COMP1010 Major Assignment

Component B - Code style and document marks

Grading

Up to the mark your group gets in Component A are up for grabs in Component B. If your group gets 10/10 for component A, the group qualifies for 10 marks in Component B. If your group gets 8/10 for component A, the group qualifies for 8 marks in Component B.

Short story: As long as you do 20% or more work in a group of 4, you qualify for the entire marks your group qualifies for.

Long story: "Grade Note 2" in specs will also kick in. So if the group qualifies for 8/10 for component B and you did 15% work in a group of 4, you'd qualify for 75% of that, which is 6 marks. I know this is a bit confusing, but don't worry about it too much. As long as you do 20% or more work in a group of 4, you qualify for the entire marks your group qualifies for. In exceptional circumstances, where group of 3 is allowed, that must be a minimum of 27% of work.

Marking Guide:

- Commenting (Up to 1 marks)
  - 1 if appropriate level of commenting at all places
  - 0.75 if appropriate level of commenting at most places
  - 0.5 if appropriate level of commenting at some places
  - 0 if appropriate level of commenting at very few (or no) places

- Indentation (Up to 1 marks)
  - 1 if perfect and consistently-styled indentation everywhere
  - 0.75 if perfect and consistently-styled indentation at most places
  - 0.5 if perfect and consistently-styled indentation at some places
  - 0 if perfect and consistently-styled indentation at very few (or no) places

- Variable naming (Up to 1 marks)
  - 1 if variable names indicate their purpose (or are widely-accepted names) at all places
  - 0.75 if variable names indicate their purpose (or are widely-accepted names) at

most places
    - 0.5 if variable names indicate their purpose (or are widely-accepted names) at some places
    - 0 if variable names indicate their purpose (or are widely-accepted names) at few (or no) places

- Delegation (Up to 2 marks)
    - 2 if appropriate delegation used at all places
    - 1.5 if appropriate delegation used at most places
    - 1 if appropriate delegation used at some places
    - 0 if appropriate delegation used at very few (or no) places

- Class Design (Up to 3 marks)
    - 3 if class design is correct at all places
    - 2 if class design could be improved in some places
    - 1 if class design could be improved in several places
    - 0 if class design is practically non-existent

- Documentation (Up to 2 marks)
    - 2 if documentation explains the problem statement, design process, task allocation, contains well-designed UML diagrams really well (and in a compact manner - suggest a 2-3 page document)
    - 1.5 if documentation explains the problem statement, design process, task allocation, contains well-designed UML diagrams well (but is either too lengthy or too short, or missing some details)
    - 1 if documentation explains the problem statement, design process, task allocation, contains UML diagrams reasonably (but is either too lengthy or too short and also missing some details)
    - 0 if documentation does not explain the problem statement, design process, task allocation, contains well-designed UML diagrams satisfactorily

Code style guideline

1. Indentation and variable-naming

PICK A STYLE

Choose between one of the following and stick to it throughout your program.

- Java-style indentation + variable naming convention, OR,
- C-style indentation + variable naming convention,

FUNDAMENTAL RULE

The three control structures are:

1. Conditions
2. Loops
3. Functions

You go forward/to the right (a tab or 4 spaces) when you enter a control structure.

Java-style indentation

Conditions

```
if(exp) {
    //if body
}
```

```
if(exp) {
    //if body
}
else {
    //else body
}
```

Loops

```
while(exp) {
    //loop body
}
```

```
for(init; exp; update) {
    //loop body
}
```

Function

```
returnType funtionName(<parameters>) {
    //function body
}
```

Example of well-indented code:

```
int mystery(int[] a) {
    int count = 0;

    //one blank line to separate logical sections, if needed
    for(int i=0; i < a.length; i++) {
        for(int k=i+1; k < a.length; k++) {
            if(a[i] == a[k]) {
                count++;
            }
        }
    }

    //again, at most one blank line
    return false;
}
```

Example of decently-indented code with minor mistakes:

```
int mystery(int[] a) {
    int count = 0;

    //one blank line to separate logical sections, if needed
    for(int i=0; i < a.length; i++) {
        for(int k=i+1; k < a.length; k++) {
            if(a[i] == a[k]) {
            count++; //this is not at the right indentation level
            }
        }
    }

    //again, at most one blank line
    return false;
}
```

Example of poorly-indented code:

```
int mystery(int[] a) {


    //what's with all the extra emptylines!?


    for(int i=0; i < a.length; i++) {
     for(int k=i+1; k < a.length; k++) {
        if(a[i] == a[k]) {
                    return true;
                }
                }
        }
        return false;
}
```

Variable-naming

- camelCased
- finals are ALL_UPPERS

Examples:

```
boolean isHit = true; //tee-hee-hee :D

final int COUNT_SQUARES = 5;
```

C-style indentation

Conditions

```
if(exp)
{
    //if body
}
```

```
if(exp)
{
    //if body
}
else
{
    //else body
}
```

Loops

```
while(exp)
{
    //loop body
}
```

```
for(init; exp; update)
{
    //loop body
}
```

Function

```
return type funtion(parameters)
{
    //function body
}
```

Variable-naming

- Underscore-separated / snake_case
- finals are ALL_UPPERS

Examples:

```
boolean is_hit = true;
final int COUNT_SQUARES = 5;
```

2. Good variable names

Variables must convey their purpose clearly. Variables must not be too short/ too long.

Examples of good variable names:

```
boolean isHit = true;
int nStudents = 15;
double angle = Math.PI/2;
```

Examples of poor variable names (even if commented to indicate purpose):

```
int totoro = 15; //a represents number of students

//you will be reported to the faculty if any code/comment is offensive
boolean f_u = true;

double ngl = Math.PI/2;
```

3. Commenting

- Comments should be clear and within the screen.
- Multi-line comments should be immediately before the code to which they relate.

Example:

```
//assuming val is a floating-point number
/*
 * a holds the rounded-off value
 * by adding 0.5, any value which is of the form x.y,
 * where y < 0.5, will stay as x.z, thereby in the same
 * integer range. however, any value of the form x.y,
 * where y >= 0.5, will translate to (x+1).z,
 * where z < 0.5, jumping to the next integer range.
 * Then we cast it to intger, to get x or x+1
 */
int a = (int)(val+0.5);
```

- Single-line comments, when they relate to a statement, should be at the end of the statement to which they relate.

```
//assuming val is a floating-point number

int a = (int)(val+0.5); //a holds val casted to nearest integer
```

Note: this is the version where the reader isn't concerned with the underlying math.

- Over-commenting is bad!

Examples (over-commenting):

```
int a = 10;
a++; //increment a by 1
if(a < 10) { //if a is less than 10
    a/=2; //half it
}
```

- Commenting should explain the logic.

For example, for a code that reverses an array:

```
for(int i=0, k=data.length-1; i < data.length/2; i++, k--) {
    int temp = data[i];
    data[i] = data[k];
    data[k] = temp;
}
```

Following comments would be useful:

```
/*
 * We swap first item with the last,
 * second item with the second-last,
 * ...
 * item to the left of "middle line" with
 * item to the right of the "middle line"
 */
for(int i=0, k=data.length-1; i < data.length/2; i++, k--) {
    int temp = data[i];
    data[i] = data[k];
    data[k] = temp;
}
```

The code to swap two variables is trivial at our level and need not be commented.

4. Delegation

(In simple terms) Each function should have a dedicated functionality and the same logic (or extensive code) should not repeat across multiple functions.

To this end, if you have already implemented a functionality in one of the functions, you can always call it in another function instead of re-implementing that functionality all over again. The functions that we invoke, are colloquially called *helper* functions.

For example:

Consider the following code:

```
int countEven(int[] data) {
    int result = 0;
    for(int i=0; i < data.length; i++) {
        if(data[i]%2 == 0) {
            count++;
        }
    }
    return count;
}

int countOdd(int[] data) {
    int result = 0;
    for(int i=0; i < data.length; i++) {
        if(data[i]%2 != 0) {
            count++;
        }
    }
    return count;
}
```

The logic is the same in both functions. In fact, even and odd are exclusive functions of an integer. Hence, the code can, and should be, converted to:

```
int countEven(int[] data) {
    int result = 0;
    for(int i=0; i < data.length; i++) {
        if(data[i]%2 == 0) {
            count++;
        }
    }
    return count;
}

int countOdd(int[] data) {
    return data.length - countEven(data);
}
```

However, one must consider efficiency when delegating.

Consider the following functions:

```
int countPositives(int[] data) {
    int result = 0;
    for(int i=0; i < data.length; i++) {
        if(data[i] > 0) {
            count++;
        }
    }
    return count;
}

boolean allPositives(int[] data) {
    for(int i=0; i < data.length; i++) {
        if(data[i] <= 0) {
            return false;
        }
    }
    return true;
}
```

These should NOT be converted to:

```
int countPositives(int[] data) {
    int result = 0;
    for(int i=0; i < data.length; i++) {
        if(data[i] > 0) {
            count++;
        }
    }
    return count;
}

boolean allPositives(int[] data) {
    return countPositives(data) < data.length;
}
```

Because, as an example, if the array contains a thousand items, and the first item is NOT positive, countPositives will still iterate through all thousand items unnecessarily.

Another example of delegation:

Poor code:

```
public static int[] getPositiveItems(int[] data) {
    // Count the number of positive integers in the input array
    int count = 0;
    for (int num : data) {
        if (num > 0) {
            count++;
        }
    }

    // Create an array to hold the positive integers
    int[] positives = new int[count];

    // Populate the new array with
    // positive integers from the input array
    int index = 0;
    for (int num : data) {
        if (num > 0) {
            positives[index++] = num;
        }
    }

    return positives;
}
```

Good code:

```
// Helper function to count the
// number of positive integers in the array
public static int countPositives(int[] data) {
    int count = 0;
    for (int num : data) {
        if (num > 0) {
            count++;
        }
    }
    return count;
}

public static int[] getPositiveItems(int[] data) {
    // Use the helper function to count positives
    int count = countPositives(data);

    int[] positives = new int[count];
    int index = 0;
    for (int num : data) {
        if (num > 0) {
            positives[index++] = num;
        }
    }

    return positives;
}
```

5. Class Design

Each class should have a fixed responsibility and data should be kept in objects if suitable. Keeping calendar dates and time in Strings is not good as you cannot analyse it easily. Hence, dates should be in a `Date` class (containing day, month, year) and times of the day should be in `Time` class (containing hour, minute, second). Similar checks will be conducted to gauge the suitability of your classes.

Samples of submission

Code (1) that is good in terms of indentation and commenting

```
/*
    to determine if point (x, y) is inside the rectangle
    defined by top left corner at (minX, minY) and
    bottom right corner at (maxX, maxY),
    I will perform bound checking.
*/

boolean inside = false; //by default we'll assume it's outside

if(x >= minX && x <= maxX) {
    if(y >= minY && y <= maxY) {
        inside = true; //override because within bounds on both axes
    }
}
```

Note, we didn't comment the condition headers because the variable names make it self-explanatory.

Code (1) that is NOT good in terms of indentation and commenting

```
boolean inside = false;
if(x >= minX && x <= maxX) { //x is between minX and maxX
    if(y >= minY && y <= maxY) { //y is between minY and maxY
        inside = true; //update inside to true
    }
}
```

The comments on lines with conditional header are redundant.

Code (2) that is good in terms of indentation and commenting

```
boolean allUnique(int[] data) {
    /*
        the logic of my design and implementation is to
        compare each item in the array with every OTHER
        item (not itself), and returning false as soon
        as two items at different indices are the same
        in their value
    */

    if(data == null) { //to avoid NullPointerException
        return false;
    }

    for(int i=0; i < data.length; i++) { //pivot item
        /*
            compare pivot item against all items AFTER
            it (indiated by k=i+1) since by comparing
            an item A against another item B that is after it,
            we have compared B against A as well, and
            therefore don't need to do it again
        */
        for(int k=i+1; k < data.length; k++) {
            if(data[i] == data[k]) {
                return false;
            }
        }
    }

    /*
        the only way we can reach this control flow point
        is if no two items were the same
    */
    return true;
}
```

Code (2) that is not good in terms of indentation, or commenting

The comments are fairly redundant in this example.

```
boolean allUnique(int[] data) {
    if(data == null) { //if data is null
        return false;
    }

    for(int i=0; i < data.length; i++) { //for each item
        for(int k=i+1; k < data.length; k++)
        {
        if(data[i] == data[k]) { //if they are equal
                return false;
        }
        }
    }

    return true;
}
```

Code (1) that is good in terms of delegation

```
void setup() {
  size(600, 600);
  int x = width/3;
  int y = height/3;
  int dia = 100;
  drawHuman(x, y, dia);
}

void drawHuman(int x, int y, int dia) {
  strokeWeight(2);
  drawTorso(x, y, dia);
  drawLegs(x, y, dia);
  drawArms(x, y, dia);
  drawHead(x, y, dia);
}

void drawArms(int x, int y, int dia) {
  line(x, y+dia, x-dia, y+dia*.5);
  line(x, y+dia*1.1, x-dia, y+dia*.5);

  line(x, y+dia, x+dia, y+dia*.5);
  line(x, y+dia*1.1, x+dia, y+dia*.5);
}

void drawLegs(int x, int y, int dia) {
  line(x, y+dia*2, x-dia/2, y+dia*3);
  line(x, y+dia*1.8, x-dia/2, y+dia*3);
```

```
  line(x, y+dia*2, x+dia/2, y+dia*3);
  line(x, y+dia*1.8, x+dia/2, y+dia*3);
}

void drawTorso(int x, int y, int dia) {
  line(x, y, x, y+dia*2);
}

void drawHead(int x, int y, int dia) {
  drawHeadOutline(x, y, dia);
  drawEyes(x, y, dia);
  strokeWeight(2);
  drawNose(x, y, dia);
  drawMouth(x, y, dia);
}

void drawEyes(int x, int y, int dia) {
  strokeWeight(dia*.2);
  point(x - dia/4, y - dia/4);
  point(x + dia/4, y - dia/4);
}

void drawHeadOutline(int x, int y, int dia) {
  circle(x, y, dia);
}

void drawNose(int x, int y, int dia) {
  line(x, y-dia/10, x + dia/8, y+dia/10);
  line(x + dia/8, y+dia/10, x - dia/8, y+dia/10);
}

void drawMouth(int x, int y, int dia) {
  line(x - dia/5, y + dia/5, x + dia/5, y + dia/5);
  line(x - dia/5, y + dia/5, x - dia/4, y + dia/6);
  line(x + dia/5, y + dia/5, x + dia/4, y + dia/6);
}
```

Code (1) that is NOT good in terms of delegation

```
void setup() {
  size(600, 600);
  int x = width/3;
  int y = height/3;
  int dia = 100;

  strokeWeight(2);
  line(x, y+dia, x-dia, y+dia*.5);
  line(x, y+dia*1.1, x-dia, y+dia*.5);

  line(x, y+dia, x+dia, y+dia*.5);
  line(x, y+dia*1.1, x+dia, y+dia*.5);
  line(x, y+dia*2, x-dia/2, y+dia*3);
  line(x, y+dia*1.8, x-dia/2, y+dia*3);

  line(x, y+dia*2, x+dia/2, y+dia*3);
  line(x, y+dia*1.8, x+dia/2, y+dia*3);
  line(x, y, x, y+dia*2);
  strokeWeight(2);
  circle(x, y, dia);
  line(x, y-dia/10, x + dia/8, y+dia/10);
  line(x + dia/8, y+dia/10, x - dia/8, y+dia/10);
  line(x - dia/5, y + dia/5, x + dia/5, y + dia/5);
  line(x - dia/5, y + dia/5, x - dia/4, y + dia/6);
  line(x + dia/5, y + dia/5, x + dia/4, y + dia/6);
  strokeWeight(dia*.2);
  point(x - dia/4, y - dia/4);
  point(x + dia/4, y - dia/4);
}
```

Code (2) that is good in terms of delegation (but not in terms of commenting)

```
boolean isAlphabetic(String str) {
    if(str == null) {
        return false;
    }

    for(int i=0; i < str.length(); i++) {
        if(!isAlphabet(str.charAt(i)) {
            return false;
        }
    }
    return true;
}

boolean isAlphabet(char ch) {
    if(isUpperCaseAlphabet(ch) || isLowerCaseAlphabet(ch)) {
        return true;
    }
    else {
        return false;
    }
}

boolean isUpperCaseAlphabet(char ch) {
    if(ch >= 'A' && ch <= 'Z') {
        return true;
    }
    else {
        return false;
    }
}

boolean isLowerCaseAlphabet(char ch) {
    if(ch >= 'a' && ch <= 'z') {
        return true;
    }
    else {
        return false;
    }
}
```

Code (2) that is NOT good in terms of delegation

It does too many things in the same function

```
boolean isAlphabetic(String str) {
    if(str == null) {
        return false;
    }

    for(int i=0; i < str.length(); i++) {
        if((str.charAt(i) < 'a' || str.charAt(i) > 'z') &&
            (str.charAt(i) < 'A' || str.charAt(i) > 'Z')) {
            return false;
        }
    }
    return true;
}
```

Code (1) that is good in terms of variable names

```
float eyeCenterX = width/2;
float eyeCenterY = height/2;
float eyeWidth = width * 0.8;
float eyeHeight = height * 0.5;
float retinaDiameter = height * 0.4;

ellipse(eyeCenterX, eyeCenterY, eyeWidth, eyeHeight);
fill(0);
circle(eyeCenterX, eyeCenterY, retinaDiameter);
```

Code (1) that is NOT good in terms of variable names

```
float x = width/2;
float y = height/2;
float w = width * 0.8;
float h = height * 0.5;
float dia = height * 0.4;

ellipse(x, y, w, h);
fill(0);
circle(x, y, dia);
```

Code (2) that is good in terms of variable names

```
boolean distance(float x1, float y1, float x2, float y2) {
    float distanceX = abs(x1-x2);
    float distanceY = abs(y1-y2);
    float dX = distanceX * distanceX;
    float dY = distanceY * distanceY;
    float distanceBetweenPoints = sqrt(dX + dY);
    return distanceBetweenPoints;
}
```

Code (2) that is NOT good in terms of variable names

```
boolean distance(float x1, float y1, float x2, float y2) {
    float a = abs(x1-x2);
    float b = abs(y1-y2);
    float c = sqrt(a*a + b*b);
    return c;
}
```

Refactoring and debugging

While refactoring and debugging, it's easier if you spread out your logic across multiple statements rather than have it all in one convoluted statement.

For example, the following condition header checks if the String `str` contains `keyPhrase` (case-insensitive).

```
if(str.toLowerCase.indexOf(keyPhrase) >= 0`) {
    //...
}
```

However, it's doing too many things in one statement and it's difficult to identify, which part, if any, is buggy.

Instead, you can spread it across several statements, so you can inspect the status at each step.

```
String lower = str.toLowerCase();
int firstIndex = lower.indexOf(keyPhrase);
boolean found = firstIndex >= 0;
if(found) {
    //...
}
```

Code that is NOT good in terms of class design

```
class Session {
    public String startDay, startTime, endDay, endTime;

    public Session(String sd, String st, String ed, String et) {
        //...
    }
}
```

Code that is good in terms of class design

```
class Session {
    public DateTime start, end;

    public Session(DateTime start, DateTime end) {
        //...
    }
}

class DateTime {
    public Date date;
    public Time time;

    public DateTime(Date d, Time t) {
        //...
    }
}

class Date {
    public int day, month, year;

    public Date(int d, int m, int y) {
        //...
    }
}

class Time {
    public int hour, minute, second;

    public Time(int h, int m, int s) {
        //...
    }
}
```