SchoolBud Milestone 3

CSSE 376 – Rose-Hulman Institute of Technology

John McCormack
Brian Padilla
Dharmin Shah

Table of Contents

Feature Completion Plan

Week 5

a. Dharmin Shah: Calculating the grades in a course and the grades needed to reach the target grade, and creating rubric. Test using the robust-worst case method. The test cases will include negative grades, minimum grades, maximum grades, and nominal grades, grades shy of minimal grade, grades shy of maximum grades, and grades over the max grades. Total of 49 cases will be generated for each component, that is 49 cases for calculating the current grade, and 49 cases for calculating the required grades.
b. Brian Padilla: Make a generic File reader that parses, loads, and stores information. Testing will be performed by using the robust worst-case method. Files will be created different combinations of letters, numbers, and symbols. Files will also be created using only letters, numbers, and symbols. This will help detect which characters cause errors in file names. With respect to reading and writing the file, the format and content will be checked. The read will be tested first because it will be used in the testing of writing the files.
c. John McCormack: The Scheduler displays the different permutations based on at least two given criteria. Testing using boundary-value. In order to thoroughly test the Scheduler class along with its respective helper classes, the test cases will employ many variations of situations that represent unique circumstances. ClassTester.java will simply provide the basic test cases in order to appropriately test initialization and all the getter and setter methods for its instance variables. The tests for the Scheduler class are much wider and more in depth. Because the main functionality of the scheduler is based of various permutation loops, the strategy taken for testing this will be based off of testing key, unique situations. For example, testing non-null initialization, a schedule with no classes, one class with zero hours, one class with one hour, two classes one hour only, two classes with one our each, and all these with one, two, three, and all days of the week (separated in various ways) ? and so on until enough special cases have been tested so as to show a broad range of code situation coverage. Then normal median cases will also be tested as well that are in between these special cases. The scheduler also implements various filtering capabilities such as the various densities of the classes in a daily schedule. The same aforementioned unique situations can be reused for testing the various filters with the added difference of that said filter.

Week 6

a. Dharmin Shah: Create the quarter class, loading a rubric from and saving a rubric to a file. Test cases include checking for valid and invalid quarter names, adding a list of courses, getting total credit hours (using robust-worst case technique), test the

calculated grade for the quarter based on the rubric (using robust-worst case technique), loading a rubric from an existing file, saving the rubric info to a file.

b. Brian Padilla: Create the structure for a simple GUI with dropdown menus, input boxes, and buttons. I will check to make sure that the values will change when selected for the dropdown menus. The input boxes will be verified by checking that the value inputted can be properly retrieved.
c. John McCormack: Finish the scheduler. In order to affectively test the various filters for the schedule permutations, there will be tests that run the special cases that a possible schedule can be. Examples of these special cases are no classes, one class, overlapping classes of same and different class sections, broken apart classes of same and different class section, all classes overlap at all places, and no classes overlap at any places.

Week 7

d. Dharmin Shah: Calculate the frequency of assignments. Test different frequency scenarios using robust-worst case. That is checking for adding negative number of assignments, no assignments, adding very few assignments over a course of time (for example 1 week), then adding a normal amount of workload, adding close to maximum, and adding maximum. Although there is not maximum defined, I will assume 20 assignments for a week to be the maximum build test cases based on that.
e. Brian Padilla: Continue adding popups for the various options within the dropdown menu. Also add functionality depending upon what classes are available. For the add pane, allow constructors for each option. Add more comboboxes as needed for desired functionality. Put action listeners for the dropdown options.
f. John McCormack: Create a simple trends graph. In order to implement a solid base foundation for the trending algorithm, I will start with very simple, short time-framed representation of user's grades. As I slowly add more tests and functionality to handle more variables, I will also be slowly making the time-frame and data points more extensive.

Week 8

a. Dharmin Shah: Work with John to finish the trends graph.
b. Brian Padilla: Develop interface for GUI and File Reader.
c. John McCormack: Finish the trends graph to take the max and min grade into account

Week 9

a. Dharmin Shah: Create GUI for scheduler.
b. Brian Padilla: Enable multi-language support
c. John McCormack: Create GUI for trends graph

Week 10

a. Dharmin Shah, Brian Padilla, and John McCormack: Give the final touches, add comments/documentation to code.

Coding Standards

The coding standards that the team has decided to follow are derived from the Sun Microsystems Coding Standards for Java (http://www.oracle.com/technetwork/java/codeconventions-150003.pdf):

File Organization

Package and Import Statements

The first non-comment line of most Java source files is a package statement. After that, import statements can follow. For example:

```
package java.awt;
import java.awt.peer.CanvasPeer;
```

Beginning Comments

All source files should begin with a c-style comment that lists the programmer(s), the date, a copyright notice, and also a brief description of the purpose of the program. For example:

```
/*
* Classname
*
* Version info
*
* Copyright notice
*/
```

Class and Interface Declarations

The following list describes the parts of a class declaration in the orders that they should appear:

a. Class or interface statement
b. Class/interface implementation comment, if necessary: This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment.
c. Class (static) variables: First the public class variables, then the protected ones, and then the private ones.
d. Instance variables: First public, then protected, and then private.
e. Constructors, if necessary.
f. Methods: These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.

Indentation

Four spaces should be used as the unit of indentation, or use Ctrl+Shift+f to auto-format the code in Eclipse.

Line Length

Avoid lines longer than 80 characters, or use Ctrl+Shift+f to auto-format the code in Eclipse.

Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles or use Ctrl+Shift+f to auto-format the code in Eclipse:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Comments

Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments should be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.
A block comment should be preceded by a blank line to set it apart from the rest of the code. Block comments have an asterisk "*" at the beginning of each line except the first.

```
/*
* Here is a block comment.
*/
```

Block comments can start with /*-, which is recognized by **indent**(1) as the beginning of a block comment that should not reformatted. Example:

```
/*
* Here is a block comment with some very special
* formatting that I want indent(1) to ignore.
*
* one
* two
* three
*/
```

Note: If you don't use **indent**(1), you don't have to use /*- in your code or make any other concessions to the possibility that someone else might run **indent**(1) on your code.

Single –Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format. A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in Java code:

```
if (condition) {
        /* Handle the condition. */
        ...
}
```

Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting. Avoid the assembly language style of commenting every line of executable code with a trailing comment. Here's an example of a trailing comment in Java code:

```
if (a == 2) {
        return TRUE; /* special case */
} else {
        return isprime(a); /* works only for odd a */
}
```

End-Of-Line Comments

The // comment delimiter begins a comment that continues to the newline. It can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code. Examples of all three styles follow:

```
if (foo > 1) {
        // Do a double-flip.
...
}
else
        return false; // Explain why here.
//if (bar > 1) {
//
//         // Do a triple-flip.
//         ...
//}
//else
        // return false;
```

Documentation Comments

Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters /**...*/, with one comment per API. This comment should appear just before the declaration:

```
/**
* The Example class provides ...
*/
```
class Example { ...

Notice that classes and interfaces are not indented, while their members are. The first line of doc comment (/**) for classes and interfaces is not indented; subsequent doc comment lines each have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have 4 spaces for the first doc comment line and 5 spaces thereafter. If you need to give information about a class, interface, variable, or method that isn't appropriate for documentation, use an implementation block comment or single-line comment immediately *after* the declaration. For example, details about the implementation of a class should go in in such an implementation block comment

following the class statement, not in the class doc comment. Doc comments should not be positioned inside a method or constructor definition block, because Java associates documentation comments with the first declaration *after* the comment.

Declarations

Number-Per-Line

- One declaration per line is recommended since it encourages commenting.
- In absolutely no case should variables and functions be declared on the same line.
- Do not put different types on the same line.

Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".) Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
void MyMethod() {
        int int1; // beginning of method block
        if (condition) {
                int int2; // beginning of "if" block
                ...
        }
}
```

The one exception to the rule is indexes of for loops, which in Java can be declared in the for statement:

```
for (int i = 0; i < maxLoops; i++) { ...
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;
...
func() {
        if (condition) {
                int count; // AVOID!
```

```
            ...
        }
        ...
}
```

Initialization

Try to initialize local variables where they are declared. The only reason not to initialize a variable where it is declared is if the initial value depends on some computation occurring first.

Class and Interface declaration

When coding Java classes and interfaces, the following formatting rules should be followed:

- No space between a method name and the parenthesis "(" starting its parameter list
- Open brace "{" appears at the end of the same line as the declaration statement
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{". However, the closing brace can also appear on the same line as the last statement of the block only when writing the test cases.

```
class Sample extends Object {
        int ivar1;
        int ivar2;
        Sample(int i, int j) {
                ivar1 = i;
                ivar2 = j;
        }
        int emptyMethod() {}
        ...
}
```

- Methods are separated by a blank line.

Statements

Simple Statements

- Each line should contain at most one statement.
- Do not use comma operator to group multiple statements unless it is for an obvious reason.

Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "{ statements }". See the following sections for examples:

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.

- Braces are used around all statements, even singletons, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

return Statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;
return myDisk.size();
return (size ? size : defaultSize);
```

if, if-else, if-else-if-else Statements

The if-else class of statements should have the following form:

```
if (condition) {
        statements;
}
if (condition) {
        statements;
} else {
        statements;
}
if (condition) {
        statements;
} else if (condition) {
        statements;
} else if (condition) {
        statements;
}
```

Note: if statements always use braces {}. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
        statement;
```

for Statements

A for statement should have the following form:

```
for (initialization; condition; update) {
        statements;
}
```

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

```
for (initialization; condition; update);
```

When using the comma operator in the initialization or update clause of a for statement, avoid the complexity of using more than three variables. If needed, use separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause).

while and do-while Statements

A while statement should have the following form:

```
while (condition) {
    statements;
}
```

An empty while statement should have the following form:

```
while (condition);
```

A do-while statement should have the following form:

```
do {
    statements;
} while (condition);
```

switch Statements

A switch statement should have the following form:

```
switch (condition) {
case ABC:
    statements;
    /* falls through */
case DEF:
    statements;
    break;
case XYZ:
    statements;
    break;
default:
    statements;
    break;
}
```

Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the /* falls through */ comment.

Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

try-catch Statements

A try-catch statement should have the following format:

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}
```

White Space

Blank Lines

Blank lines improve readability by setting off sections of code that are logically related. Two blank lines can be used in the following circumstances:

- Between sections of a source file
- Between class and interface definitions
- Between Methods.

One blank line can be used in the following circumstances:

- Between methods
- Between the local variables in a method and its first statement
- Before a block (see section 5.1.1) or single-line (see section 5.1.2) comment
- Between logical sections inside a method to improve readability

Blank Space

Blank spaces can be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space. Example:

```
while (true) {
        ...
}
```

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.
- All binary operators except (.)should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment (“++”), and decrement (“--”) from their operands. Example:

```
a += c + d;
a = (a + b) / (c * d);
while (d++ = s++) {
        n++;
}
prints("size is " + foo + "\n");
```

- The expressions in a for statement should be separated by blank spaces. Example:

```
for (expr1; expr2; expr3)
```

- Casts should be followed by a blank. Examples:

```
myMethod((byte) aNum, (Object) x);
myFunc((int) (cp + 5), ((int) (i + 3))
                                    + 1);
```

Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier—for example, whether it’s a

constant, package, or class—which can be helpful in understanding the code. The conventions given in this section are high level. Further conventions are given at (*to be determined*).

- Classes: Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words—avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).
- Interfaces: Interface names should be capitalized like class names.
- Methods: Methods should be verbs in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.
- Variables: Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should be short yet meaningful. The choice of a variable name should be mnemonic—that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.
- Constants: The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)

Programming Practices

Providing Access to Instance and Class Variables

Don't make any instance or class variable public without good reason. Often, instance variables don't need to be explicitly set or gotten—often that happens as a side effect of method calls. One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, if you would have used a struct instead of a class (if Java supported struct), then it's appropriate to make the class's instance variables public.

Referring to Class Variables and Methods

Avoid using an object to access a class (static) variable or method. Use a class name instead. For example:

```
classMethod(); //OK
AClass.classMethod(); //OK
anObject.classMethod(); //AVOID!
```

Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.

Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

Do not use the assignment operator in a place where it can be easily confused with the equality operator. Example:

```
if (c++ = d++) { // AVOID! Java disallows
        ...
}
```

should be written as

```
if ((c++ = d++) != 0) {
        ...
}
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler, and besides, it rarely actually helps. Example:

```
d = (a = b + c) + r; // AVOID!
```

should be written as

```
a = b + c;
d = a + r;
```

Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others—you shouldn't assume that other programmers know precedence as well as you do.

```
if (a == b && c == d) // AVOID!
if ((a == b) && (c == d)) // RIGHT
```

Returning Values

Try to make the structure of your program match the intent. Example:

```
if (booleanExpression) {
        return TRUE;
} else {
        return FALSE;
}
```

should instead be written as

```
return booleanExpression;
```

Similarly,

```
if (condition) {
        return x;
}
return y;
```

should be written as

```
return (condition ? x : y);
```

Expressions before '?' in the Conditional Operator

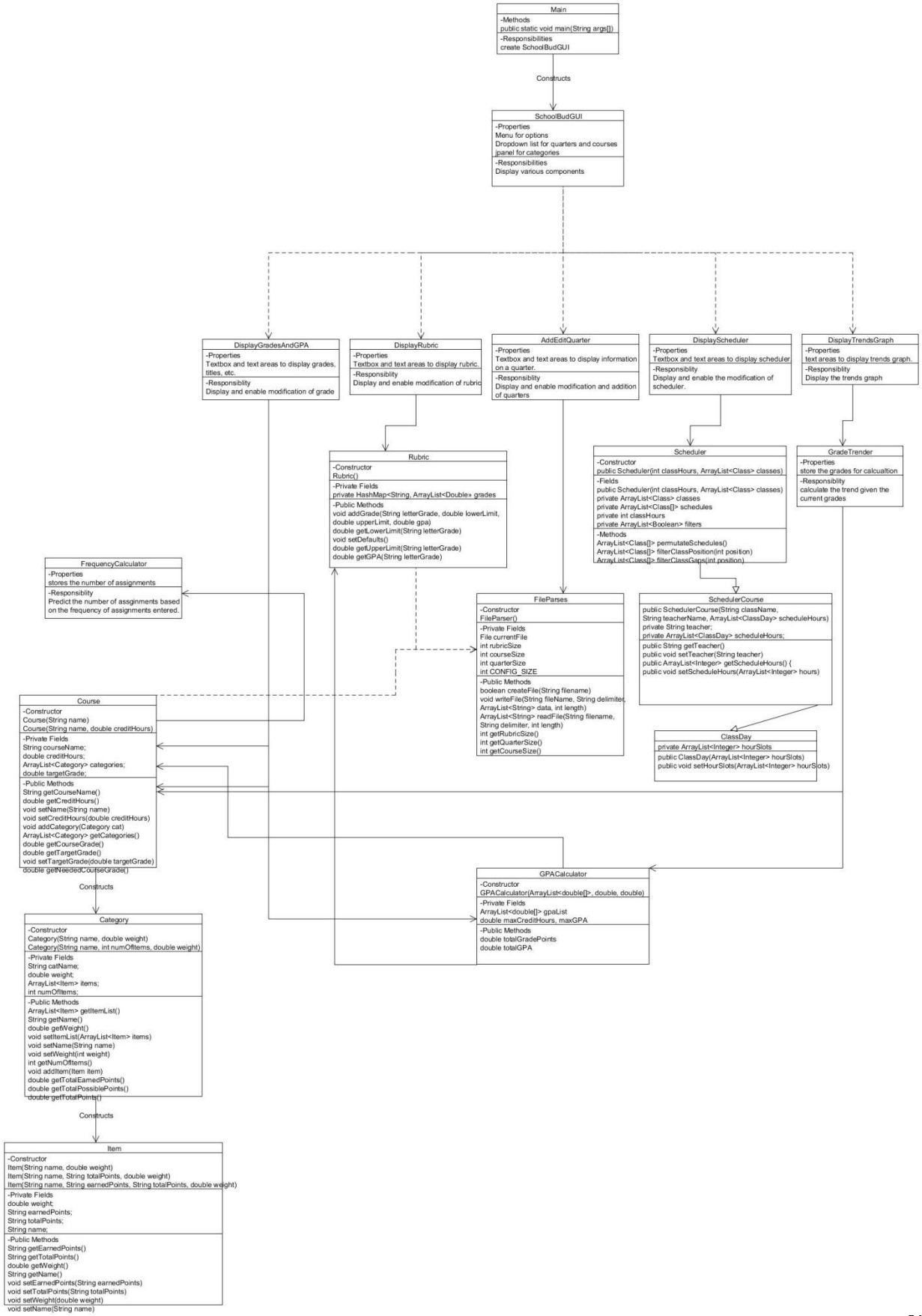
If an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesized. Example:

```
(x >= 0) ? x : -x
```

Code Coverage Tool

We are using an Eclipse plugin called eCobertura as our code coverage tool. Our team will primarily focus on Statement coverage for this project.

Class Diagram


Main
-Methods
public static void main(String args[])
-Responsibilities
create SchoolBudGUI
Constructs
SchoolBudGUI
-Properties
Menu for options
Dropdown list for quarters and courses
jpanel for categories
-Responsibilities
Display various components
DisplayGradesAndGPA
-Properties
Textbox and text areas to display grades, titles, etc.
-Responsibility
Display and enable modification of grade
DisplayRubric
-Properties
Textbox and text areas to display rubric.
-Responsibilty
Display and enable modification of rubric
AddEditQuarter
-Properties
Textbox and text areas to display information on a quarter.
-Responsibilty
Display and enable modification and addition of quarters
DisplayScheduler
-Properties
Textbox and text areas to display scheduler
-Responsibilty
Display and enable the modification of scheduler.
DisplayTrendsGraph
-Properties
text areas to display trends graph.
-Responsiblity
Display the trends graph
Rubric
-Constructor
Rubric()
-Private Fields
private HashMap<String, ArrayList<Double» grades
-Public Methods
void addGrade(String letterGrade, double lowerLimit, double upperLimit, double gpa)
double getLowerLimit(String letterGrade)
void setDefaults()
double getUpperLimit(String letterGrade)
double getGPA(String letterGrade)
Scheduler
-Constructor
public Scheduler(int classHours, ArrayList<Class> classes)
-Fields
public Scheduler(int classHours, ArrayList<Class> classes)
private ArrayList<Class> classes
private ArrayList<Class[]> schedules
private int classHours
private ArrayList<Boolean> filters
-Methods
ArrayList<Class[]> permutateSchedules()
ArrayList<Class[]> filterClassPosition(int position)
ArrayList<Class[]> filterClassGaps(int position)
GradeTrender
-Properties
store the grades for calcualtion
-Responsiblity
calculate the trend given the current grades
FrequencyCalculator
-Properties
stores the number of assignments
-Responsiblity
Predict the number of assginments based on the frequency of assignments entered.
FileParses
-Constructor
FileParser()
-Private Fields
File currentFile
int rubricSize
int courseSize
int quarterSize
int CONFIG_SIZE
-Public Methods
boolean createFile(String filename)
void writeFile(String fileName, String delimiter, ArrayList<String> data, int length)
ArrayList<String> readFile(String filename, String delimiter, int length)
int getRubricSize()
int getQuarterSize()
int getCourseSize()
SchedulerCourse
public SchedulerCourse(String className, String teacherName, ArrayList<ClassDay> scheduleHours)
private String teacher;
private ArrayList<ClassDay> scheduleHours;
public String getTeacher()
public void setTeacher(String teacher)
public ArrayList<Integer> getScheduleHours() {
public void setScheduleHours(ArrayList<Integer> hours)
ClassDay
private ArrayList<Integer> hourSlots
public ClassDay(ArrayList<Integer> hourSlots)
public void setHourSlots(ArrayList<Integer> hourSlots)
Course
-Constructor
Course(String name)
Course(String name, double creditHours)
-Private Fields
String courseName;
double creditHours;
ArrayList<Category> categories;
double targetGrade;
-Public Methods
String getCourseName()
double getCreditHours()
void setName(String name)
void setCreditHours(double creditHours)
void addCategory(Category cat)
ArrayList<Category> getCategories()
double getCourseGrade()
double getTargetGrade()
void setTargetGrade(double targetGrade)
double getNeededCourseGrade()
Constructs
GPACalculator
-Constructor
GPACalculator(ArrayList<double[]>, double, double)
-Private Fields
ArrayList<double[]> gpaList
double maxCreditHours, maxGPA
-Public Methods
double totalGradePoints
double totalGPA
Category
-Constructor
Category(String name, double weight)
Category(String name, int numOfItems, double weight)
-Private Fields
String catName;
double weight;
ArrayList<Item> items;
int numOfItems;
-Public Methods
ArrayList<Item> getItemList()
String getName()
double getWeight()
void setItemList(ArrayList<Item> items)
void setName(String name)
void setWeight(int weight)
int getNumOfItems()
void addItem(Item item)
double getTotalEarnedPoints()
double getTotalPossiblePoints()
double getTotalPoints()
Constructs
Item
-Constructor
Item(String name, double weight)
Item(String name, String totalPoints, double weight)
Item(String name, String earnedPoints, String totalPoints, double weight)
-Private Fields
double weight;
String earnedPoints;
String totalPoints;
String name;
-Public Methods
String getEarnedPoints()
String getTotalPoints()
double getWeight()
String getName()
void setEarnedPoints(String earnedPoints)
void setTotalPoints(String totalPoints)
void setWeight(double weight)
void setName(String name)