instructor or Assistant) is allowed to view and manipulate.
- selectedTask: It is a Task object. It is initialized when a task is selected by the user to see its details and submissions or to edit it.
- tempTask: It is a temporary Task object instantiated when User wants to add a new Task to the system. This temporary object is saved to database on the addTask() method.
- lazyLoading: It is a reference to the singleton object of LazyLoading. It is used in lazy loading process of objects. The detailed information about it can be seen on the description of LazyLoading class.

Methods are:

- addTask(): This method is called once the user fills the input fields of "Add Task" dialog and clicks to the save button. The temporary Task object is instantiated and saved to database via this method. This method accesses to DataService for DB operations.
- editTask(): This method is called once the user completes the necessary changes to the Task object. The temporary Task object is replaced with the old object and saved to DB via DataService.
- isInstructor(): It just returns a boolean value which indicates whether the currently signed user is an Instructor or not with the help of LoginMB object.
- isAssistant: It just returns a boolean value which indicates whether the currently signed user is an Assistant or not with the help of LoginMB object.
- loadData() and loadComponents(): This method installs all the required data and components at the start of the page initialization.

CourseManagerMB

This is controller class for the page named as "Course Manager". This page can only be viewed by Instructors. This controller class is responsible for all the manipulations done on Courses.

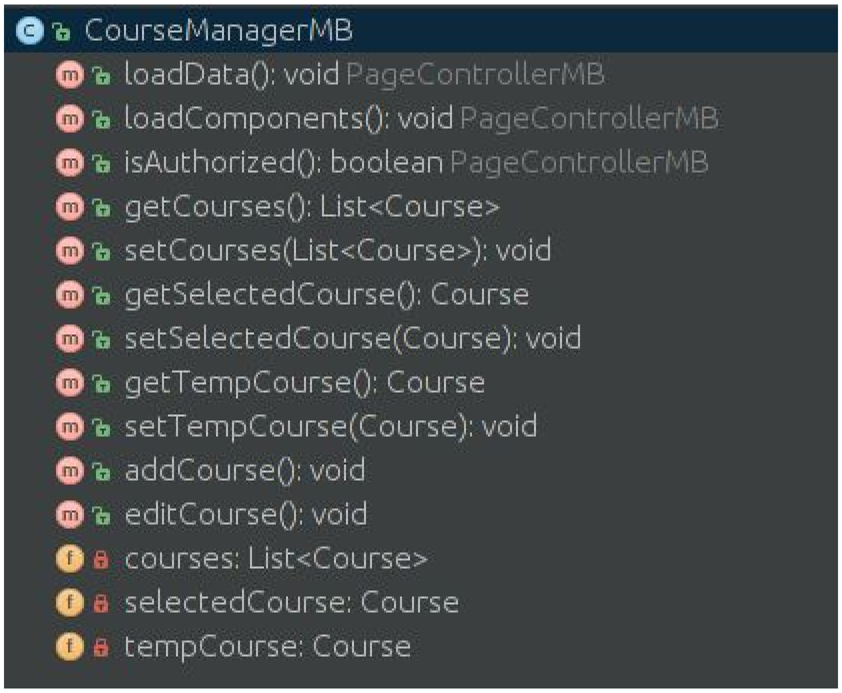

```
CourseManagerMB
  loadData(): void PageControllerMB
  loadComponents(): void PageControllerMB
  isAuthorized(): boolean PageControllerMB
  getCourses(): List<Course>
  setCourses(List<Course>): void
  getSelectedCourse(): Course
  setSelectedCourse(Course): void
  getTempCourse(): Course
  setTempCourse(Course): void
  addCourse(): void
  editCourse(): void
  courses: List<Course>
  selectedCourse: Course
  tempCourse: Course
```

Figure: The properties and methods of CourseManagerMB class

Properties are:

- courses: It is a List<Course> object. This holds all the Course objects which can Instructor is allowed to view and manipulate.
- selectedCourse: It is a Course object. It is initialized when a course is selected by the instructor to see its details and task or to edit it.
- tempCourse: It is a temporary Course object instantiated when user wants to add a new course to the system. This temporary object is saved to database on the addCourse() method.
- lazyLoading: It is a reference to the singleton object of LazyLoading. It is used in lazy loading process of objects. The detailed information about it can be seen on the description of LazyLoading class.

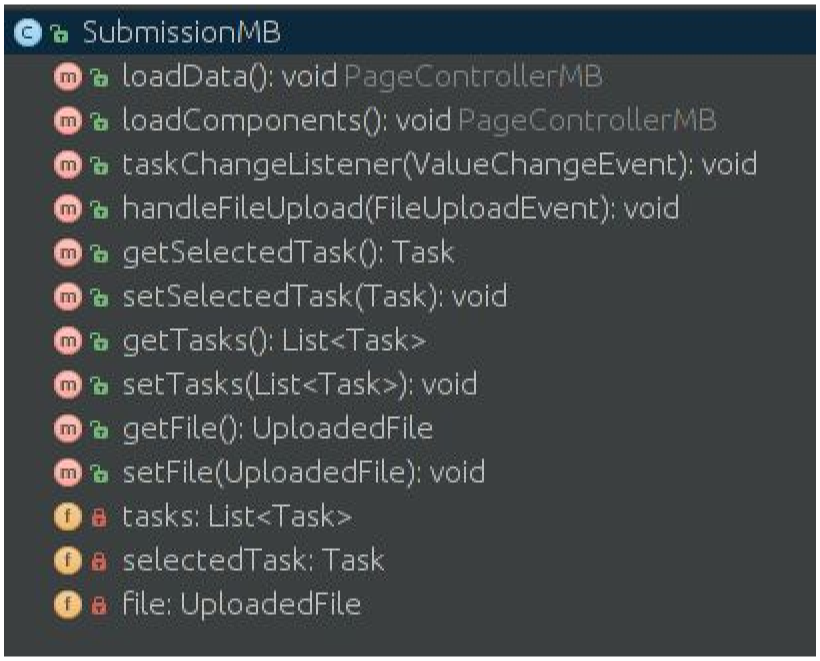
Methods are:

- addCourse(): This method is called once the user fills the input fields of "Add Course" dialog and clicks to the save button. The temporary Course object is instantiated and saved to database via this method. This method accesses to DataService for DB operations.
- editCourse(): This method is called once the user completes the necessary changes to the Course object. The temporary Course object is replaced with the old object and saved to DB via DataService.

- isAuthorized(): It just returns a boolean value which indicates whether the currently signed user is an Instructor or not with the help of LoginMB object.
- loadData() and loadComponents(): This method installs all the required data and components at the start of the page initialization.

SubmissionMB

This is the controller class which is associated with Quick Submission page which is special to the Students.



```
SubmissionMB
    loadData(): void PageControllerMB
    loadComponents(): void PageControllerMB
    taskChangeListener(ValueChangeEvent): void
    handleFileUpload(FileUploadEvent): void
    getSelectedTask(): Task
    setSelectedTask(Task): void
    getTasks(): List<Task>
    setTasks(List<Task>): void
    getFile(): UploadedFile
    setFile(UploadedFile): void
    tasks: List<Task>
    selectedTask: Task
    file: UploadedFile
```

Figure: The properties and methods of SubmissionMB class

Properties are:

- tasks: It holds the tasks which the signed Student is assigned to and has not submitted to yet.
- selectedTask: It is instantiated when the student selects a Task in the taskChangeListener(..) method
- file: This is an UploadedFile object and instantiated once the file uploading process is started. handleFileUpload(...) sends this object to our remote server.

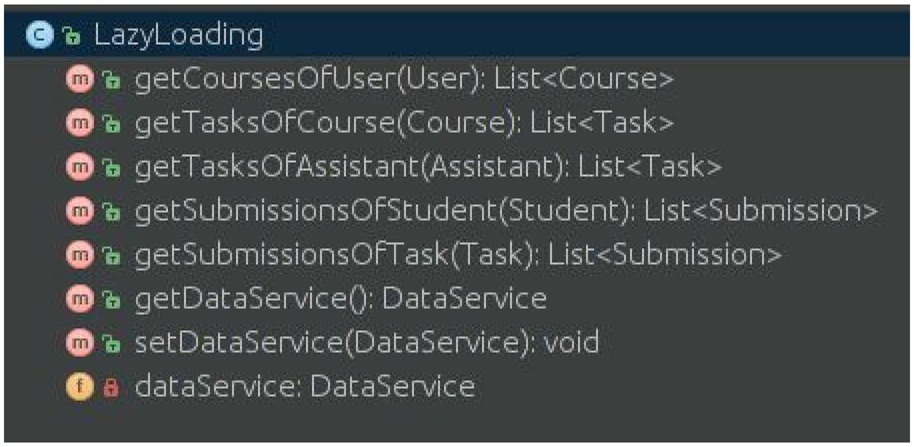
Methods are:

- taskChangeListener(ValueChangeEvent event): This method is invoked when the user selects a task from menu. It assigns the Task selected, to selectedTask object.
- handleFileUpload(FileUploadEvent event): This method deals with the file uploading process. It takes the selectedFile and sends to core system via an HTTP request.
- loadData() and loadComponents(): This method installs all the required data and components at the start of the page initialization.

Misc package

LazyLoading

This is a *Singleton* class and used by many controller classes. It deals with the *"Lazy Loading"* process which is a design pattern. As it can be seen from the class diagram, our classes have many references among themselves and it gets complicated over the time. Particularly, the first initialization of objects is a very heavy task for the system to perform. For instance, installation of a single Course object requires initalization of related Task objects, Submission objects, and CodeFile objects. We overcome this heavy-loading issue via the *Lazy Loading* pattern, we don't set the related objects in the first initialization of object. We fill the references only when they are requested during the process. That is, for example, list of submissions are set once getSubmissions() are called. This process is managed by LazyLoading class.



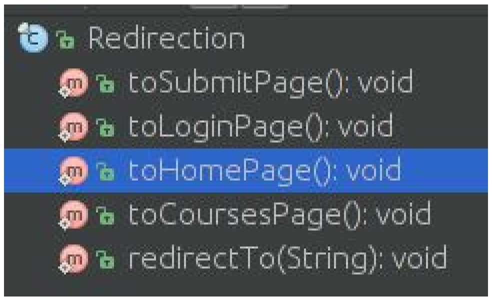
```
LazyLoading
    getCoursesOfUser(User): List<Course>
    getTasksOfCourse(Course): List<Task>
    getTasksOfAssistant(Assistant): List<Task>
    getSubmissionsOfStudent(Student): List<Submission>
    getSubmissionsOfTask(Task): List<Submission>
    getDataService(): DataService
    setDataService(DataService): void
    dataService: DataService
```

Figure: The properties and methods of LazyLoading class

This class has one property which is DataService. It uses this service for completing all the methods below. Methods of this class are as follows:

- getCoursesOfUser(User user): This method returns the list of Courses associated with the given user.
- getTasksOfCourse(Course course): This method returns the list of Tasks associated with the given course.
- getTasksofAssistant(Assistant assistant): This method returns the list of Tasks associated with the given assistant.
- getSubmissionsOfStudent(Student student):This method returns the list of Submissions associated with the given student.
- getSubmissionsOfTask(Task task):This method returns the list of Submissions associated with the given task.

Redirection



```
Redirection
    toSubmitPage(): void
    toLoginPage(): void
    toHomePage(): void
    toCoursesPage(): void
    redirectTo(String): void
```

Figure: The methods of Redirection class

This is another Singleton class which is used for handling the redirection process between pages. When the user clicks to a link of the page from the menu, it invokes the corresponding method among the methods toSubmitPage(), toLoginPage(), toHomePage(), toCoursesPage(), redirectTo(String pageName).

Components Package

This package holds some customized components which belongs to the views.

AssistantMenu

This is a child class of DefaultMenuModel class of PrimeFaces framework. It consists of customized buttons and navigator links for an Assistant user. It has only a constructor which takes an object of Assistant as a parameter

InstructorMenu

This is a child class of DefaultMenuModel class of PrimeFaces framework. It consists of customized buttons and navigator links for an Instructor user. It has only a constructor which takes an object of Instructor as a parameter

StudentMenu

This is a child class of DefaultMenuModel class of PrimeFaces framework. It consists of customized buttons and navigator links for an Assistant user. It has only a constructor which takes an object of Student as a parameter

Converter Package

This package holds some classes named as “Converter”s. This classes’ purpose is to establish the transformation betweeen an ID key and a model object. For example, when a Task is selected from the menu via the web-page, the ID of it is received and the necessary Task object is instantiated through this converter classes. It has only two methods:

- public Object getAsObject(FacesContext context, UIComponent component, String value): It is for retrieving object from the database with an id.
- public String getAsString(FacesContext context, UIComponent component, Object value) : It is for retrieving the ID of the specific object’s from the database.

All the converter classes listed has the same two methods, each of them is to convert a specific entity: AssistantConverter, UserConverter, SubmissionConverter, TaskConverter, CourseConverter.

5.3 Specifying Contracts

A-) graderppWeb.models.User

1. context User **inv:**

self.username not null

2. context User **inv:**

self.password not null

3. context User **inv:**

self.fullName not null

B-) graderppWeb.models.Course

4. context Course **inv:**

self.courseId not null

5. context Course **inv:**

self.courseCode not null

6. context Course **inv:**

self.courseName not null

C-) graderppWeb.models.Submission

7. context Submission **inv:**

self.submissionId not null

8. context Submission **inv:**

self.submitter not null

9. context Submission **inv:**

self.submissionDate not null

10. context Submission **inv:**

self.task not null

D-) graderppWeb.converter.AssistantConverter

11. context Converter::getAsString(c, comp, value) **pre:**

value not null

E-) graderppWeb.service.DataBaseDataImp

12. context DataBaseDataImp::addUser(name, pass, f_name, type) **pre:**

getSignedUser(name, pass) = null

13. context DataBaseDataImp::deleteUser(id) **pre:**

getUserDoc(id) not null

14. context DataBaseDataImp::deleteUser(id) **post:**

getUserDoc(id) = null

15. context DataBaseDataImp **inv:**

self._user_list not null

16. context DataBaseDataImp **inv:**

self._course_list not null

17. context DataBaseDataImp::subscribe2Course(user_id, course_id) **post:**

self._user_list->course_list->includes(course_id)

18. context DataBaseDataImp::subscribe2Task(user_id, task_id) **post:**

self._user_list->task_list->includes(task_id)

19. context DataBaseDataImp::subscribe2Submission(user_id, submission_id) **post:**

self._user_list->_submission_list->includes(submission_id)

E-) graderppWeb.misc.LazyLoading

20. context LazyLoading::getTaskOfSubmission(submission) **pre:**

submission not null

21. context LazyLoading::getTaskOfCourse(course) **pre:**

course not null

22. context LazyLoading::getTaskOfUser(user) **pre:**

user not null

23. context LazyLoading::getSubmissionsOfStudent(student) **pre:**

student not null

24. context LazyLoading::getSubmissionsOfTask(task) **pre:**

task not null

H-) graderppCore.utils.Util

25. context Util::saveFile(uploadInpStream, serverLoc) **pre:**

uploadInpStream not null

H-) graderppCore.api.Task

26. context Task::uploadInput(inpStream, dispHandler, taskID, inpID) **pre:**

inpID not null

27. context Task::uploadCompileFiles(inpStream, dispHandler, taskID) **pre:**

graderppWeb.service.DataBaseDataImp::findTaskById(id) not null

28. context Task::uploadCompileFiles(inpStream, dispHandler, taskID) **pre:**

inpStream not null

29. context Task::saveFile(inpStream, serverLocation) **pre:**

inpStream not null

30. context Task::saveFile(inpStream, serverLocation) **pre:**

serverLocation not null

6. Conclusion

In this report, we have focused on what the system should do, what functions it should include, and with the design part we thought of the general implementation and design decisions. Expected features of the system and given requirements were our main helper to form use case models and their scenarios in analysis part of report. And finally, we decided the implementation specifics and worked on object design that included actual implementation details.

Writing the analysis report was a useful process because it helped us to think about the design process thoroughly even before we started the design and implementation phase. Because of this, we were able to easily make designing decisions. In the analysis part of the report, our main target was to make people understand the analysis of our system and explain our user-friendly system for students and instructors. To achieve this goal, we used design tools such as Visual Paradigm, and used Unified Modeling Language (UML). Since UML is a unique programming language, it really helped us to communicate within our group and with other people easily.

Designing part of the report was to explain the design process and, help in deciding our general implementation path. We used Visual Paradigm to create our UML diagrams such as deployment diagram, component diagram, use case diagrams, class diagram, sequence diagrams, state and activity diagrams etc. Aside from the diagrams we also discussed and eventually decided how user interface should be formed by taking design goals into account such as user friendliness. We decided how system should respond to the user actions. We generally tried to provide understandable and informative user interface. We want our users to reach their goal with just a few steps. Consequently, we detailed our core and UI designs to implement them in object-oriented fashion.

Finally, in the object design part of the report we decided and then provided the actual implementation details, and explained the design patterns applied within our systems. Additionally we provided the interfaces of our subsystems, specified how our classes should interact with each other, and specified contracts.

Throughout the production of our project we learned that working on analysis, design reports beforehand ensures that the implemented version of our project will meet with the requirements that are needed, and will function as stable as possible. We learned that this is because we acquired the appropriate perspective about how to address software development processes, how to define requirements and to define ways of implementing our solutions, and we learned how to cooperate with each other within a teamwork environment.