CS Lab 110

**Objects**

Over the course of this lab, you will work with a partner to design a portion of three larger projects. The three large projects will be designed with three groups:

|  |  |  |  |
| --- | --- | --- | --- |
| **GROUP** | **Part A** | **Part B** | **Part C** |
| 1 | A1, A2 | B3, B4 | C3 |
| 2 | A3, A4 | B5 | C1 |
| 3 | A5 | B1, B2 | C2 |

You may NOT send each other .java files at any point during this project, nor can you look at each other’s .java files. You can only pass .class files to each other. The .java files will only ever be pulled together during the final code interview, which will also be your evaluation.

This is a major lab, and will be worth 4 weighted points.

# Part A: PowerSchool

Over the course of this portion of the lab, you will be implementing a small Powerschool-like system in Objects. There will be four Object components of the system. Each Object will have its instance variables, constructors, accessors, and mutators described (in that order). All descriptive comments are **above** the relevant method headers.

# Part A1:

**Filename**: Grade.java

The lowest-level component of this system is the Grade. All comments refer to a method or variable below.

/\* Grades must be between 0.0 and 100.0 (inclusive) \*/

**double** *grade*;

/\* Audited classes are not counted in a Student’s GPA or class rank. A Student may audit

\* a class even if other Students in the same class are not auditing. \*/

**boolean** *audit*;

/\* Honors classes have a 5% boost to the grade. \*/

**boolean** *honors*;

// Basic constructor

**public** Grade (**boolean** honors, **boolean** audited);

// If the grade provided is out of range, the closest in-range grade is entered

**public** Grade (**boolean** honors, **boolean** audited, **double** grade)

/\* Returns the grade. If this is an honors grade, the reported grade is raised by 5% (max \* 105.0) \*/

**public** **double** getGrade();

**public** **boolean** honors();

**public** **boolean** audited();

// Sets the Grade to a number between 0.0 and 100.0, inclusive. If the grade provided is

\* out of range, the closest in-range grade is entered.

**public** **void** setGrade(**double** grade);

**public** **void** setAudit(**boolean** audit);

# Part A2:

**Filename**: Student.java

/\* When you declare the Student class, declare it as:

\* public class Student implements Comparable<Student>{

\* This is your promise to include the compareTo() method.

\*/

String *Name*;

**int** *gradYear*;

School *Academy*;

Course[] *schedule*; // Contains no more than ten courses.

**public** Student (String name, **int** gradYear, School Academy);

**public** String getName();

**public** **int** getGradYear();

**public** School getAcademy();

// Calculates GPA as average of classes

**public** **double** getGPA();

// Returns compareTo value for the GPAs of the two students

**public** **int** compareTo(Student someOtherKid);

/\* Adds course and returns true if successful.

\* Returns false if: course would brings classes to more than

\* 10, or student is already enrolled. \*/

**public** **boolean** addCourse(Course someCourse);

/\* Removes course and returns true if successful.

\* Returns false if: course was not on student’s schedule.

\*/

**public** **boolean** dropCourse(Course someCourse);

# Part A3:

**Filename**: Course.java

String *courseTitle*;

String *teacher*;

Student[] *enrolledStudents*;

Grade[] *Grades*; // Must correspond 1-1 with enrolledStudents.

**boolean** *honors*;

**public** Course(String Teacher, String title, **boolean** honors);

**public** String getTitle();

**public** String getTeacher();

**public** Grade getGrade(Student someStudent);

**public** int numberEnrolled();

**public** Student[] enrolledStudents();

// returns the top student in this course. (Can arbitrarily choose between a tie)

**public** Student topStudent();

**public** **void** setGrade(Student someStudent, **double** grade);

/\* Enrolls Student and returns true if successful.

\* Returns false if: course would give Student more than 10 classes, student is already

\* enrolled, or if enrollment would cause more than 20 students to be in the course.

\*/

**public** **boolean** enroll(Student someStudent);

**public** **void** unenroll(Student someStudent);

# Part A4:

**Filename**: School.java

String *Name*;

Student[] *enrolledStudents*; // Maximum of 300 Students per school

Course[] *schoolCourses*; // Maximum of 300 Courses per school

**public** School(String name);

// Returns name of school

**public** String getName();

// Returns number of enrolled students

**public** **int** enrolledStudents();

/\* Returns the array of courses. (\* I am allowing a shallow copy here, though a deep

\* would be better. Why is this?

\*/

**public** Course[] getCourses();

/\* returns Student with specified class rank. Since you have implemented compareTo,

\* you can use Arrays.sort to sort enrolledStudets.

\*/

**public** Student kidWithClassRank(**int** rank);

/\* Adds a student to the school. Returns false if the school is already full or if

\* Student is already enrolled at that school.

\*/

**public** **boolean** addStudent(Student someKid);

/\* Removes a student to the school. Also removes them from all of their courses.

\*/

**public** **void** unenrollStudent(Student someKid);

/\* Creates a class. Returns false if a class with exactly the same specifications

\* have already been made.

\*/

**public** **boolean** createClass(String Teacher, String title, **boolean** honors);

**public** **void** setName(String name);

/\* Enrolls Student in Course and returns true if successful.

\* Returns false if: Course not in school, Course would have more than 20 students, or

\* Student would have more than 10 Courses.

\*/

**public** **boolean** enroll (Student kid, Course someCourse);

/\* Unenrolls Student in Course and returns true if successful.

\* Returns false if: Course not in school, or Student not in Course.

\*/

**public** **boolean** unenroll (Student kid, Course someCourse);

# Part A5:

**Filename**: SchoolRunner.java

Make your own tester class that thoroughly tests your classes. The method should instantiate the various classes and test them out, with readouts providing the actions being taken, whether the Objects acted as expected, and a final percentage tally of tests passed. I use *your* SchoolRunner to test the code of other students, and test yours with the SchoolRunners that others have made.

You may wish to exchange runners with other students before your final submission in order to make sure that your code is all to spec!

Also, be sure to place your name in a comment at the top of this runner, as the runners will be moved around.

# Part B: FantasySports

Over the course of this lab, you will be implementing a sports-league simulation with betting. You should make your initial design specifications ON PAPER. No code, just a list of methods that each component will need.

Then keep track of the changes you make as you go. During the code interview, you will be asked to go over how you have had to modify your object design as the project moved along.

# Part B1a:

**Filename**: Player.java

A player must keep track of 5 stats in addition to the player’s name. Please get together *as a group* as write your initial specs together.

# Part B1b:

**Filename**: Coach.java

A coach will also have the same 5 stats, and will apply some sort of multipliers to the team members.

# Part B2:

**Filename**: Team.java

Teams have Players and Coaches, and can create some sort of strength-number based on the players and coach on the team. Teams must also have a PlayGame method that takes another team, and uses some combination of a random number and the strength numbers of the two teams to determine a winner of the match.

# Part B3:

**Filename**: Gambler.java

Gambler is an interface. The interface designates ways for a gambler, given a list of teams, to place bets, to win, and to lose. It should also have accessors for the amount of money the Gambler has and for the chronological win/loss record.

**Filename**: Bet.java

Bets are generated by Gamblers. Bets just contain an amount that has been bet, the names of the two teams playing, and the winner that the Bet designates.

# Part B4:

**Filename**: SmartGambler.java

**Filename**: DumbGambler.java

These Gamblers implement Gambler. DumbGambler chooses bets at random, while SmartGambler chooses bets and amounts to bet based on reasonable critera (this is up to you to define!) SmartGamblers should, if well-designed, considerably outperform DumbGamblers on average. Each Gambler should be asked to make a bet, and then later be told whether it won or lost. No Gambler has to keep track of more than one bet at a time! One bet, then it must be informed if it won or lost before making another bet.

Gamblers cannot bet more money than they have. A gambler can place a bet of 0.00, however, if it has no money.

# Part B5:

**Filename**: Fantasy\_\_\_\_\_\_.java

Fantasy\_\_\_\_ should be filled in with your sport of choice, such as FantasyFootball, FantasyBasketball, FantasyQuidditch, FantastyFoosball, FantasySpeedReading, FantastySleeping, etc. This should make a series of Teams in the league and populate it with players. It should then make various gamblers and run some simulated games. It should output the results of the games and the results from the betters to the screen. This is the runner of this project!

# Part C: MazeSolver

Over the course of this portion of the lab, you will be implementing a maze generator, a maze solver, and a maze animator. Again, you should make your initial design specifications ON PAPER. No code, just a list of methods that each component will need.

Then keep track of the changes you make as you go. During the code interview, you will be asked to go over how you have had to modify your object design as the project moved along.

# Part C1:

**Filename**: MazeGenerator.java

Generates a maze. (I recommend generating an ASCII maze because I feel it is easiest, but that may just be my own biases; you are welcome to use other approaches if you wish). MazeGenerator does not need to KNOW the solution, only that one exists.

# Part C2:

**Filename**: MazeSolver.java

Given a Maze generated by MazeGenerator, must create an optimized set of instructions to solve it.

# Part C3:

**Filename**: MazeAnimator.java

This is the runner. The runner will generate a maze, and continue to write the maze to the screen over and over with the Player moving one space per printing. It will do this until the maze is solved. It will then generate a new maze and repeat the process over and over until the executable it force-quit. Use java’s Thread.sleep(int milliseconds) to make your animations run at a reasonable speed!

# PLANNER

# B

**Player.java**

* Instance Variables
  + Start here…
* Accessors
  + Start here…
* Mutators
  + Start here…

**Coach.java**

* Instance Variables
  + Start here…
* Accessors
  + Start here…
* Mutators
  + Start here…

**Team.java**

* Instance Variables
  + Start here…
* Accessors
  + Start here…
* Mutators
  + Start here…

**Gambler.java**

* Instance Variables
  + Start here…
* Accessors
  + Start here…
* Mutators
  + Start here…

**SmartGambler.java**

* What criteria will SmartGambler use to generate bets?

# PLANNER

# C

**MazeGenerator.java**

* Instance Variables
  + Start here…
* Accessors
  + Start here…
* Mutators
  + Start here…

**MazeSolver.java**

* Instance Variables
  + Start here…
* Accessors
  + Start here…
* Mutators
  + Start here…