

**Adora Apparels Accounts Management System**

Coding standards document

Software Engineering Project 2015

Project ID: SEP-006

Submitted by:

IT 13070236 – Senevirathna J.N.

IT 13060572 – Ekanayake E.M.S.S.

IT 12006146 – Jayasinghe P.S.

IT 13063856 – Pathmalal K.P.S.U.

Submitted to:

…………………………..

Ms. Asanthika Imbulpitiya

7th August 2015

Date of submission

**Introduction**

There can be many solutions for a single problem. The output of each and every solution can be correct all the time in all the solutions. Having the correct output to a given set of inputs does not imply that the solution is ideal. The interior of the solution should be analyzed to see the quality of the solution.

In a software application, interior can be derived through analyzing the code. Analyzing code structure gives an idea of the standard of the application. Here in this document, the code structure is analyzed under different categories such as code layout, subroutines, naming conventions and comments.

The code structure is analyzed for C# and XAML because those were followed with the Software Engineering Project in the 1st semester of the final year. By looking at the different categories an idea about the whole code format followed in the project can be derived.

**Table of Contents**

1. **General**
   1. Code Formatting
   2. Indentation
   3. Initialization
2. **Project Dependent Standards**
   1. Naming Conventions
   2. File Organization
   3. Specifications for Error Handling
   4. Revision and Version Control
   5. Code Quality Checking
   6. Include Files and Header Files
   7. Comments and White Spacing
   8. Standardization of the Development Environment
3. **File and Module Guidelines** 
   1. Source Code Grouping
   2. Header Files
   3. File Naming Guidelines
   4. Method Declaration and Comments
4. **Constants**
5. **Subroutines**
   1. Naming Subroutine
   2. Return Statement
6. **Comments**
7. **Code Layout**
   1. Error Handling
   2. Exception Handling
8. **Naming Convention for Identifiers (Variables, Constants, and Subroutines)** 
   1. Select clear and Meaningful Names
   2. Naming Subroutines (Verb and Object)
   3. Naming Constants, variables (noun)
   4. Naming Boolean identifiers (verb and ((object or adjective))
   5. Abbreviations
9. **Misc. Rules for Coding**
   1. Conditionals and comparisons
   2. Binding time of variables and values
   3. Go-tos, pointers, and issues of clarity
   4. Strive to develop clear code
   5. Use libraries when available
10. **Modularization**
    1. The information hiding and scope of variable
    2. High Cohesion Low coupling
    3. Clean Interfaces
11. **General**
    1. **Code formatting**

* Only one namespace must be declared for a single line.
* There must be not more than one class per single file.
* Curly braces ({ }) in must be started and ended with a new line.
* Namespace “using” statements must be placed together at the top of file. The .NET namespaces should be grouped above custom namespaces.

using System;

using System.Collections.Generic;

namespace Adora.Functions;

//Documentation Block Here

class Person

{

// Documentation Block Here

public void printDetails()

{

// all contents of function

// must be indented four spaces

}

}

* 1. **Indentation**

The code must be properly indented and formatted in order to increase the readability and the understandability. The IDE used in this project is Visual studio and it is used only in windows platform. Therefor the indentation issues that rises with tab space is not effected.

* Throughout the project a single tab space is used in indentation.

//Documentation Block Here

class Abc

{

// Documentation Block Here

public void defGhi()

{

if()

{

}

}

}

* 1. **Initialization**

Local variables should be initialized 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.

**2. Project Dependent Standards**

**2.1 Naming Conventions:**

Since C#/WPF technologies used to build our project. Very long descriptive names are used with camel case to define all the variables. That makes the code more readable and Visual Studio 2013’s intelli-sense feature helps with automatically filling when using those variables

Ex: int customerID

**2.2 File Organization:**

Layered architecture is used to this project. All the views and data access objects are separated according with class names. Each specific section of the application has its own view and its controller. All the necessary data are taken from the data access classes using the data access objects. For every validation there’s a validation class. All the validation classes, Data access classes and View + Controller classes have their own directories for separation.

**2.3 Specifications for Error Handling:**

Specific validation methods are used to validate all the data inputs before sending to the database classes. When handling errors and exceptions very specific exception handling methods are combined with data access classes. All the generic I/O exceptions and project specific errors and exceptions are handled and tested.

**2.4 Revision and Version Control:**

GIT version controlling technology is used from github to version controlling of this application.

**2.5 Code quality checking:**

To achieve the code quality .NET compiler platform is used which open source compiler framework from Microsoft. To achieve the maximum performance of the application FxCop analyzer is used. Since this is very database intensive application all the bottlenecks of database access classes are analyzed and fixed using FxCop.

**2.6 Include files and Header files:**

Begin of every class only the essential header files are imported. All the unnecessary and redundant import libraries are removed to achieve maximum readability and the maximum resource utilization.

**2.7 Comments and White spacing:**

Every class and every single method is commented extensively commented to achieve maximum understandability to other programmers so the application can further develop and do changes to achieve more function in the future.

By using tabs and new lines code is optimized for maximum readability.

**2.8 Standardization of the development environment:**

To develop the Adora application Visual studio 2013 ultimate is used with C#/WPF technologies. To code testing and performance analysis .NET compiler platform and FxCop analyzer are used. To compile the application standard .NET compiler that comes with Visual Studio 2013 is used.

**3. File and Module Guidelines**

**3.1 Source Code Grouping:**

Each source code file belongs to the specific part of the Application. Since for this application is developed using the layered architecture every source file is belong to the View+Controller, Validation or Data access sections. By using the code separation and reuse objected orientation is extensively used for building application

**3.2 Header files:**

Begin of each source fil all the necessary header libraries are imported that the class is essential to perform its operation. All the unnecessary include statements are removed using FxCop analyzer for maximum readability and for the source code and enhance the performance of the application.

Most of the import statements are generated by the Visual Studio and belong to the standard .NET framework

Examples of most used header libraries/namespaces

* System.Collections.Generic Namespace

(The System.Collections.Generic namespace contains interfaces and classes that define generic collections, which allow users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections.)

* System.Collections.ObjectModel namespace

(The System.Collections.ObjectModel namespace contains classes that can be used as collections in the object model of a reusable library. Use these classes when properties or methods return collections)

* System.Globalization namespace

(The System.Globalization namespace contains classes that define culture-related information, including language, country/region, calendars in use, format patterns for dates, currency, and numbers, and sort order for strings. These classes are useful for writing globalized (internationalized) applications. Classes such as StringInfo and TextInfo provide advanced globalization functionalities, including surrogate support and text element processing.)

* System.IO namespace

(The System.IO namespace contains types that allow reading and writing to files and data streams, and types that provide basic file and directory support.)

* System.Linq Namespace

(The System.Linq namespace provides classes and interfaces that support queries that use Language-Integrated Query (LINQ).)

* System.Text Namespace

(The System.Text namespace contains classes that represent ASCII and Unicode character encodings; abstract base classes for converting blocks of characters to and from blocks of bytes; and a helper class that manipulates and formats String objects without creating intermediate instances of String.)

* System.Threading.Tasks

(Represents an asynchronous operation.)

* System.Windows Namespace

(Provides several important Windows Presentation Foundation (WPF) base element classes, various classes that support the WPF property system and event logic, and other types that are more broadly consumed by the WPF core and framework.)

* using System.Windows.Controls;
* using System.Windows.Data;
* using System.Windows.Documents;
* using System.Windows.Input;
* using System.Windows.Media;
* using System.Windows.Media.Imaging;
* using System.Windows.Navigation;

**3.3 File Naming Guidelines:**

Every source file named according to their class name using the camel case. No spaces, non- alpha numeric characters are used in source files for the consistency.

Ex: Validation class – Validation.cs

**3.4 Method Declaration and comments:**

Beginning of every method and each complex source code lines are extensively commented for the future improvements of the software and for the understandability to other programmers.

**4**. **Constants**

Constants are immutable values which are known at compile time and do not change for the life of the program. Constants are declared with the const modifier. Only the C# built-in types (excluding System.Object) may be declared as const. For a list of the built-in types, see Built-In Types Table (C# Reference). User-defined types, including classes, structs, and arrays, cannot be const. Use the read-only modifier to create a class, struct, or array that is initialized one time at runtime (for example in a constructor) and thereafter cannot be changed.

C# does not support const methods, properties, or events.

The enum type enables you to define named constants for integral built-in types (for example int, uint, long, and so on).

Constants are used in the project in the following format.

class FixedOverheads

{

public const int months = 12;

//rest of the code

}

**5. Subroutine**

**5.1 Naming subroutine**

The naming of the function and methods must be done according the camelcase convention. The first letter must be a simple letter. The name should be meaningful and understandable. Only when naming classes, the first letter must be capital letter.

//Documentation Block Here

class Person

{

// Documentation Block Here

public void printDetails()

{

// all contents of function

// must be indented four spaces

}

}

If the same set lines of code is replicated more than once, it should be made subroutine. Whenever appropriate, sections of code which are almost the same, except for the identity of some variables, should be made subroutines with parameters to allow for the differing variables.

**5.2 Return statement**

A return statement with a value should not use parentheses unless it is an expression that requires parentheses or the parentheses provide clarity.

return;

return age;

return firstName + “ ” + lastName;

**6. Comments**

There are 2 ways of commenting in C#. Those are,

* Single Line Comments (by using /\*\*/)
* Block Comments (by using //)

Comments play a huge role in the code. The main usage of using is to increase the understandability of the code. The comments are placed in a way that the code is understandable and meaningful. The following format is used in commenting.

<block comment>

<code>

<inline comment>

class ClassName

{

<inline comment>

<code>

<inline comment>

Method(){

<inline comments>

}

}

/\*

Author : ..

Class : ..

\*/

<code>

//Class ClassName is used for ..

class ClassName

{

//attribute details

<code>

//method is used for ..

Method(){

//operation details

}

}

1. **Code layout**

**7.1 Error handling**

* In error handling the error message must be properly formatted with appropriate messages, title, icons and colors.
* The buttons of the message box must be appropriate to the situation.

System.Windows.MessageBox.Show("Need to fill all the fields", "Empty Fields", MessageBoxButton.OK, MessageBoxImage.Warning);

**7.2 Exception Handling**

Exception handling with try-catch

* When using try catch block, the supper class exceptions shall not be captured. Instead of that the specific exceptions must be captured.
* The exception itself must not be shown to the user as a message; instead a proper message should be displayed.
* The exception details shall be sent to the error log with appropriate details.
* Nested try-catch blocks should be avoided.

Try

{

//the code

}

catch (FormatException formatException)

{

}

catch (InvalidCastException invalidCastException)

{

}

catch (OverflowException overflowException)

{

}

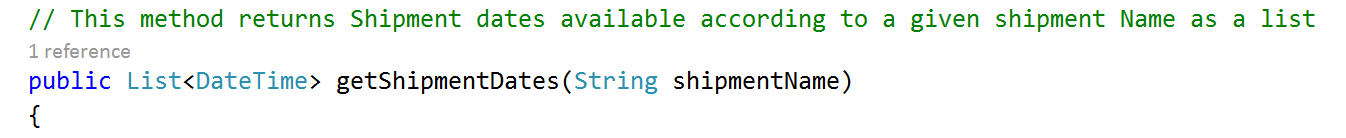
**8. Naming Convention for Identifiers (Variables, Constants, and Subroutines)**

**8.1 Select Clear and Meaningful Names**

The most important consideration in naming a variable, constant, or subroutine is that the name fully and accurately describes the entity or action that the structure represents. Clear, complete, and meaningful names make the code more readable and minimizes the need for comments. For example; suppose a subroutine called “*processInputLine*” calls “*pushInputCharacter*.” If “*pushInputCharacter*” happens also to echo the input character, then either a different subroutine should be extracted (named “*echoInputCharacter*”) and called by “*processInputLine*”, or the name of “*pushInputCharacter*” should be changed to “*pushAndEchoInputCharacter*”.

**8.2 Naming Subroutines (Verb and Object)**

* Names of procedures shall consist of a verb and (whenever appropriate) an object, such as “*pushInputCharacter*”. This will make both the action and the object of the action clear. Subroutine names such as “*Subroutine*\_1” are discouraged.
* Always a method name should begin with a simple letter and should follow the camel style of writing. When a function name consists of more than one word, the first letter of each new word must be capitalized. No underscores are added between words in a method name.



* For object-oriented programming,

accessors -for instance or static variables should always be prefixed with "get" or "set".

-for boolean instance variables should usually be prefixed with "is" and "set".

public int getValue()

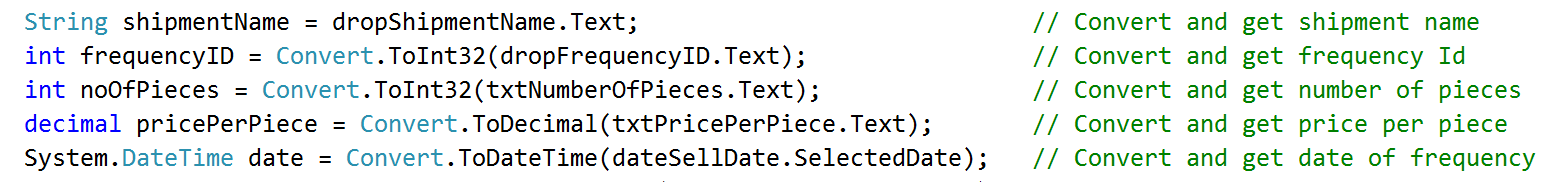
public void setName(String name)

public bool isSelected()

public void setSelected(bool status)

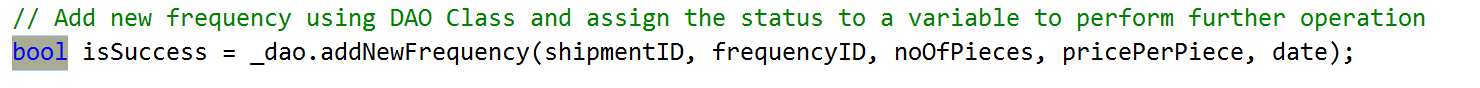
**8.3** **Naming Constants, variables (noun)**

* Names of constants, variables, and functions shall be nouns, with or without modifiers (e.g., “*line*”, “*inputLine*”, “*numInputLines*”), with the exception of Boolean identifiers as noted below. Constants (variables whose values can not be changed during runtime) should be capitalized: MAX\_LINES.
* Variable names must always start with a lowercase letter and follow the "camelCaps" capitalization convention
* Verbosity is generally encouraged.
* Variable name should describe the data that the developer intends to store in them.



**8.4 Naming Boolean identifiers (verb and ((object or adjective))**

* Each name of a Boolean constant, Boolean variable, or Boolean function shall consist of a verb and (whenever appropriate) an object or an adjective. As an example, if this rule were not followed, would the subroutine call “*Black\_King (Checker)*” mean make Checker a black king, or report whether Checker is a black king? This Boolean function would be better named “*Is\_Black\_King*”. Note that the form “*Is\_Black*” illustrates another appropriate construct. A subject for the verb may also be appropriate, as in a Boolean function named “*Stack\_Is\_Updated*”.



**8.5 Abbreviations**

An abbreviation shall only be considered if it saves a considerable number of characters (e.g., “FFT” for “Fast Fourier Transform” is acceptable, but “Snd” for “Send” is not), as long as the language does to restrict identifier lengths severely. Again, the use of these abbreviations shall be consistent. For example, if series of procedures sends various types of messages using the identifiers Send\_Hello\_Msg, Send\_Connect\_Msg, and Send\_Data\_Msg, then the name of a new procedure in the series should not be Send\_Disconnect\_Message.

Example:

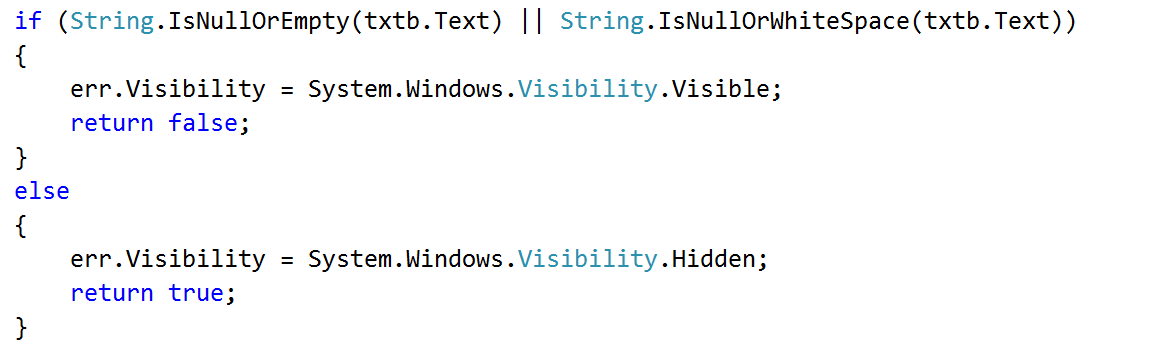
Average: avg

Calculate: calc

**9. Misc. Rules for Coding**

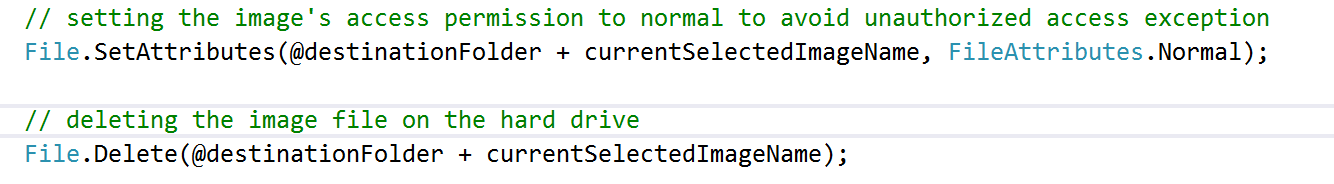
**9.1 Conditionals and comparisons**

* Always test floating-point numbers as <= or >= relational operator, never use exact comparisons ( = = or != ).
* No assumptions shall be made about the value of uninitialized variables, unless the language definition makes a clear statement about this.
* Never use implied processing.... always state clearly a conditional’s intent
* Use built in functions for comparisons when ever possible.



**9.2 Binding time of variables and values**

When data files are accessed in a tree-structured directory environment, the names of the file directories shall not be hardwired in the code; whenever possible, environment variables or some similar mechanism shall be used to provide exact directory names dynamically. The same applies to the names of nodes which are accessed in a network.



**9.3 Go-tos, pointers, and issues of clarity**

Go-tos are not to be “avoided at all costs”. It is, instead, *serpentine code* that needs to be avoided. Simplicity and clarity should override most other design decisions.

Other clarity-based suggestions**:**Use case statements instead of nested ifs, use arrays instead of linked lists, optimize through solid design rather than bit-tuning, get a faster CPU instead of writing assembler, pay for the extra memory, buy code if it’s available.

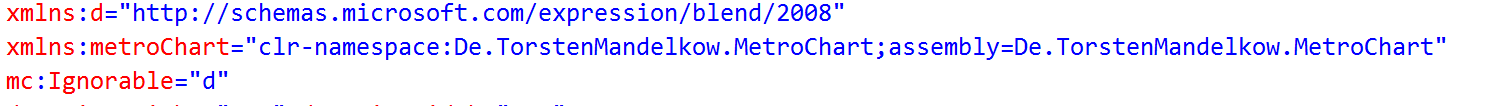
**9.4 Strive to develop clear code**

Engineers should strive to develop code that is both clear, and efficient in its use of CPU time, memory, and other resources. However, when efficiency and clarity conflict, then clarity should take strong precedence over resource stinginess, unless it is proven that using the clear but less efficient method impairs the program critically. Micro-optimizations to small areas of code are especially to be avoided if they impair clarity in any way, since it is generally only the program’s overall algorithms that affect resource utilization significantly.

**9.5 Use libraries when available**

Whenever library routines, graphics packages, compiler/assembler features, or other sorts of utilities are helpful to the program, they should be utilized. The danger of losing the access to the utility if the hardware, the compiler/assembler, or the operating system should change is generally overridden by the savings in software creation time. In those cases, in which there is no significant savings of software creation time, it is preferable to use the standard language features, for portability’s sake.

Example: Metro Charts external library was used for chart generation purposes



**10. Modularization**

**10.1 The information hiding and scope of variables:**

The class variables must be private and other classes must access the variables through getters and setters. The variable scope must be clearly declared and must be protected and must have the minimized scope. public, protected private key words must be used appropriately with variables. Extensive protection should achieve using proper access modifiers.

**10.2 High Cohesion Low coupling:**

High cohesion and low coupling is achieved by separating each class according to their domain. Each class belong to the either View Controller, Data access layer or validation. In each domain all the classes are extensively in the scope of that particular domain. All the domains are totally separated from each other.

**View** – This is what displays to the user (User Interface) designed using XAML

**Controller** – Controls the user inputs and connect with the data access class using

data access objects.

**Data access layer** – Do all the database tasks using LINQ

**Validation**– Do all the validation tasks

**10.3 Clean Interfaces:**

For each domain each class has clean, simple interfaces that provide extensively scoped service that necessary to perform the application tasks easier and safer.

**11. References**

*buffalo.edu. “General Style and Coding Standard for Software Project.” Accessed August 6, 2015.* [*http://www.cse.buffalo.edu/~rapaport/code.documentation.Excerpts.pdf*](http://www.cse.buffalo.edu/~rapaport/code.documentation.Excerpts.pdf)

*Review Board. “Coding Standards”. Accessed August 5, 2015.*

[*https://www.reviewboard.org/docs/codebase/dev/coding-standards/*](https://www.reviewboard.org/docs/codebase/dev/coding-standards/)