Software Development Standards

❖ General

The purpose of software development standards is to facilitate the software development process. Standards help developers understand and maintain application systems. Computers can execute code (instructions) that does not conform to standards as easily as they can execute code developed with rigorous standards. However, code developed without standards can be much more difficult for people to understand, making it difficult to change and enhance software.

Software development standards are typically adopted by an organization for the reasons mentioned above and software developers employed by the organization are required to write code conforming to the standards.

Standards, however, differ from organization to organization depending on the needs and preferences of the organization and there is no one "right" set of standards.

In this class, students will be expected to develop applications in conformance with the standards outlined in this document. Following the standards will be part of the grading criteria for each project. Please note that these standards are not always the same as those used in your textbook. For this class, the standards topics are:

- **User Interface Design Standards**
- **Coding Standards**
  - **Documentation: Comments**
  - **Naming convention**
  - **Spacing Conventions**
  - **Declaration Conventions**
- **Test Plan**
- **Project Documentation**

Detailed information on these standards follows.

➢ User Interface Design Standards

- To the user of your application, the Interface is the application. Make your interface as efficient and **user-friendly** as possible.
- Try to limit the keystroke interaction where applicable (avoid too many confirmation questions such as "Are you ready to continue? Y/N)
- Screen Layout (structural). Whitespace is important, as a cluttered screen is difficult to read and interact with.
- Use identifying labels for data displayed on the screen,both input and output..For example, if the screen displayed 47. How does one interpret the data? Help the end-user and display an identifying label such as
  - Age: 47
  - Years Employed: 47
  - Miles to destination: 47
  - Now the data makes more sense.
- Correct spelling of words displayed on the screen.

CODING STANDARDS

➢ DOCUMENTATION: Comments

Comments help other developers understand the code that has been written and help the original developer remember his/her own work. They help developers resolve problems with the application once it is up and running and once it needs revision. Comments assist with **what** the program is doing, not **how**. You may use statements from your pseudocode to guide you, and be sure to describe any complexities, but don't go overboard. A comment on every line is not efficient and may reflect poor quality of code.

Block comments are used to provide descriptions of files, class, methods, and data structures. 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 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.

1. Required **File** documentation placed at the very beginning of the file

```
// PROG 110, Fall 2011
// File: DisplayText.cs
// Class: FirstClass
// Author: <your name.
// Date: <today's date>
```

If you are submitting a revised file, add the following:

```
// Revised: < date when submitted>
```

2. Required **class** documentation placed right above the class name – state the purpose of the class

```
/*****************************************
 * Purpose: This class will display quotes
 * commonly used by Jean-Luc Picard
 * from ST:TNG
 * ***************************************/
```

Note: comments are not designed to express outcomes from the assignment, but to express the application's functionality

3. Required **method** documentation placed right above the method heading

```
/*
    *  Method: calculateTax
    *  Purpose: This method will calculate the tax costs
    *
    * Input: subtotal (float)
    * Output: tax amount (float)
 */
```

4. Internal documentation or comments

Comes from your pseudocode, for groups of statements

```
// display quotes
Console.WriteLine("Make It So");
Console.WriteLine("Do It");
Console.WriteLine("Engage");

//pause application
Console.Write( "Press any key to continue. ");
Console.ReadKey();
```

➢ Naming Conventions

1. Use meaningful, descriptive words as identifiers.
2. Do not use abbreviations for identifiers unless it is a well-known abbreviation.
3. With the exception of variables used as iterators in a `for` loop, do not use single character identifiers.
4. Use Pascal casing (first character of all words are uppercase, all other characters are lowercase) for the following:
    a. Class and type identifiers
    b. Method names
    c. Namespace identifiers
    d. Property names
5. Use Camel casing (first character of all words, except the first word, are uppercase, all other characters are lowercase) for the following:
    a. Local variable identifiers
    b. Object identifiers
    c. Private data embers
    d. Parameters
6. Filename should match the class name
7. Use all uppercase characters to name constants

➢ Spacing Conventions

1. Use tabs instead of spaces for indentation.
2. Use white space (one blank line) to separate and organize logical groups of code.

3. Place curly braces on a new line
4. 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 can be declared in the `for` statement:

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

5. Simple statements: Each line should contain only one statement.
   `num++; cost--; // AVOID!`
   Avoid long lines of code; lines should not exceed 80 characters.

6. If a statement must be split over multiple lines, use indentation to improve readability.

```
Console.Write("This may be a long sentence, " +
              "\nWhich separated, makes it easier to read" +
              "\nbut is still one statement.");
```

7. Indent, indent, and indent. Repetition, decision structures, and body of functions are all to be indented. Use curly braces to identify each block of a control structure, such as an `if-else` or `for` statement, even single statements. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

➢ Declaration Conventions

1. Minimize scope level of variables. Pass variables instead of declaring class level variables.
2. Use the simplest data type (int, float)
3. Declare and initialize local variables. Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.
4. Use the `const` keyword to define constant values.

- **PROGRAMMING TECHNIQUES**

- Proper indentation and spacing are critical in making programs easy for people (like me) to read.
- Place only 1 statement per line.
- Use blanks around operators to clarify grouping.
- When positioning braces, pick a style that suits you, then use it consistently.
- **for** structure:
  - Place only expressions involving the control variable in the initialization and increment/decrement section.
  - Do not change the value of the control variable inside the body of the loop.
  - Rethink and rewrite any loop that needs to use **break** or **continue**.
- **switch** structure:
  - Remove redundant code to outside the decision structure
  - Always include a **default** case clause and make it the last clause in a switch statement. Document the "default" case
- **if/else if** structures:
  - Remove redundant code to outside the decision structure
  - Include an else for all other values not specifically tested
- **while, do-while** structures:
  - Rethink and rewrite any block that needs to use **break** or **continue**.
- **User-defined methods**
  - Arguments will be passed by value, unless two or more values must be modified (Note: protect all data from unnecessary changes – use 'const" where applicable.)

Test Plan

- Run through your program. No run time error messages should pop up.
- Invite a friend to test your application for ease of use. They may provide some helpful comments.
- Test the application without entering any data.
- Test all branches of each decision structure.
- **When doing data validation:**
  - Test the application with both valid and invalid data.
  - Test the application by entering letters where numbers are expected.

Project Documentation

- **Cover memo** (example attached) with the following **required** information:
  - Use a memo format (see Word's memo templates) and 12-point type-size.
  - Correct English grammar and spelling.
  - Timesheet showing time spent on the project as part of the Software Development Life Cycle. Please make this as accurate as possible – there is no right or wrong answer as this is to assist with your time management.
  - In paragraph format, explain how you tested your application.
  - Offer suggestions or recommendations for future "application" enhancements (e.g. value-added functionality).
  - Optional: If your project includes "extra features"; explain what they are and how to use them.
- **Files**
  - All required files are in one file folder. Make sure to test that you are not submitting a shortcut to a folder on your home computer.
  - File folder has the required name (solution folder, project folder)
  - Executable file has the required name.
  - Zip file name submitted follows the format of LastNameProjectX.zip (e.g. UnwinProject1.zip)

❖ **Project Documentation - Cover Memo Example**

MEMO

To: Sylvia Unwin, Instructor
From: James T. Kirk, PROG 110
Date: April 15, 2010
Subject: Project 1

Time Sheet

Date	Task	Estimated Hours	Actual Hours
4/8/10	Design (Plan the application)	2	3
4/9/10	Write the pseudocode	1	3
	Build the interface	1	1
4/10/10	Code	2	3
4/11/10	Test/Debug	3	3
4/14/10	Assemble the documentation	.5	1
	Total	**9.5**	**14**

Must be included

This project involved developing a retirement planning application as specified in the requirements document for Project 1. Calculations based upon user input were performed to assist the user in planning his/her investment strategy for retirement income.

All data entry requirements were met, and the user can easily exit this application. The test plan included the following:
Investment input/calculation: I entered positive numbers, negative numbers, and no numbers and the program had one problem with no numbers that I did not know how to handle. Do we know how to check for that?
Investment summary: I ran the program with various values to make sure the placement of the data was user-friendly and not overly crowded.

With this project I learned that laying out the input screen is very tedious, and that the IDE is not as easy to learn as I thought. I also noted that this project took longer than I imagined it would. Maybe next time, I will plan for more time and not be so rushed.

I recommend that the requirement to have the screen cleared after entering the data, and use a second screen for further summary information. This approach would work best to eliminate user confusion and make this application as user-friendly as possible.