Captator's code standard for C# and VB.NET

Published 2011-04-06

Inspired by Microsoft's 'Design Guidelines for Developing Class Libraries'.

For comments, suggestions and bug reports please contact .Henrik Lykke Nielsen (lykke@captator.dk).

If you modify this code standard or use a part of it please attach a comment like this: 'This code standard is based on Captator's code standard, which can be found at www.captator.dk.' or like this: 'This code standard has borrowed rules from Captator's code standard, which can be found at www.captator.dk.'.

Remember: Following a code standard is no excuse for creating bad code. A code standard cannot specify each and every scenario you will encounter so if the code will be better by violating the code standard in specific scenarios please do so at your own discretion. Such violations should however be well founded.

Reading instructions:

Captator.Eifos is the root namespace for Captator's generic framework.

- Rules are marked with a 'disc'.
  - Examples are marked with a 'circle'. An example is attached to the element that it is a subelement of.

Terms

- 'null' is used to denote a null value independent af the specific language. So 'null' denotes 'null' in C# and 'Nothing' in VB.

Capitalization Styles

Pascal case

- The initial letter of the identifier is upper case. The first letter in each concatenated word is upper case.
  - `PascalCase`

Camel Case

- The initial letter of the identifier is lower case. The first letter in each concatenated word is upper case.
  - `camelCase`

Upper Case

- All letters of the identifier is upper case.
  - `SILVERLIGHT`

Identifier Capitalization

- When identifiers consist of multiple words, capitalize each word. Do not separate them with special characters such as underscores ('_').
  - `FileName`
- Compound words which (according to a relevant dictionary) are considered individual terms themselves do not have each component of the compound word capitalized.

    - Keyboard
    - Keyword
    - password

Text Capitalization

Proper Case

- Capitalize each non-trivial word in the text.
    - Close All Documents
    - Connect to Team Foundation Server
    - Recent Files

Normal Case

- Capitalize the text as normal text (capitalize names as well as the first word in each sentence).
    - Show status bar
    - This text is copyrighted by Captator.

Acronyms and Abbreviations

- Only use acronyms that are widely understood (within the context that it is beeing used).
- Acronyms which are themselves considered individual terms are for the sake of capitalization considered words.
- Do not use abbreviations except when the abbreviations themselves are considered individual terms such as Id and Ok.

- ClrType
- Html
- Olap
- Ui
- Xml
- Sql
- Id
- Ok

General Naming Rules

- Use Pascal case for namespace names, type names, enumeration values, methods and public members in general.
- Use camel case for private and protected fields as well as local identifiers (local variables, local constants and parameters).
- Do not use the underscore character '_' except for private fields.
- Do not use Hungarian notation.

Identifier Choice

- Use identifier names that are precise, easy to read and easy to understand (more important than choosing short identifier names).
- Do not use identifiers that are keywords in either VB.NET, C#.NET or C++.NET.
- Use english wording in identifiers except in situations where the domain makes it too difficult to get precise and understandable identifiers using english wording. In these cases the use of a local language for the wording of identifiers is allowed.
- Unambiguousness of identifiers should never be achieved by difference only in casing.
- Use names that describe a type's meaning rather than names that describe the type.
- If a parameter or local variable of a type for which a keyword exists (e.g. string, int, Integer) has no semantic meaning beyond its type, use a generic name. Do not use the type's name as the name of a variable (including parameters).
    - value
    - item
- If the type is important for expressing the semantics, use the CLR type to denote the type. Do not use language specific type names.
    - ParseAsInt32
    - DateTimeAsString

```
◦ VB: Public Sub WriteName(name As String)
◦ C#: public void WriteName(string name)
◦ VB: Public Sub WriteString(s As String)
◦ C#: public void WriteString(string s)
◦ VB: Public Sub WriteInt32(value As Integer)
◦ C#: public void WriteInt32(int value)
```

Assembly

Assembly Naming

- Consider and prefer naming assemblies used as components after their root-namespace.
  - `System.Data.dll`
  - `System.Xml.dll`
  - `Captator.Eifos.dll`
  - `Captator.Eifos.Tests.dll`

Namespace

Namespace Naming

- Name namespaces after the functionality they provide and not after organizational hierarchies.
- Use the company name followed by a product/system name and optionally a list of technology/feature/application names.
  CompanyName.(ProductName|SystemName)[.(TechnologyName|FeatureName|ApplicationName)]*
- Use version-independent product/system names.
- The company name should preferably be globally unique.
- Prefer a stable, recognized technology name.
- Name a namespace that contains types that provide design-time functionality for a base namespace with the .Design suffix.
  CompanyName.TechnologyName[.Feature]*[.Design]
  - `Captator.Windows.Forms.Design`
  - `System.ComponentModel.Design`
  - `System.Workflow.ComponentModel.Design`
- Use Pascal case for each namespace part and separate the namespace parts with periods. Let however your brand names follow their traditional casing.
- Use plural namespace names if it is semantically appropriate.
  - `System.Collections`
- If a custom namespace has the same name as a .NET framework namespace the functionality in the namespaces should match. The functionality in Captator.Eifos.Data matches (is based on/is an extension of) for example the functionality in the System.Data namespace

Namespace Usage

- Use namespaces to organize types based on features and/or functionality.
- Let namespaces be mirrored in the file/folder hierarchy. Files defining types in the 'Data' subnamespace should for example be placed in a subfolder called 'Data'.
- Specify the full namespace types belong to by placing a namespace declaration in all files containing types.
  - `VB: Namespace Captator.Eifos.Data`
  - `C#: namespace Captator.Eifos.Data`
- Do not use a root namespace for the project.
- Do not use 'using' (C#) / 'Import' (VB) for importing namespaces to files. [This is a rule that Captator follows, but it is considered controversial.]
- If 'using' (C#) / 'Import' (VB) is used for importing namespaces, place all such constructs at the top of the file. Sort them alphabetically with .NET framework namespaces first, followed by custom namespaces.

- Specify full type-names (including namespace) when referring to types and static (C#) / shared (VB) members of that type. [This is a rule that Captator follows, but it is considered controversial.] Types placed in the same namespace or a sub namespace hereof can however be referred to by specifying only the relative namespace path.

Type

Type Naming

- Try to avoid type names already used in the .NET framework.

Type Usage

- Place each type in a separate file. An exception to this rule is when defining nested types. Give the file the same name as the type.
- All types must explicitly specify its accessibility level: public (C#) / Public (VB), protected (C#) / Protected (VB), internal (C#) / Friend (VB), protected internal (C#) / Protected Friend (VB) or private (C#) / Private (VB).
- Do not use generic type prefixes except for interface names (upper case 'I') and generic type place holders (upper case 'T').
- Types for which a language has corresponding keywords should be referred to by these keywords as opposed to by the corresponding .NET classes. When calling static members use the .NET classes and not the keywords though.
  - `VB: Dim s As String = ".NET"`
  - `C#: string s = ".NET";`
  - `VB: Dim i As Integer = 42`
  - `C#: int i = 42;`
- When a type name is part of an identifier use the .NET class name and not corresponding language keywords.

Nested Type Usage

- Do not nest types for purposes of grouping types - use namespaces instead.
- Do not use nested types when defining interfaces as not all languages support this.

Generic Type Parameter

Generic Type Parameter Usage

- If a generic type parameter is self-explanatory consider using 'T' as the type placeholder name.
- If a generic type parameter placeholder benefits from having a descriptive name prefix the name with an upper case 'T' as the type placeholder name.
  - `TKey`
  - `TValue`
- If there is multiple generic type parameter placeholders always use descriptive names prefixed with an upper case 'T'.
  - `TKey`
  - `TValue`
- Consider naming a generic type parameter after the constraints placed on the type placeholder.

Class

Class Naming

- Use a noun or an adjective phrase to name a class.
  - `Person`
  - `PersonCollection`
  - `TextBox`
- Use Pascal case.
- Do not use a type prefix (such as C for class) on a class name.

- Consider naming base classes (classes that are supposed to be inherited from) with a postfix of 'Base'.
  - `DataStrategyBase`
- Consider postfixing subclasses with the name of the base class (do not however include a 'Base' postfix if the base class has that).
  - `SqlClientDataStrategy`
  - `FileStream`
  - `InvalidOperationException`
- Partial classes are placed in separate files named based on the type name. Postfixes to the basepart of the filename is allowed for partial class file names after the following scheme: BaseFileName.Postfix.Extension.
  - `Form1.cs and Form1.Designer.cs`
  - `ReflectionUtil.cs and ReflectionUtil.SilverLight.cs`
- If an interface has a standard/default implementation name the interface and the class the same except for the 'I' prefix of the interface.
  - `Collection and ICollection`
  - `AsyncResult and IAsyncResult`

Util Class

Util Class Naming

- Static utility classes have their name suffixed with 'Util'.
  - `DataUtil`

Util Class Usage

- Util classes are classes with no instance methods. In C# these are declared as static classes. In VB they are declared as NotInheritable classes with a private parameterless constructor and no public constructors.
- The VB type 'module' is not used as its members have global scope without prefixing with the type.

Collection

Collection Class Naming

- Add 'Collection' as a suffix to classes that implements System.Collections.IEnumerable, System.Collections.ICollection, System.Collections.IEnumerable<T> or System.Collections.ICollection<T>.
  - `DataRowCollection`
- Add 'List' as a suffix to classes that implements System.Collections.IList or System.Collections.IList<T>. This rule overrides the more general collection naming rule mentioned above.
- Add 'Dictionary' as a suffix to classes that implements System.Collections.IDictionary or System.Collections.Generic.IDictionary<TKey, TValue>. This rule overrides the more general collection naming rule mentioned above.

Collection Class Usage

- Let custom list classes (such as those implementing System.Collections.IList<T>) be zero-based.

Collection Instance Naming

- Name instances of collections in the plural form of the name of the collection class.
  - `Singular: Employee, plural: Employees`
  - `Singular: Category, plural: Categories`

Member

Member Usage

- All class members must explicitly specify its accessibility level: public (C#) / Public (VB), protected (C#) / Protected (VB), internal (C#) / Friend (VB), protected internal (C#) / Protected Friend (VB) or private (C#) / Private (VB).
- Never hide existing members implicitly. In those extremely rare cases where hiding is needed use explicit new (C#) / Shadows (VB).

Method

Method Naming

- Use verbs or verb phrases to name methods.
- Use Pascal case.
- Use a name that signals the semantic of the method and not the specific implementation.

Method Overloading Usage

- Only use method overloading to provide different methods that do semantically (almost) the same thing.
- Do not overload methods solely based on 'by reference' / 'by value' / 'out' parameter differences.
- Implement the overloaded variations of a method in terms of the most complete overload.
- If a reference type parameter is declared as optional let a value of null result in the optional parameters default value being used.
- Use the same semantics for the shared set of parameters for each of the overloaded variations of a method.
- If virtuality is needed then only the method(s) in the group that has the most parameters should be virtual. Methods with default parameter values (either using optional arguments or using overloaded methods) should not be virtual.
- Instead of creating overloaded methods consider to define parameters as optional.

Constructor Usage

- Name constructor parameters identical to properties that they are used to initialize.
- Do not call virtual members inside its constructor as instance members has not yet been initialized.

Parameter

Parameter Naming

- Use camel case.
- Use names based on a parameter's semantics rather than names based on it's type.
- If however a parameter has no semantic meaning beyond it's type, use a generic name. Do not use the type's name as the name of a parameter.
- Use a consistent ordering and naming pattern for method parameters.
- Strive to reuse parameter names when parameters in different methods represents the same input.
- When parameters in overloaded methods represents the same input strive to reuse both parameter names and positions.
- Use the same parameter names in an overriding method as in the virtual/abstract method.
- Use the same parameter names in an implementing method as in the interface definition of a method.
- Use the same parameter names when implementing an event sink as in the event handler delegate type.

- When naming optional parameters (including parameters that are not present in all overloaded versions of a method) use names that indicate a change from the default state of the optional parameters.
  - VB:
    ```
    ' GetMethod1: ignoreCase = false.
    Function Type.GetMethod1(name As String) As MethodInfo
    Function Type.GetMethod2(name As String, ignoreCase As Boolean) As
    MethodInfo
    ```
  - C#:
    ```
    // GetMethod1: ignoreCase = false.
    MethodInfo Type.GetMethod1(string name);
    MethodInfo Type.GetMethod2(string name, bool ignoreCase);
    ```

Parameter Usage

- Define parameters specifying the most general type that offers the needed functionality.
- Place 'out' parameters after 'by value' and 'by reference' parameters even if conflicts with the rule on parameter ordering for overloaded methods.

Field

Field Naming

- Use camelCase prefixed with an underscore '_' for private fields.
- Use a noun phrase to name a field.

Field Usage

- Do not use instance fields that are public, protected or internal. Use properties instead.
- Use the const (C#) / Const (VB) keyword to declare constant fields that will never ever change.

Property

Property Naming

- Use a noun or an adjective phrase to name properties.
  - `Name`
  - `Parent`
- Do not create a property matching a Get-method. For example do not create both a PersonId-property and a GetPersonId-method.
- Name properties of type System.Boolean with an affirmative phrasing and where appropriate consider prefixing properties of type System.Boolean with 'Is', 'Can', or 'Has'.
  - `CanSave`
  - `IsMouseOver`
  - `HasItems`
- An enumeration typed property that returns a single value of the enumeration type can be given the same name as the enumeration type.
  - `VB: Public Property Color As System.Drawing.Color`
  - `C#: public System.Drawing.Color Color { get; set; }`

Property Usage

- When defining a write-only property consider whether it ought to be a method instead.
- Do not assume that properties must be set in a specific order.
- When implementing a property setter preserve the value of the property before you change it so that data is not lost if the set accessor throws an exception.
- Exceptions should not be thrown in property getters.

Event

Event Naming

- Use Pascal case.
- Name an event with a verb or a verb phrase.
- Use an EventHandler suffix on event handler names (and only on those).
  - `System.Windows.Forms.MouseEventHandler`

- EventHandlers must have two parameters named 'sender' and 'e'.
- The 'sender' parameter must be a reference to the object that raised the event and it must always be of type object.
- The 'e' parameter represents the state associated with the event. The type of the 'e' parameter must either be System.EventArgs or a subclass thereof.
- Name event argument classes by adding the suffix 'EventArgs'.
- Use present tense to denote a pre event and past tense to denote a post event.
  - `Closing`
  - `Closed`
- Do not prefix or suffix event names. E.g.: Do not use OnClose, use Close instead.

Event Usage

- Eventhandlers should be declared as 'void' (C#) / 'Sub' (VB).
- Use System.EventHandler<T> (C#) / System.EventHandler(Of T) (VB) as the evanthandler type.
- If an event can provide useful data use a subclass of System.EventArgs to carry such data.
- Use System.EventArgs.Empty when no event data is available. The System.EventArgs argument 'e' should never be null.
- The 'sender' argument should never be null for instance events.
- In C# do a null check before invoking event instances. VB does this automatically when raising events.
  - `C#: if (this.Click != null) {    this.Click(sender, e); }`
- Consider using Captator.Eifos.DelegateUtil.CallInvocationListSafely or Captator.Eifos.DelegateUtil.FireEvent when invoking delegates. [Captator.Eifos specific rule.]

Operator Overloading

Operator Overloading Usage

- Only implement operator overloading when operators are well defined for the type in question.
- Implement operators symetrically. E.g.: If the equality operator is implemented also implement the inequality operator and if the less than operator is implemented also implement the greater than operator.
- Consider providing standard methods with the same functionality as the implemented operators. E.g. the '+' operator and the 'Add' method.
- Implicit casts should never throw exceptions.
- Do not modify the operands.

Structure

Structure Naming

- Use a noun or an adjective phrase to name a Struct (C#) / Structure (VB) .
- Use Pascal case.

Structure Usage

- If a data type represents a single value (something that is considered a primitive value in itself) consider it to be implemented using a struct (C#) / Structure (VB).

Enumeration

Enumeration Type Naming

- Use Pascal case for both enumeration type names and value names.
- Do not add 'Enum' as a suffix to enumeration type names.
- Do not include the enumeration type name in the enumeration value names.

- Do not prefix enumeration value names.
- Use a noun or an adjective phrase to name an enumeration.
- Do not add 'Flags' as a suffix to enumeration type names.
- Use a singular name for simple enumeration types, but use a plural name for flag enumerations.
  - VB:
```
Public Enum DayOfWeek
   Sunday
   Monday
   Tuesday
   Wednesday
   Thursday
   Saturday
   Friday
End Enum
```
  - C#:
```
public enum DayOfWeek
{
   Sunday,
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday
}
```

Enumeration Usage

- Do not define numeric values for the enumeration values except when the specific numeric values are of importance e.g. when the values are persisted or if the values represents flags.
- Be wary of using enumerations when values might be added, changed, reordered or removed in the future.
- Specify an enumeration value of zero. Name this 'None', 'Unspecified' or similar.
- Prefer System.Int32 as the underlying type of an enumeration unless the enumeration represents flags and there is (or will be) need for more than 32 flags or if another type will ease interoperability scenarios.
- Use an enumeration instead of a collection of correlated integer based constants.
- Consider providing explicit enumeration values for commonly used combinations of flags (summing up the values of those flags). Define such combinations using enumeration value identifiers.
- Add the System.FlagsAttribute to a bit field enumeration type.
  - VB:
```
<Flags> _
Public Enum FileAttribute
   Archive = &H20
   Directory = &H10
   Hidden = 2
   Normal = 0
   [ReadOnly] = 1
   System = 4
   Volume = 8
End Enum
```
  - C#:
```
[Flags]
public enum FileAttribute
{
   Archive = 0x20,
   Directory = 0x10,
   Hidden = 2,
   Normal = 0,
   ReadOnly = 1,
   System = 4,
```

```
        Volume = 8
    }
```

Attribute

Attribute Naming

- Add the 'Attribute' suffix to attribute class names (and only to those).
  - `ObsoleteAttribute`

Attribute Usage

- Specify AttributeUsage on your attributes.
- Implement read-only properties for each optional argument.

Interface

Interface Naming

- Use a noun or an adjective phrase to name an interface.
  - `ICollection`
  - `ICollectionView`
  - `ISerializable`
- Use Pascal case.
- Prefix interface names with the letter 'I'.

Interface Implementation Usage

Instance Access to Interface Method Implementation

- If a method being implemented on an interface is an inherent part of the semantics of the class implementing the interface it is valid to publish the method on the class.
- If two different interface methods (on the same or on different interfaces) have the same semantics it is valid to let them share implementation.
  - VB:

```
Public Sub Close() Implements IDisposable.Dispose, ICleanupable.Cleanup
    '...
End Sub
```

  - C#:

```
void IDisposable.Dispose()
{
    Close();
}

void ICleanupable.Cleanup()
{
    Close();
}

public void Close()
{
    //...
}
```

- Prefer implementing interfaces explicitly. Let the implementation default to being private. [This is a rule that Captator follows, but it is considered controversial.]
  - VB:

```
Private Sub Dispose() Implements IDisposable.Dispose
    '...
End Sub
```

  - C#:

```
void IDisposable.Dispose()
{
    // ...
}
```

- If an explicitly implemented member is meant to be overridden in a subclass provide a protected virtual member implementing the same functionality.

IDisposable

- Always dispose disposable objects either explicitly using the 'IDisposable.Dispose' method (or another method delegating to the 'IDisposable.Dispose' method e.g. the common Close method) or implicitly by using the 'using' (C#) / Using (VB) language construct.
- Consider using the 'using' (C#) / Using (VB) language construct instead of explicitly disposing.
- Implement the 'Dispose' design pattern for types that depends on external resources.

Exception

Exception Naming

- Add the 'Exception' suffix to exception class names (and only to those).
- Do not name exception instances 'e' as it will often clash with the name of an EventArgs-parameter. 'ex' is often used in catch (C#) / Catch (VB) constructs.

Exception Usage

- Avoid returning error codes from methods. Throw exceptions instead. An exception to this is when the name of a method explicitly states that error handling is part of its semantics.
  - C#:
```
bool result;

if (System.Boolean.TryParse("true", out result))
{
   // Do something...
}
```
- A static constructor should never throw exceptions.
- Only throw exceptions in exceptional cases. Do not use exception handling for flow control.
- Never inherit from System.ApplicationException.
- Use existing exception types if appropriate. Where appropriate prefer inheriting specific exception types instead of creating new ones inheriting System.Exception.
- When rethrowing exceptions as a default rule prefer 'throw' without parameters to rethrowing the caught exception explicitly as that resets the exception call stack.
  - VB: `Throw`
  - C#: `throw;`
- Code defensively: Prefer checking validity of state before executing an action to handling exceptions after executing the action.
- An empty catch block must always be documented.
  - VB:
```
Try
   connInfo.RollbackTransaction()
Catch
   ' Rollback exceptions are ignored as the original exception is the
interesting one.
End Try
```
  - C#:
```
try
{
   connInfo.RollbackTransaction();
}
catch
{
   // Rollback exceptions are ignored as the original exception is the
interesting one.
}
```

Delegate

Delegate Type Naming

- Consider naming delegate types that are not event handler delegate types with the suffix 'Callback'.
- Do not add 'Delegate' as a suffix to delegate types.

Delegate Usage

- Do a null check before invoking delegate instances.
- In C# the code block of multiline anonymous methods and lambda expressions should be considered standard code blocks with respect to indendation.
  - ```
    C#: System.Action mainAction = delegate
    {
        // ...
        // ...
    };

    mainAction();
    ```

Array

Array Usage

- Return an empty array instead of a null reference.

XML Comment

XML Comment Usage

- All XML comments should be spellcheked and with correct punctuation.
- Document public parts of APIs using XML comments and not standard comments.
- Specify references instead of plain text.
  - ```
    /// <param name="s">The <see cref="System.String"/> to test.</param>
    ```

Comment

Comment Usage

- All comments should be in US English.
- All comments should be spellcheked and with correct punctuation.
- Prefer writing clear and well structured code to writing comments.
- Prefer creating well named methods instead of commenting blocks of code.

General Layout Rules

- Default to using Visual Studio's automatic code layout.
- Use empty lines to group code.
  - Insert an empty line between methods and properties with explicit get and/or set implementations.
  - Insert an empty line between logical groups of instance variables and/or automatic properties.
  - Insert an empty line between logical groups of statements.
- In C# place '{' (block start) and '}' (block end) on separate lines.

Code Constructs

Implicitly Typed Local Variables

- Only use implicitly typed local variables (declared using 'var' (C#) / 'Dim' (VB) without specifying an explicit type) in one of the following two scenarios:
- 1) When the type is explicitly specified immediately on the right hand side of the assignment operator (the '=' operator) either as an explicit type cast of a datatype for which no keyword exist or by creating a new instance using the new (C#) / New (VB) operator. In this case prefer using implicitly typed local variables to specifying the type explicitly.
  - ```
    C#: var p = (Person) obj;
    ```
  - ```
    C#: var p = new Person();
    ```
  - ```
    VB: Dim p = New Person()
    ```
  - ```
    VB: Dim p As New Person()
    ```
- 2) When assigning instances of anonymous types or collections of anonymous types (when using e.g. LINQ).

Type Casting

- To ensure that type cast exceptions occurs at the earliest possible time prefer using standard typecasts to using the 'as' (C# / 'TryCast' (VB) operator when a type check is needed.
  An exception to this rule is if no type check is necessary and _no_ error can follow from the use of the 'as' (C# / 'TryCast' (VB) operator returning null. E.g. if the casted value is parsed to a function that accepts null as a valid value.
  Note however that the use of the 'as' (C# / 'TryCast' (VB) operator saves a type check compared to explicitly checking the type before doing a standard type cast so the 'as' (C# / 'TryCast' (VB) operator will run slightly faster. If such performance is imperative feel free to use the 'as' (C# / 'TryCast' (VB) operator. That should however not be necessary in most scenarios.

Comparing Data

- When comparing strings case insensitively use the Equals or the Compare method specifying the StringComparison-argument instead of converting the strings to a common casing using e.g. the ToUpper/ToLower methods.
  Note that both the static Equals and the Compare-methods as well as the Equals instance method have several overloads.
  - VB:
    ```
    If System.String.Equals(".NET", ".net",
    System.StringComparison.InvariantCultureIgnoreCase) Then
        ' Do something
    End If
    ```
  - C#:
    ```
    if (System.String.Equals(".NET", ".net",
    System.StringComparison.InvariantCultureIgnoreCase))
    {
       // Do something
    }
    ```

Finalizer

- Do not implement empty destructors (C#) / finalizers (VB).

- As a general rule numerical literals should only be specified inside constant declarations.
  Exceptions to this rule are:
  Trivial values such as '0', '1', '2' and '-1'.
  Mathematical formulas.
  When the semantics is immediately clear from the context.
  - VB: `Dim milliSeconds As Integer = seconds * 1000`
  - C#: `int milliSeconds = seconds * 1000;`
- If a constant value is dependent on another constant value make that dependency explicit.
  - VB:
    ```
    Const a As Integer = 17
    Const b As Integer = a + 42
    ```
  - C#:
    ```
    const int a = 17;
    const int b = a +42;
    ```
- Always use block constructs where applicable. Do not use single statement blocks.
  - VB:
    ```
    If 2 + 2 = 4 Then
        ' Do something
    End If
    ```
  - C#:
    ```
    if (2 + 2 == 4)
    {
       // Do something
    }
    ```
- Only place one statement on each line. An exception to this rule is else-if statements.
  - VB:
    ```
    If 2 + 2 = 4 Then
        ' Do this
    ElseIf pigsCanFly Then
    ```

```
    'Do that
Else
    ' And now for something completely different.
End If
```

◦ C#:

```
if (2 + 2 == 4)
{
    // Do this
}
else if (pigsCanFly)
{
    // Do that
}
else
{
    // And now for something completely different.
}
```

- In VB prefer 'Do-Loop' constructs to 'While-End While' constructs.
  ◦ VB:

```
Do While pigsCanFly
    ' Do something
Loop
```

- Infinite loops should be implemented as a while loop (C#) / Do-While loop (VB).
  ◦ VB:

```
Do While True
    ' Do something
Loop
```

  ◦ C#:

```
while (true)
{
    // Do something
}
```

- Do not explicitly compare boolean expressions with true and false literals.
  ◦ VB:

```
If pigsCanFly Then
    ' Do something
End If
```

  ◦ C#:

```
if (pigsCanFly)
{
    // Do something
}
```

- Use local variables liberally to split up complex method calls and expressions.
- Only use the C# '++' and '--' operators as standalone statements. Never use them as part of expressions.
  ◦ C#: `i++; i--;`
- Only use goto statements when it will make the code distinctly simpler.

Resource Naming

- Use Pascal case for resource keys.
- Provide descriptive names for the resource keys.
- Use the '.' separator to create a name-hierarchy similar to a namespace-hierarchy.
- The name for exception type message resources keys should be ExceptionName.ExceptionMessageName.

Conditional Compilation Symbol Naming

- Use upper case only.
  ◦ DEBUG
  ◦ SILVERLIGHT

File Naming

- Use Pascal case.

User Interface Design

Texts in User Interfaces

Texts in User Interfaces - English

- Texts in command controls (menus and buttons) are proper case.
  - `Save All`
- Texts in tab controls and in header columns of grid and listview controls are proper case.
  - `Customer Details`
  - `First Name`
- Texts in labels and similar controls (e.g. checkboxes, radiobuttons and groupboxes) are normal case.
  - `First name`
- If the activation of a command control (menus and buttons) does not in itself execute functionality that changes state, but only opens a (new) dialog, then suffix the text of the command control with an elipsis (three full stops '...').
  - `Page Setup...`
  - `Print...`

User Interface Control

User Interface Control Usage

- If a string that is considered data (as opposed to a label with a simple and fixed content) is to be presented as a read only text use a ReadOnly TextBox instead of a label, as a TextBox supports scrolling and copying of the text.

User Interface Control Naming

- Either prefix or suffix all user interface controls with the control type name. Select only one of these two approaches. [Captator prefixes with the control type name.]
  - `ButtonOk`
  - `OkButton`
- Controls that can contain multiple items are named using the singular form (similar to database table names) of the item.
  - `ListBoxPerson`
  - `ComboBoxType`
  - `GridUser`
- If a label is used as header control to e.g. a TextBox consider appending 'Header' as a suffix.
  - `LabelNameHeader`

Multi Threading

- Consider naming threads so they are easier to identify when debugging.

Database Access

Data Access Layer

Data Access Layer Naming

- Dal classes are classes that contains data access logic. Their names are suffixed with 'Dal'. If the class is a static class do not suffix with the usual 'Util' suffix.
- Dal classes are placed in a 'DataServices' namespace.

Data Code

Database Connections

- Support optimal usage of connection pooling by always using the exact same connectionstring for specifying the same database connection - use exactly _one_ global constant, variable, property or function to hold/return the connectionstring to use.

- All domain specific SQL placed in CLR-code must be placed in Dal classes.
- When using System.Data for data access all data values in SQL statements must be represented by System.Data.Common.DbParameter objects.
- Keywords in SQL statements are written as upper case only.

- Prefer placing SQL statement strings in temporary variables before sending them as parameters to database functions. This makes it is easier to debug database code (especially when the SQL statement string is a complex dynamically created string).

Visual Studio Options

- Tools|Code Snippets Manager: Add the code snippets directory (from the source code versioning system) to each language's code snippet location list.
- The text editor must insert spaces ('Tools|Options|Text Editor|All Languages|Tabs|Insert spaces' must be selected). [This is a rule that Captator follows but many developers choose to insert tabs instead.]
- The text editor must have a tab and indent size of 2 spaces ('Tools|Options|Text Editor|All Languages|Tabs' must have both textboxes filled with the value '2').

Visual Studio Project Properties

- Enable 'Check for arithmetic overflow/underflow' (C#). Use the checked keyword explicitly on code blocks and expressions where overflow/underflow checks are unwanted.
- Only enable 'Remove integer overflow checks' (VB) when overflow/underflow checks are unwanted.
- Place assembly attributes in the 'AssemblyInfo.cs' file.
- All C# projects must have compiler warning CS0108 ("'member1' hides inherited member 'member2'. Use the new keyword if hiding was intended.") set in 'Treat warnings as errors' in the 'Build' properties.

Database Design

Database Objects

Database Object Naming

- Database names, Table names, View names and Field names are Pascal Case.
  - `Person`
- Primary key fields are named 'Id'.
- Association tables are named by concatenating the names of all the tables that are associated.
  - `CompanyEmployee`
- Table names are named using the singular form of the records in the table.
  - `Person`

Database Object Usage

- All tables (including association tables) must have a primary key field.
- Primary keys must always be artificial (the key values must not have any domain semantics).
- Primary key fields are specified as a single field; either an integer (System.Int32) or a GUID (System.Guid).
- When selecting the type type of primary key fields prefer GUIDs because of flexibility in e.g. synchronization scenarios and prefer integers where the size and/or performance is an issue.

Database Object Usage

- Only use code for implementing data access. Do not drop visual data access components into design views (forms, windows etc.).