# MVS

MVS Coding Standards

1. PURPOSE

To develop reliable, extensible, and easily maintainable applications utilizing common coding standards and best practices. This document is not meant to be exhaustive. For an exhaustive list see the references below.

1. SCOPE
2. This policy applies to all developers.
3. This policy describes the organization’s standards regarding the coding of software for MVS.
4. REFERENCES - The following books should be referenced while developing software for this project:

* Clean Code by Robert C. Martin
* Agile Principles, Patterns and Practices in C# by Robert C. and Micah Martin
* Patterns of Enterprise Application Architecture by Martin Fowler
* The Art of Unit Testing – Roy Osherove

1. DEFINITIONS

* Pascal Casing – First character of a word is upper case, the rest of the characters are lower case. Example : DesignWidth
* Camel Casing – The first character of the first word is lower case. Each additional word begins with an Upper case letter. Example: totalBranchesInLayer
* Hungarian Notation – preface an object or variable with an identifier such as cls for class and str for string. Example: clsBranch, strBranchName
* Tab Spacing – A tab will be defined with a size of four.

1. THIRD PARTY SOFTWARE

* Unit Testing – NUnit
* Logging – Nlog
* Refactoring – ReSharper
* Unit Test Code Coverage – dotCover
* Mocking – MOQ
* MVVM Framework - MVVMLight
* WPF Controls - Telerik
* Database – SQLite
* Entity Framework Design – Devaart
* Ioc Container - Autofac

1. PRINICPLES - The following principles presented by Bob Martin will be strictly followed. The definitions can be found at <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

* Single Responsibility Principle
* Open Close Principle
* Lishkov Substitution Principle
* Dependency Inversion Principle
* Interface Segregation Principle

1. NAMING CONVENTIONS

* Hungarian Notation is not to be used… Today’s IDE’s make it extremely easy to see the type of an object, ui element or primitive.
* Use Pascal Casing for Class names
* Use Pascal Casing for Method Names
* Use Camel Casing for variables
* Use Camel Casing for method parameters
* Prefix member variables with an underscore, “\_”. Do not use the “this” keyword
* Do not use the “\_” underscore for local variable names
* Auto Properties should be used if no code is required for a property.
* Use meaningful, descriptive words to name variables, methods, and classes.
  + Good Use – string businessAddress
  + Bad Use – string busadd or busAdd
* Comments should be used sparingly if at all. Comments are deprecated the moment they are completed. If descriptive words are used, the code will document itself.
* Do not use single character variable names unless they are being used in an iterative loop.
* Do not use variable names that resemble keywords.
* Prefix Boolean variables with “is”. Example – *private bool isOk*.
* Namespaces will follow the standard pattern
  + - ProductName.TopLevelModule.LowerLevelModule
* Only one class per file
* Class file names should match the class name. Example Foo.cs and public class Foo.
* Class names are to use Pascal casing.

1. INDENTATION & SPACING

* Do not use spaces for indentation. Use a tab
* Use a single line to separate methods of a class.
* Curly Braces for a method or block of code should be placed on a new line. No curly braces should be used if there is only one item in a logical block of code.
  + if (likesCheese)
    - {
      * SliceCheese();
      * PlaceOnSandWhich();
    - }
    - else
      * ThrowInTrashCan();
* If the If/Else statement each has only one line of code, use the ternary operator (?).
* Keep member variables private.
* Private methods should be defined right after the first method to call the private method.
* Use #region to group related pieces of code.
* Exceptions to the new line for curly braces will be made for simple properties. In this case, the curly braces can reside on the same line.

1. PROGRAMMING PRACTICES

* See the reference section above!
* A method should do one, and only one thing. If the method does more than one thing, re-factor into multiple methods.
* A class should do one, and only one thing. If the class does more than one thing, re-factor
* Do not pass a null value as an expected parameter to a constructor or method.
* Check parameters of methods and constructors for null and throw an ArgumentNull Exception to the consumer.
* Do not pass a Boolean as a parameter. A Boolean parameter denotes the method does more than one thing.
* Do not use more than three parameters in a method. If more than three parameters are required, refactor to pass an object, list or dictionary. Each parameter exponentially increases the number of unit test required.
* Do not use signature overloads for Constructors. Constructor parameters are to initialize the object. Constructor parameters are to be considered required. If a constructor parameter is not required refactor to use Property injection.
* Properties are to be considered optional.
* Reiterated here, method names should be descriptive. State what it does.
* Methods should be minimalistic. If a method has more than four lines of code, look to refactor.
* Do not use signature overloads for methods. Refactor to use an object, list or dictionary.
* Do not expose member variables. Properties will be used to expose member variables and methods will manipulate member variables.
* No magic values. Use a constant or a configuration.
* Error messages should be descriptive to the user and should help the user to solve the problem. Never give error messages like "Error in Application", "There is an error" etc. Instead, give specific messages like "Failed to update database and a corrective action to take like, “Please make sure the login id and password are correct.”
* All errors are to be logged.
* Avoid very large code files. If a single file has more than 200 lines of code, it should be refactored into two or more classes.
* If a method returns an object or collection, return an empty collection and a default object instead of null, if you have no data to return. This will make it easy for the calling application to just check for the “count” or other valid data rather than doing an additional check for “null”.
* Use names for types instead of Single characters. Example MyDictionary<Key,Value> instead of MyDictionary<K,V>
* Use String.Empty instead of “”.
* All public methods and properties will be made virtual for unit testing.
* Use defensive casting instead of explicit casting.
* If a class is dependent on another class, refactor the class to use constructor or property injection. Create an interface and wire the interface and injection into the Autofac container

1. EXCEPTION HANDLING

* If a custom exception needs to be created, inherit from ApplicationException.
* Always catch specific exceptions.
* Log all exceptions.
* Give the user a user-friendly error message.
* Return the application to a useable state after an exception has occurred.
* Use an empty throw statement to re-throw the original exception in order to preserve the stack-trace.

1. UNIT & INTEGRATION TESTING

* All code paths should be tested in a high-level manner, using integration & unit tests.
* Code test coverage will be as close to 100% as possible.
* Use the nUnit testing framework to create unit and integration tests.
* Use MOQ for a mocking framework.

1. SANCTIONS

Failing to follow this policy could:

1. Delay the completion of sprints and release to customers
2. Make it difficult for effective team collaboration
3. Could affect the future compensation of the developer