GradeBook05

Requirements Specification

March 27, 2014

Team 5 - Members
Nick Alekhine - alekhn@ccs.neu.edu
Chris Clark - cclark07@ccs.neu.edu
Austin Colcord - acolcord@ccs.neu.edu
Charles Perrone - chperr@ccs.neu.edu

Table of Contents

I. Introduction

II. Use Cases

III. Non-Functional Requirements

IV. Functional Requirements

V. Development & Target Platforms

VI. UML Diagram

VII. Module Dependency Diagram

VIII. Team Member Contributions

I. Introduction

Old: We are building a digital grade book for teachers at a school system to easily manage their courses (which are called grade books) by assigning different assignments to their respective students. Only instructors and school administrators will have access to the system and update content. There may be future revisions where students will have read access to their grades.

New: We are building a digital grade book for an instructor to easily manage one of it's courses, where they can assign assignments to their students and get information from that data in an easy-to-understand manner. (*we made the changes to this new way because of the time constraint of the assignment, as well as the given MyGradeBook class requirements)*

II. Use Cases

Before we get into creating specific use case instances, let us define who is going to be using the system. This piece of software is meant to complement an instructor's pre-existing scheme for keeping track of assignments in the courses that they teach. For this project, the scope will only extend to the instructors and school administrators. Students will not have access to the system in this release.

Use Case 1 - OLD

Instructor wants to log into the system and get an overview of the courses they are teaching.

=> Instructor goes into the console
=> a prompt asks them to enter their name
=> the name gets checked with the existing list of instructors and is validated

=> if name is validated, then they will immediately get a display of all the courses they are teaching and a list of methods that they can enter into console (eg. avg course grade)
=> instructor can type in course name into console to get a list of assignments and students.

=> if name is not validated, they will be prompted to enter a name again.

Use Case 2 - OLD

Instructor wants to add an assignment to a course

Instructor goes into the console
=> a prompt asks them to enter their name
=> the name gets checked with the existing list of instructors and is validated
=> if name is validated, then they will immediately get a display of all the courses they are teaching and a list of methods that they can enter into console
=> instructor can type in course name into console to get a list of assignments and students
=> instructor can call a method 'addAssignment' to their list of courses, entering a name for the assignment
=> the system adds the assignment to the course/gradebook, which adds the assignment to all of the students in that course with a null grade

Use Case 3 - OLD

Instructor wants to change the grade of an assignment for a specific student

Instructor goes into the console
=> a prompt asks them to enter their name
=> the name gets checked with the existing list of instructors and is validated
=> if name is validated, then they will immediately get a display of all the courses they are teaching and a list of methods that they can enter into console
=> instructor can type in course name into console to get a list of assignments and students
=> instructor will type in the student's name into the console
=> system will return the list of assignments for the student
=> instructor will enter an assignment's name into the console
=> system will return the assignment's current grade (or '-' if none is added yet), and prompt for a new grade to be entered

Use Case 4 - OLD

Instructor wants to add a new course.

Instructor goes into the console
=> a prompt asks them to enter their name, the instructor will enter it
=> the name gets checked with the existing list of instructors and is validated
=> if name is validated, then they will immediately get a display of all the courses they are teaching and a list of methods that they can enter into console
=> instructor can call a method addCourse with a name for it
=> system will create a new gradebook in the instructor's list of gradebooks, with an empty list of students and an empty list of assignments
=> from here, the instructor can call methods addStudent or addAssignment continuously to add students and assignments to the course

Use Case 5 - OLD

System administrator adds a new student to school system

System administrator will go into the console
=> a prompt will ask them for their name, the administrator will enter “admin”
=> the system will display the name of the school, the list of instructors, and list of students
=> the system admin will call a method addStudent with the student’s name
=> the system will add the student to the school’s list of students

Use Case 6 - OLD
System administrator adds a new instructor to school system

System administrator will go into the console
=> a prompt will ask them for their name, the administrator will enter “admin”
=> the system will display the name of the school, the list of instructors, and list of students
=> the system admin will call a method addInstructor with the instructor’s name
=> the system will add the instructor to the school’s list of instructors

Use Case 7 - OLD
System administrator wants to add a student to one of their courses

System administrator will go into the console
=> a prompt will ask them for their name, the administrator will enter “admin”
=> the system will display the name of the school, the list of instructors, and list of students
=> the system admin will call a method addStudent with the student’s name
=> the system will add the student to the school’s list of students

Use Case 8 - NEW
Instructor wants to create a new gradebook

=> Instructor goes into Terminal and runs the Interfacer program
=> Instructor is then prompted with a set of commands to create a new gradebook.
=> Instructor then enters one of 3 commands to create a new gradebook.

COMMAND 1
=> An empty instance of gradebook is created

COMMAND 2
=> Instructor is prompted to enter the path of a text file that contains a gradebook

=> Text file is then parsed and a new gradebook is created.

COMMAND 3
=> Instructor is prompted to enter a text input of a gradebook
=> Text input is then parsed and a new gradebook is created

=> Program then displays a list of commands that can be used on newly created gradebook

Use Case 9 - NEW

Instructor wants to view the grades of all students in the gradebook
ASSUME USE CASE 8

=> UI displays a list of commands available to instructor
=> Instructor types in command to view grades of all students
=> UI displays all grades
=> Instructor returned to gradebook main menu.

Use Case 10 - NEW

Instructor wants to view the average for a specific assignment
ASSUME USE CASE 8

=> UI displays a list of commands available to instructor
=> Instructor types in command for viewing average grade for assignment
=> Instructor is then prompted to type in name of assignment
=> UI then displays the assignment average (if assignment exists)
=> Instructor returned to gradebook main menu.

Use Case 11 - NEW

Instructor wants to change the grade of a student’s assignment
ASSUME USE CASE 8

=> UI displays a list of commands available to instructor
=> Instructor types in command for changing an assignment grade
=> Instructor prompted to type in name of assignment
=> Instructor prompted to type in username of student
=> Instructor prompted to type in new grade
=> If assignment and username exist, grade is updated.
=> Instructor returned to gradebook main menu.

Use Case 12 - NEW

Instructor wants to add a gradebook file to existing gradebook
ASSUME USE CASE 8

=> UI displays a list of commands available to instructor
=> Instructor types in command for processing a file
=> Instructor prompted to insert path of file to be processed
=> If file is properly formatted, file is processed and added to existing gradebook.
=> Instructor returned to gradebook main menu

III. Non-Functional Requirements

The gradebook should have these qualities:

- Modular
  - The ability to add new data structures and make links between existing structures.
- Secure
  - Only instructors and system administrators should be able to access the system.
- Reliable
  - Functions that are run by both instructors and administrators must have known behaviors.
- Maintainable
  - The code should be well-organized and follow object-oriented design principles for future developers to easily understand the underlying structure of the system.
- Performance-oriented
  - Queries should be fast to execute.
- Test-Driven
  - The system should be built using test-driven development with every added method and class tested extensively.

IV. Functional Requirements

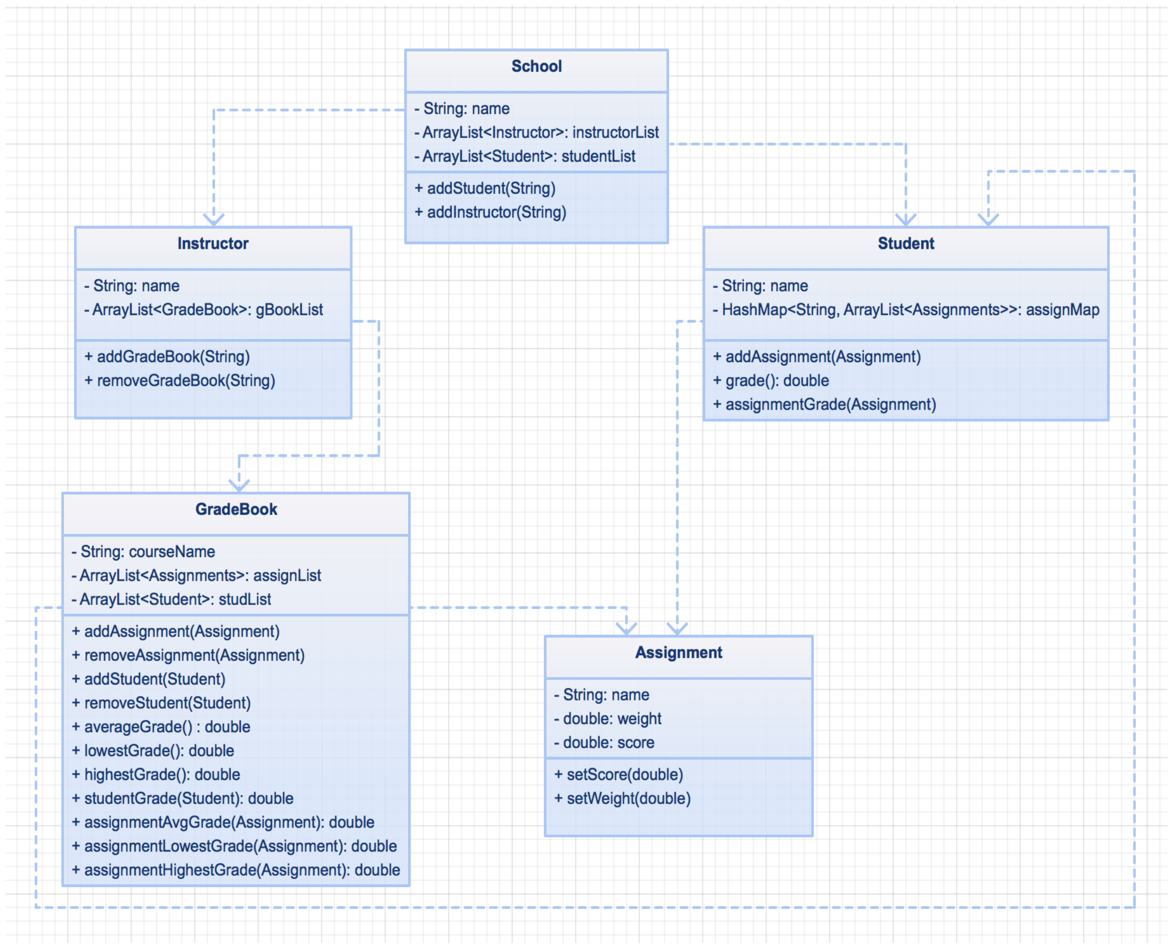
- The instructor shall be able to add new assignments
- The instructor shall be able to add new students
- The instructor shall be able to input a given gradebook and use it’s information
- The instructor shall be able to update (change) assignment info for courses and students

V. Development & Target Platforms

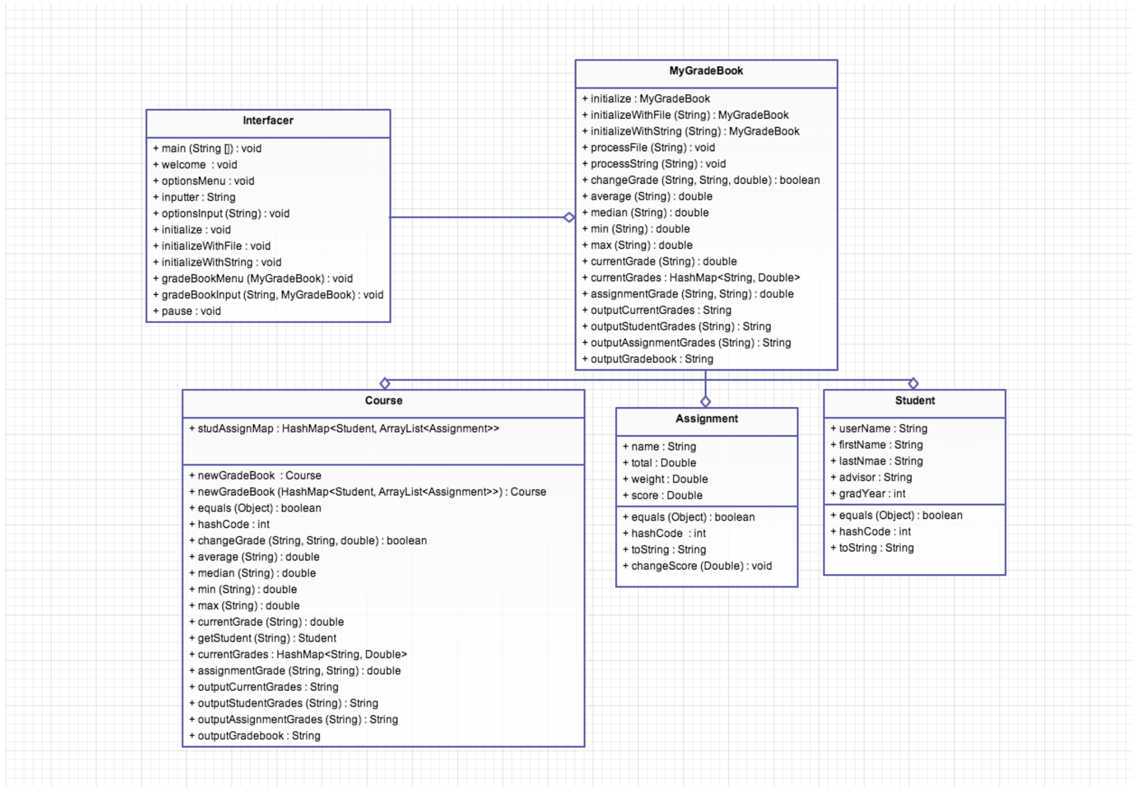
We are developing this system using Java 1.6 with the Eclipse IDE and JUnit 4 testing. The target platforms are any machines that can run JVM Runtime Environment and has access to a console. Essentially every major platform can run the JVM Environment (Windows, Mac OS, *nix).

VI. UML Diagram

Old


School
- String: name
- ArrayList<Instructor>: instructorList
- ArrayList<Student>: studentList
+ addStudent(String)
+ addInstructor(String)
Instructor
- String: name
- ArrayList<GradeBook>: gBookList
+ addGradeBook(String)
+ removeGradeBook(String)
Student
- String: name
- HashMap<String, ArrayList<Assignments>>: assignMap
+ addAssignment(Assignment)
+ grade(): double
+ assignmentGrade(Assignment)
GradeBook
- String: courseName
- ArrayList<Assignments>: assignList
- ArrayList<Student>: studList
+ addAssignment(Assignment)
+ removeAssignment(Assignment)
+ addStudent(Student)
+ removeStudent(Student)
+ averageGrade() : double
+ lowestGrade(): double
+ highestGrade(): double
+ studentGrade(Student): double
+ assignmentAvgGrade(Assignment): double
+ assignmentLowestGrade(Assignment): double
+ assignmentHighestGrade(Assignment): double
Assignment
- String: name
- double: weight
- double: score
+ setScore(double)
+ setWeight(double)

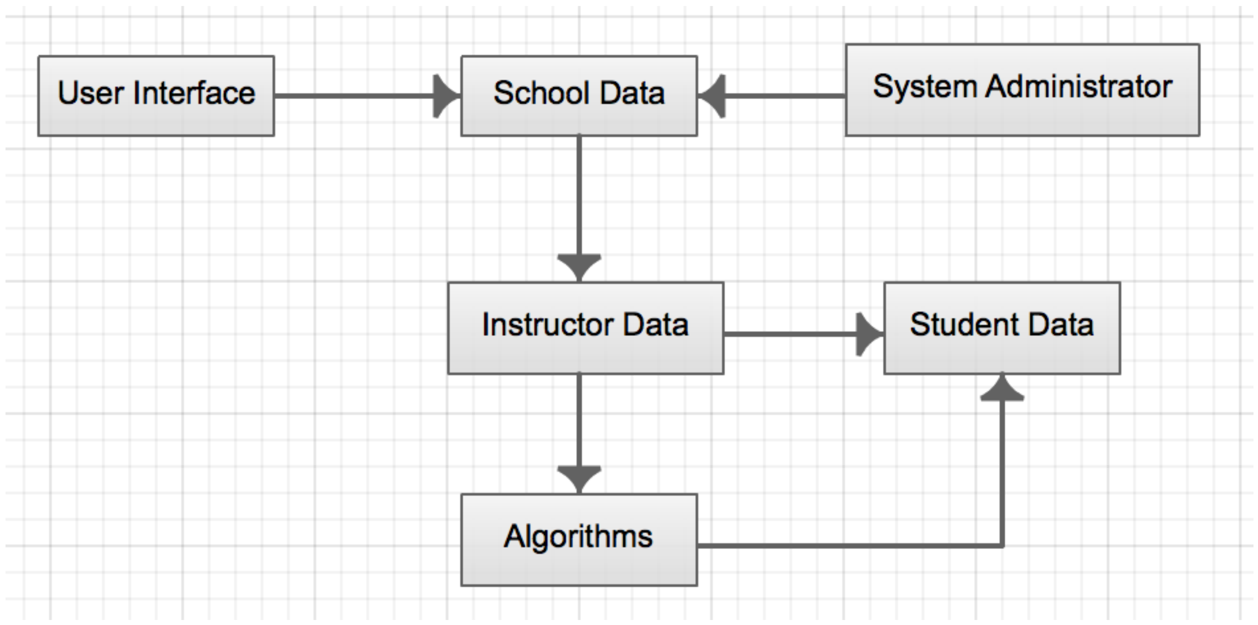
<u>New</u>


MyGradeBook
+ initialize : MyGradeBook
+ initializeWithFile (String) : MyGradeBook
+ initializeWithString (String) : MyGradeBook
+ processFile (String) : void
+ processString (String) : void
+ changeGrade (String, String, double) : boolean
+ average (String) : double
+ median (String) : double
+ min (String) : double
+ max (String) : double
+ currentGrade (String) : double
+ currentGrades : HashMap<String, Double>
+ assignmentGrade (String, String) : double
+ outputCurrentGrades : String
+ outputStudentGrades (String) : String
+ outputAssignmentGrades (String) : String
+ outputGradebook : String
Interfacer
+ main (String []) : void
+ welcome : void
+ optionsMenu : void
+ inputter : String
+ optionsInput (String) : void
+ initialize : void
+ initializeWithFile : void
+ initializeWithString : void
+ gradeBookMenu (MyGradeBook) : void
+ gradeBookInput (String, MyGradeBook) : void
+ pause : void
Course
+ studAssignMap : HashMap<Student, ArrayList<Assignment>>
+ newGradeBook : Course
+ newGradeBook (HashMap<Student, ArrayList<Assignment>>) : Course
+ equals (Object) : boolean
+ hashCode : int
+ changeGrade (String, String, double) : boolean
+ average (String) : double
+ median (String) : double
+ min (String) : double
+ max (String) : double
+ currentGrade (String) : double
+ getStudent (String) : Student
+ currentGrades : HashMap<String, Double>
+ assignmentGrade (String, String) : double
+ outputCurrentGrades : String
+ outputStudentGrades (String) : String
+ outputAssignmentGrades (String) : String
+ outputGradebook : String
Assignment
+ name : String
+ total : Double
+ weight : Double
+ score : Double
+ equals (Object) : boolean
+ hashCode : int
+ toString : String
+ changeScore (Double) : void
Student
+ userName : String
+ firstName : String
+ lastNmae : String
+ advisor : String
+ gradYear : int
+ equals (Object) : boolean
+ hashCode : int
+ toString : String

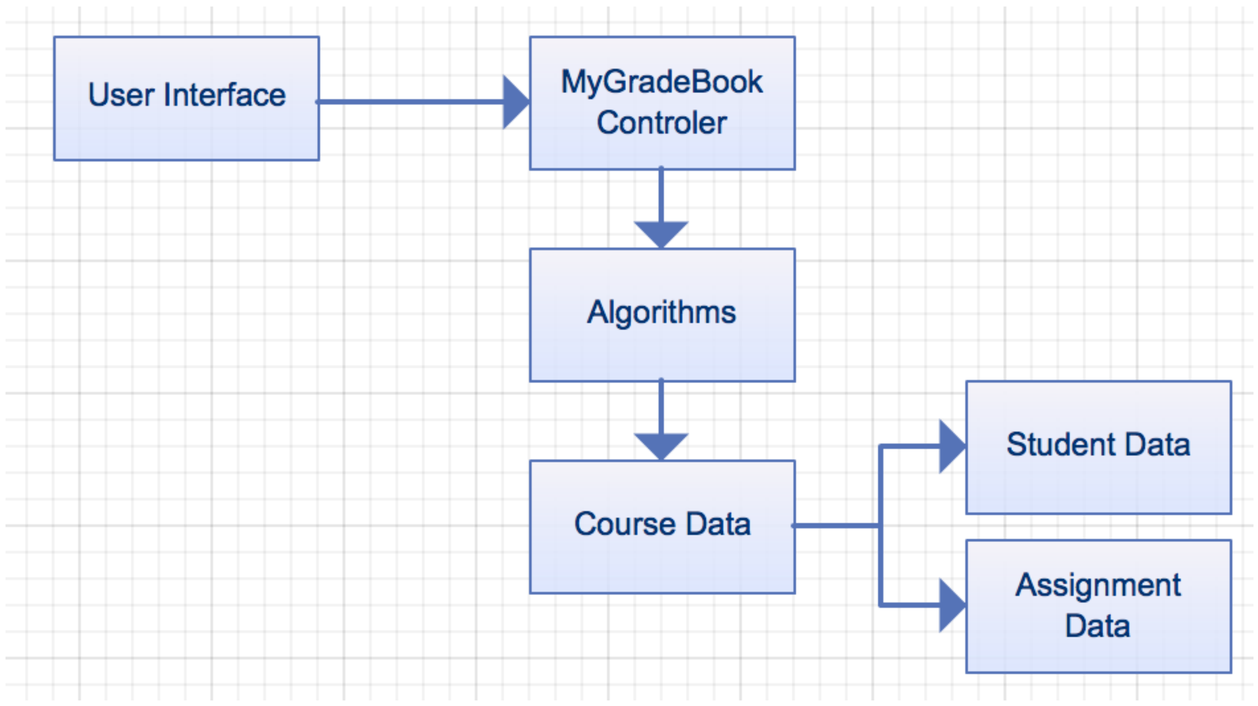
(these changes were made due to the time restraint of the assignment and the requirements stated for the MyGradeBook class that was provided to us)

VII. Module Dependency Diagram

Old



New



(*these changes were based off of the new structure of the classes*)

VIII. Team Member Contributions

1. Austin:
    a. Initial Discussion and Draw-Ups of Class Structure
    b. Use-Cases 2-7
    c. Final UML Diagram Construction
    d. Final Module Dependency Diagram Construction
    e. changeGrade method (course class)
    f. all output methods (course class)
    g. MyGradeBookTests.java
    h. StringComparator.java
    i. Collaberating on Presentation with Nick
    j. Updating Documents
2. Nick:
    a. Initial Discussion and Draw-Ups of Class Structure
    b. Initial conversations with use cases
    c. Introduction
    d. Worked on functional and non-functional requirements
    e. Use-Case 1
    f. User Interface
    g. UI Testing
    h. Collaborating with Charles on Input/Parsing Methods

    i. Creating Updated UML
    j. Creating Presentation
3. Charles:
    a. Initial Discussion and Draw-Ups of Class Structure
    b. Worked on functional and non-functional requirements
    c. Worked on creating logical use cases
    d. Input Methods
    e. Processing Methods
4. Chris:
    a. Discussion with functional and non-functional requirements
    b. Worked on method ideas for class structure
    c. Helped with UML Diagram
    d. BlackBox Testing