Programming Project Design Notes

Overview

When you hand in a project to be graded there are certain criteria that I will use to determine your grade. This document is intended to help you understand how to get the best grade you possibly can.

The Requirements

On the bottom of each project will be a series of lines that look like this:

Grading

To receive full credit for this program the following must be submitted:

1)	Design for the algorithms utilized.	(25%)
2)	Well commented Syntactically Correct Java code.	(60%)
3)	Three (3) "runs" of the program.	(15%)

So, what does this mean, and what am I interested in?

Algorithmic Design

The **main goal** I am looking for in algorithmic design is very simple. I want to know that you applied thought to the program before you sat down and wrote the code. I will often refer to on the fly coding as composing, mainly because that is how it was presented to me, many moons ago. Interestingly enough a design that perfectly matches the code usually indicates a design that was done after the code.

I expect one of several different possibilities for some sort of algorithmic design. Possibilities include (but are not limited to:

- **Pseudo Code**
- **UML**
- **Flow Charts**

Here are brief descriptions of these three, I have seen others, and am more than willing to accept anything that fulfills the main goal listed above. Each method should have some design algorithm associated with it. Each class may or may not have some design, depending on whether there are methods in it, if there are methods please see the previous sentence.

Pseudo Code

By definition pseudo-code is any ad-hoc description of a design algorithm. Therefore everything from a few words, to a few sentences, to a snippet of code will suffice for pseudo-code. For example:

Pseudo-code to calculate a quiz average.

```
Read in number of quizzes
sum = 0
count = 0
while count < number of quizzes
        read in quiz grade
        sum = sum + quiz grade
```

count = count + 1
average = sum / number of quizzes
output average

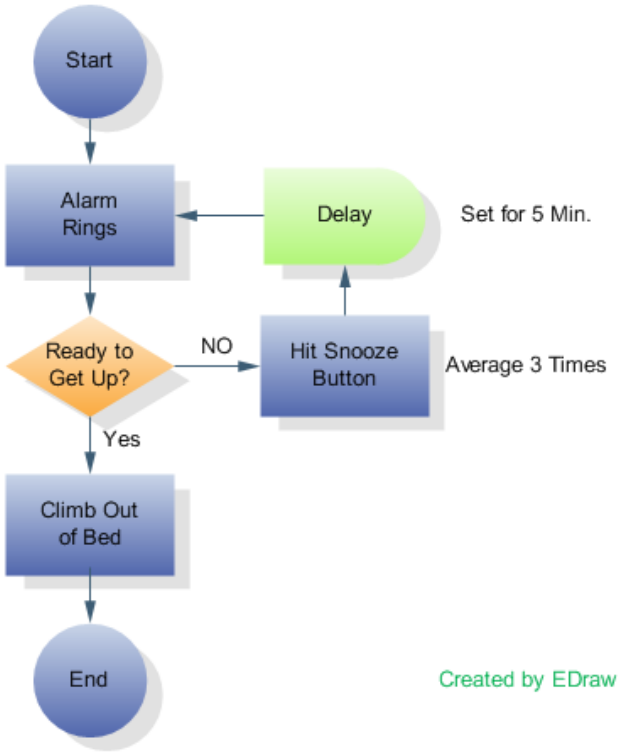
UML

UML or the unified Modeling Language is a way to describe object oriented designs. I am not going to give a tutorial on UML here. If you want to learn about UML there are several good tutorials on the web. UML divides a class into data and operations and displays them something like this:

Circle
itsRadius:double itsCenter:Point
Area():double Circumference():double SetCenter(Point) SetRadius(double) itsRadius:double itsCenter:Point

Flow Chart

A flow chart looks like this it denotes decisions, and operations:



Any of the above listed Algorithmic designs will work! My favorite is pseudo-code, because it more models the way I think. Your mileage may vary.

Well Commented Code

This subject borders on religious battle. I do believe that well documented code is about 50% comments. And I believe the following:

1) Every Project should have a header comment
2) Every Method must have a header comment
3) Every Class must have a header comment
4) Every sufficiently complex algorithm should be commented

I do believe that code can be somewhat self documenting as follows:

Which makes more sense to you?

```
int x;

int secondsTilTimeout;
```

You can add comments that are in any style, they just need to be there.

Here are examples of comments:

This is a Project Header (typically found on file with main, or init method):

```
//--------------------------------------------------------------------------
//
//  Tee Game
//      Solves the 5 Tee triangle game that you see in gift shops
//
//  Author:    Vic Berry
//  Date:      02/23/18
//  Class:     MET CS342
//  Issues:    None known
//
//  Description:
//    This program is a recursive implementation of a brute force
//    method of attacking this problem.  It stores the board in a
//    partially filled 5 x 5 array.  See comments below for more
//    information.
//
//  Assumptions:
//    The board will start with the top location empty.
//
```

This is a method header:

```
/////////////////////////////////////////////////////////////////
///   Queue_add            (Add item to queue)                  ///
///    Input : data to be inserted                              ///
///    Output: None                                             ///
///    Returns true if insert is successful, false otherwise   ///
///                                                             ///
/////////////////////////////////////////////////////////////////
bool queue_add(Item data)
```

This is a class header:

```
//
// Class:  myClass
//
// Description:
//     All the things needed to support my class.  My class is a class that
//     does class like things.  It has methods, and data, etc.
//
```

Three runs of program

Make sure you run the program as many times as the requirements state, in addition try to show all the functionality that is required of the project. Highlight error correction, good runs, and bad runs.

Code Development

As hard as I try sometimes there will be ambiguities in the design specifications I give you for assignments. If you encounter one of these, write a comment in the code that explains why you did what you did. If your logic is sound, and the assumptions are valid, there will be no points taken off.

Break the project down into small modules that are easy to code, use them as building blocks to build up to the whole program. This makes the code easier to debug, and generally builds better code. This will also highlight the reason that you did a design document before you started writing the code.

Do not be afraid to re-implement code. It is natural to start out with what you thought was a great idea, only to find out that the implementation is too complex, and you are concerned that you will build something that may or may not work and becomes very fragile. If you are worried that you can't change the code because you might break it, it may be time to consider rewriting it.

Building small modules helps you get out of trouble quickly and suffer a small penalty in time in the event the module was not what you wanted it to be.

Remember, small corrections are much easier than big ones.