# C# Coding Standards

The C# coding standards described below must be followed for code that is contributed back to Ed-Fi core repositories for the Ed-Fi ODS/API. We also highly recommend that you follow it for all Ed-Fi implementations.

Note: These guidelines are currently in draft form, and we welcome feedback and input from the Ed-Fi community. They will also be expanded to address the Ed-Fi Dashboards.

## Naming Conventions

1. Use Pascal casing for type, method names and constants:

```
public class SomeClass
{
    private const int DefaultSize = 100;

    public void SomeMethod()
    {
         …
    }
}
```

2. Use camel casing for local variable names and method arguments:

```
public void SomeMethod(int someNumber)
{
    int number;
    …
}
```

3. Apply camel casing and Pascal casing as follows:
    a. Make sure the changes in casing match individual words and are not actually splitting compound words.

```
// Good
bool grasshopper = false;
string grandmother = "Nellie";

// Bad
bool grassHopper = false;
string grandMother = "Nellie";
```

    b. If a parameter or variable name matches a C# language keyword, either rename the variable or use the @ symbol to prevent compilation errors. (Do not modify the casing to resolve the compilation error and thereby effectively split a compound word in two.)

```
// Bad
private string StripNamespacePrefixFromValue(string value, string nameSpace)
{
    if (value.BeginsWith(nameSpace))
    {
        return value.Substring(nameSpace.Length);
    }
    return value;
}

// Better
private string StripNamespacePrefixFromValue(string value, string @namespace)
{
    if (value.BeginsWith(@namespace))
    {
        return value.Substring(@namespace.Length);
    }

    return value;
}

// Best
private string StripNamespacePrefixFromValue(string value, string namespacePrefix)
{
    if (value.BeginsWith(namespacePrefix))
    {
        return value.Substring(namespacePrefix.Length);
    }

    return value;
}
```

4. Prefix interfaces with `I`.

```
public interface ISomethingProvider
{
    …
}
```

5. Suffix interface implementations with the non-prefixed interface name.

```
public class ThisSomethingProvider : ISomethingProvider
{
    …
}
```

6. Prefix all fields with an underscore (_). If a file happens to differ in style from these guidelines, the existing style in that file takes precedence.

```
public class SomeClass
{
    private ISomethingProvider _thisSomethingProvider;
}
```

7. Name methods using a verb or verb-object pair (unless implementing a fluent API).

```
public decimal CalculateTax(decimal amount)
{
    …
}
```

8. What's in a name? Everything. Use clear and descriptive names for classes, methods, fields and variables to reduce maintenance costs. Make sure that after refactoring code that the names are still appropriate.
   For example:

**A Method Name That Does Not Reflect Its Behavior**

```
private static bool IsNotDuplicate(Dictionary<Tuple<string, string>, int> idsByValue, Tuple<string,
string> key, StringBuilder duplicatesMessageBuilder)
{
    if (idsByValue.ContainsKey(key))
    {
        duplicatesMessageBuilder.AppendFormat("\tDescriptor Type: {0}, Code Value: {1}\r\n", key.Item1,
key.Item2);
        return false;
    }

    return true;
}
```

a. The method name includes the text "Not" and returns a boolean value. This could create scenarios where callers are performing double negatives, which are hard to read (e.g. `if (!IsNotDuplicate(…) { )`
b. The behavior of the method actually appends to the provided `StringBuilder`, but this is not reflected in the method name at all and is only discoverable by a developer inspecting the code.
c. Since the main behavioral aspect of this method is to modify the `StringBuilder`, it would make more sense to pass it first in the list of arguments.
d. Without refactoring the code (which is probably the better option here), a more descriptive method name and signature *might* look like this (note the flip in boolean semantics):

**A New Name that Captures Actual Behavior**

```
private static bool TryAppendDuplicateValueMessage(StringBuilder duplicatesMessageBuilder,
Tuple<string, string> key, Dictionary<Tuple<string, string>, int> idsByValue)
{
    …
}
```

9. You can use single-character or mnemonic variable names in the following scenarios:
    a. In `for` loops where it is a common convention to use variables like `i`, `j`, and `k`.
    b. In LINQ expressions where it is a common convention to use variables like `x` or a mnemonic for the item being represented (e.g. `ssa` for an object of type `StudentSchoolAssociation`).
10. Always use `ex` for exception handling variables.

```
try
{
    int x = 0;
    int y = 5 / x;
}
catch (DivideByZeroException ex)
{
        _logger.Warn("Somebody was dividing by zero.");
}
catch (Exception ex)
{
        _logger.Error(ex);
    throw;
}
```

11. Name dictionaries using a name format of {*ValueName*}By{*KeyName*}.
    a. Precisely describe the keys and values (e.g. `SchoolNameById` would indicate that you can obtain a school's *name* by its identifier).
    b. If each entry's value is a *single item*, the name should be *singularized* (e.g. `StudentById`).
    c. If each entry's value is a *collection*, the name should be *pluralized* (e.g. `StudentsBySectionId`).
12. Do not abbreviate terms as this leads to usage inconsistency in the code and other application artifacts. Specifically, do not abbreviate "EducationOrganization" as “EdOrg” or “LocalEducationAgency” as “Lea”.
13. Name abstract classes using a suffix of `Base` (e.g. `EdFiControllerBase`).
14. Use the following guidelines when defining generic types:
    a. For types with a single generic type, prefer the use of "T".

```
public interface IList<T>
```

b. For types with multiple generic types, use a capital "T" followed by an optional secondary name for additional clarity. In all cases, start the type with a capital letter.

```
public interface IService<TRequest, TResponse>
```

15. Name custom attribute and exception classes using suffixes of `Attribute` and `Exception`, respectively.
16. Name enumerations in the *singular* (e.g. `FileMode`) unless the enumeration is representing a bit flag value (e.g. `FileAttributes`) in which case the `[Flags]` attribute should also be applied.

# Coding Style

1. Use 4 spaces for indentation. Do not use tabs.
2. Use the idomatic C# types instead of .NET Framework types.

```
// Good
string a;
object b;
int c;
double d;

// Bad
String a;
Object b;
Int32 c;
Double d;
```

3. Use explicit types for value-typed variables.

```
// Good
int age = 7;
string greeting = "Hello world.";
double elephantWeight = 123.45;

// Bad
var age = 7;
var greeting = "Hello world.";
var elephantWeight = 123.45;
```

4. Use `var` for everything else (including Linq query results regardless of the type), remembering to name the variable in a way that makes the type obvious.
5. Prefer C#'s SQL-like syntax for LINQ queries instead of method chaining.

```
var students = _studentsService.GetAll();

// Preferred
var firstBobsLastName =
    (from s in students
    where s.FirstName == "Bob"
    select s.LastName)
    .FirstOrDefault();

// Less preferred
var firstBobsLastName = students
    .Where(x => x.FirstName == "Bob")
    .Select(x => x.LastName)
    .FirstOrDefault();

// Bad
var firstBobsLastName = students.Where(x => x.FirstName == "Bob").Select(x => x.LastName).
FirstOrDefault();
```

6. Put auto-properties on a single line.

```
// Good
public int Age { get; set; }

// Bad
public int Age
{
     get;
     set;
}
```

7. Bring `base` or `this` constructors onto a separate line (indented).

```
public NotFoundException(string message, string typeName, string identifier)
    : base(message)
{
    TypeName = typeName;
    Identifier = identifier;
}
```

8. Constructors with no bodies can be shortened to one line.

```
public NotFoundException() {}

public NotFoundException(string message)
    : base(message) {}
```

9. Bring constraints for generic types onto separate lines (indented).

```
public abstract class EdFiControllerBase<TResource, TEntityInterface, TAggregateRoot, TGetByKeyRequest,
TPutRequest, TPostRequest, TDeleteRequest> : ApiController
    where TResource : IHasIdentifier, IHasETag, new()
    where TEntityInterface : class
    where TAggregateRoot : class, IHasIdentifier, new()
    where TPutRequest : TResource
    where TPostRequest : TResource
    where TDeleteRequest : IHasIdentifier
{
    …
}
```

10. Always use explicit scope.

```
// Good
public class Something
{
    private const int MaximumAge = 100;

    public int Age { get; set; }

    public void SayHello()
    {
        Console.WriteLine("Hello.");
    }
}

// Bad
class Something
{
    const int MaximumAge = 100;

    int Age { get; set; }

    void SayHello()
    {
        Console.WriteLine("Hello.");
    }
}
```

11. Use `string.Empty` rather than empty quotes (`""`) for empty strings. It clarifies true intent.
12. Use comments liberally to provide *useful/ context* for another developer to be able to follow the intent/logic of the code.
    a. Comments should include a space between the forward slashes and the text.

```
// This is the preferred style

//this is not preferred
```

    b. Comments should appear on their own lines – they should not appear at the end of a line of code.

**Bad Commenting Style**

```
foreach (var student in students.Where(x => x.Name == "Bob")) // What about Bob?
{
    …
}
```

    c. Comments should have a blank line before and, at the very least, after the segment of code to which the comment applies.

```
var scores = _injectedService.GetScores();

// Calculate the average
int count = scores.Count;
double sum = scores.Sum();
double average = sum / count;

foreach (var score in scores)
{
        Console.WriteLine("Doing something else.")
}
```

13. When leaving "TODO" comments, include your initials so other developers will know who to talk to if they have questions.

```
// TODO: GKM - Will need to do something with this eventually.
```

14. Use the following guidelines when using braces ({ and }):

a. Use Allman style braces where each brace begins on a new line.

```
if (a == b)
{
    // More logic here
}
```

b. Do not use single-line `if` statements.

```
// Good
if (source == null)
    throw new ArgumentException();

// Bad
if (source == null) throw new ArgumentNullException();
```

c. Braces may be eliminated, but only if the nested blocks are single line statements.

```
// Good
if (_items == null)
    _items = source.ToList();

// Good
if (itemResults.Any())
{
    foreach (var itemResult in itemResults)
    {
        if (itemResult is CompositeValidationResult)
            compositeResults.AddResult(itemResult);
        else
            compositeResults.AddResult(new ValidationResult(…));
    }
}

// Bad
if (itemResults.Any())
    foreach (var itemResult in itemResults)
        if (itemResult is CompositeValidationResult)
            compositeResults.AddResult(itemResult);
        else
            compositeResults.AddResult(new ValidationResult(…));
```

d. Don’t mix and match bracing styles at the same level of a code block.

```
// Bad
if (source == null)
    throw new Exception(“Something bad happened.”);
else
{
    // More logic here
    …
}
```

e. Braces are always acceptable, but reducing their use is preferred because they tend to add “noise” to the code.

15. When building multi-line conditional statements, put the conditional operator at the *beginning* of each line.

```
// Good
public bool IsEmpty()
{
    return
        !(EducationOrganizationId.HasValue || StateEducationAgencyId.HasValue
          || EducationServiceCenterId.HasValue || LocalEducationAgencyId.HasValue
          || SchoolId.HasValue || EducationOrganizationNetworkId.HasValue)
          && string.IsNullOrWhiteSpace(StaffUniqueId)
          && string.IsNullOrWhiteSpace(StudentUniqueId)
          && string.IsNullOrWhiteSpace(ParentUniqueId)
          && string.IsNullOrWhiteSpace(Namespace)
          && string.IsNullOrWhiteSpace(AssessmentFamilyTitle)
          && string.IsNullOrWhiteSpace(AssessmentTitle);
}

// Bad
public bool IsEmpty()
{
    return
        !(EducationOrganizationId.HasValue || StateEducationAgencyId.HasValue
          || EducationServiceCenterId.HasValue || LocalEducationAgencyId.HasValue
          || SchoolId.HasValue || EducationOrganizationNetworkId.HasValue) &&
          string.IsNullOrWhiteSpace(StaffUniqueId) &&
          string.IsNullOrWhiteSpace(StudentUniqueId) &&
          string.IsNullOrWhiteSpace(ParentUniqueId) &&
          string.IsNullOrWhiteSpace(Namespace) &&
          string.IsNullOrWhiteSpace(AssessmentFamilyTitle) &&
          string.IsNullOrWhiteSpace(AssessmentTitle);
}
```

16. Avoid more than one empty line at any time. For example, do not have two blank lines between statements or members of a type.

```
if (a == b)
    DoSomething();
else
    DoSomethingElse();


Console.WriteLine("There's too many blank lines above this one.");
```

17. Do not add blank lines between sets of of closing braces.

```
if (a == b)
{
    if (c == d)
    {
        if (e == f)
        {
            DoSomething();
        }
    }

} // Blank line above this brace should be removed.
```

18. Include blank lines after closing braces as long as the next statement isn't part of a continuing language construct (e.g. `if` / `else`, `try` / `catch` / `finally`).

```
// Good
public static string ToCamelCase(this string text)
{
    if (string.IsNullOrWhiteSpace(text))
    {
        return text;
    }

    return char.ToLower(text[0]) + text.Substring(1);
}

// Bad
public static string ToCamelCase(this string text)
{
    if (string.IsNullOrWhiteSpace(text))
    {
        return text;
    }
    return char.ToLower(text[0]) + text.Substring(1);
}
```

19. Do not exceed about 30-40 lines of code in a single method.  After that, refactor the method into smaller well-named methods.
20. Limit the length of a line of code to about 100-110 characters to reduce unnecessary scrolling.  Github shows about 113 characters, and many developers now use their monitors in "portrait" orientation.
21. Use collection and object initializers on lists and dictionaries when possible.

```
// Good
var claims = new List<Claims>
    {
        new Claim("name1", "value1"),
        new Claim("name2", "value2"),
    };

var studentById = new Dictionary<int, Student>
    {
        { 123456, new Student { Name = "John Doe", Age = 17} },
        { 234567, new Student { Name = "Jane Doe", Age = 16} },
    };

// Bad
var claims = new List<Claims>();
claims.Add(new Claim("name1", "value1"));
claims.Add(new Claim("name2", "value2"));

// Even Worse
var studentById = new Dictionary<int, Student>();

var student1 = new Student();
student1.Name = "John Doe";
student1.Name = 17;

var student2 = new Student();
student2.Name = "Jane Doe";
student2.Name = 16;

studentById.Add(123456, student1);
studentById.Add(234567, student2);
```

22. Use case-insensitive checks rather than converting the casing of strings for case-sensitive comparison.

```
string value1 = "Bob";
string value2 = "BoB";

// Good
if (value1.Equals(value2, StringComparison.InvariantCultureIgnoreCase))
    return true;

// Bad
if (value1.ToLower() == value2.ToLower())
    return true;
```

23. Namespace imports should be specified at the top of the file, *outside* of `namespace` declarations and should be sorted alphabetically, with `System.` namespaces at the top.

```
// Good
using System;
using System.Web;
using EdFi.Ods;
using EdFi.Ods.Common;

namespace EdFi.Ods.Data
{
    …
}
```

```
// Bad (not alphabetical)
using EdFi.Ods;
using System.Web;
using System;

namespace EdFi.Ods.Data
{
    // Bad ("using" is nested within namespace)
    using EdFi.Ods.Common;
    …
}
```

24. Class artifacts should be organized as follows:
    a. Fields (primarily holding injected dependencies)
    b. Constructor(s)
    c. Public members (properties and methods)
        i. Define property-backing fields immediately before the property.

```
private IEnumerable<Student> _students;

public IEnumerable<Student> Students
{
    get { return _students; }
}
```

        ii. Define non-shared supporting methods immediately following the method they were introduced to support.

```
public void DoSomethingInteresting()
{
    …
    int something = GetSomething();
    …
    int anotherThing = GetAnotherThing();
}

private int GetSomething()
{
    …
}

private int GetAnotherThing()
{
    …
}

public void DoSomethingCompletelyDifferent()
{
    …
}
```

25. Configuration files should be organized as follows:
    a. `configSections` (optional)
    b. `appSettings`
    c. `connectionStrings` (optional)
    d. Custom sections, matching the order of entries in the `configSections` section.
    e. Everything else...

**Correctly ordered configuration file**

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="hibernate-configuration" type="NHibernate.Cfg.ConfigurationSectionHandler,
NHibernate" />
    <section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler, log4net" />
    <section name="inversionOfControl" type="EdFi.Common.Configuration.
InversionOfControlConfiguration, EdFi.Common, Version=1.0.0.0, Culture=neutral" />
  </configSections>
  <appSettings>
    <add key="AJAXAllowedDomains" value="*" />
    …
  </appSettings>
  <connectionStrings>
    <add name="EdFi_Ods" connectionString="Database=EdFi_Ods; Data Source=(local);
Trusted_Connection=True;" providerName="System.Data.SqlClient" />
    …
  </connectionStrings>
  <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
    <session-factory>
      <property name="dialect">NHibernate.Dialect.MsSql2008Dialect</property>
      …
    </session-factory>
  </hibernate-configuration>
  <log4net>
    <appender name="TraceAppender" type="log4net.Appender.TraceAppender">
        …
    </appender>
    …
 </log4net>
  <inversionOfControl>
    <installers>
      <add name="Configuration Specific Installer" typeName="EdFi.Ods.WebApi._Installers.
{Configuration}.ConfigurationSpecificInstaller, EdFi.Ods.WebApi" />
    </installers>
  </inversionOfControl>
  <system.web>
    <compilation debug="true" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
    …
```

26. Favor a coding style of "fail fast" (and also "exit fast"). If there is an exception to be thrown or a value to be returned with no further logic, perform that logic immediately rather than leaving it for an `else` clause.

```
// Good
public void SomeMethod(ISomethingProvider somethingProvider)
{
    // Check for known failure condition and exit quickly
    if (somethingProvider == null)
        throw new ArgumentNullException("somethingProvider");

    var things = _somethingElseProvider.GetThoseThings();
    …
}

// Bad
public void SomeMethod(ISomethingProvider somethingProvider)
{
    if (somethingProvider != null)
    {
        var things = _somethingElseProvider.GetThoseThings();
        … // Lots of logic here that a maintainer has to scroll through.
    }
    else
    {
        // This should be moved to the top of the method
        throw new ArgumentNullException("somethingProvider");
    }
}
```

27. Do not use regions (i.e. `#region`).
28. Avoid the use of the `dynamic` keyword, particularly in code that executes frequently. Dynamic dispatch is very expensive and is rarely actually needed.
29. Avoid using `Tuple` in C#. They were added to support F#, which has far more expressive syntax with support for *pattern matching*. In C#, they end up being used as structures without any semantics.

```
// F#
let t5 = ("hello",42)
let t6 = ("goodbye",99)

// C#
var t5 = Tuple.Create("hello", 42);
var t6 = Tuple.Create("goodbye", 99);
```

30. Avoid passing values around using `KeyValuePair<TKey, TValue>`, but if you do, use a name format like "{*keyName*}And{*valueName*}" to ease maintenance (e.g. use `schoolIdAndName` for an entry from a dictionary named `schoolNameById`).
31. Express exception messages as grammatically correct sentences.
    a. When appropriate, format the message using `string.Format` and wrap parameterized values with single quotes (`'`).

```
throw new Exception(string.Format("There is no data for resource '{0}'.", resourceName));

throw new Exception(string.Format("Type '{0}' not found in assembly '{1}'.", typeName,
assemblyName));
```

    b. For exceptions that are targeted for a developer audience (such as configuration errors that shouldn't happen in production), you may include a helpful suggestion or question to help resolve the error message.
32. Favor adding XML documentation for public methods and properties of interfaces and classes.
    a. Follow documentation conventions from Microsoft (when in doubt, explore http://msdn.microsoft.com as a style reference):
        i. Start the XML summary of methods and properties as though you are finishing a sentence that has already started with the text "*This property or method …*"
        ii. Use `<see cref="SomeOtherClass" />` to reference other classes in the documentation.
        iii. Use `<b>true</b>` tags to emphasize keywords like `true`, `false` and `null`.
        iv. Constructors – "Initializes a new instance of the {*classname*} class [using the specified …]."
        v. Properties – "Gets or sets the …" or "Gets the …"
        vi. Methods – "Appends a copy of the specified string…."
        vii. When specifying <return> documentation, use a format like you're finishing a sentence that starts with "Returns…", and use semi-colons to add the "otherwise" condition.

**Actual content for .NET's String.Contains "returns" documentation**

```
<returns><b>true</b> if the value parameter occurs within this string, or if value is the
empty string(""); otherwise, <b>false</b>.</returns>
```

b. Do not use XML documentation blocks for comments.

```
/// <summary>
/// ClaimName is actually an Uri so length needs to be around 2048
/// </summary>
[StringLength(2048)]
[Required]
public string ClaimName { get; set; }
```

Instead, use the XML for documentation, and leave comments where they can be found by the developer maintaining the code.

```
/// <summary>
/// The name of the claim, expressed as a URI.
/// </summary>
// ClaimName is actually an Uri so length needs to be around 2048
[StringLength(2048)]
[Required]
public string ClaimName { get; set; }
```

33. Define custom exception classes when you expect the exception to be explicitly handled somewhere else in your application. It's a lot more reliable to match an exception based on its type than by extracting information from the message.
34. Do not reuse .NET framework exceptions outside of their semantic context (e.g. `ObjectNotFoundException` is defined in multiple places, and `AuthorizationFailedException` and `SecurityAccessDeniedException` have specific context where they're used).

# Patterns and Idioms

1. Create "marker" *interfaces* (not classes) in the root namespace of assemblies to assist with easily obtaining `Assembly` references, particularly for reflection-based queries. Name the interface using a `Marker_` prefix, and replace the periods in the assembly's root namespace with underscores.

```
public interface Marker_EdFi_Ods_Common { }
```

2. Group extension methods into classes as follows:
   a. By the Type being extended, with the file named accordingly.

**IEnumerableExtensions.cs**

```
public static class IEnumerableExtensions
{
    public static void Do<T>(this IEnumerable<T> enumerable, Action<T> action)
    {
        foreach (var element in enumerable)
            action(element);
    }

    public static bool None<T>(this IEnumerable<T> enumerable)
    {
        return !enumerable.Any();
    }
}
```

   b. By theme, with the file named using a `Helper` suffix.

**ComponentNameHelper.cs**

```
public static class ComponentNameHelper
{
    public static string GetServiceNameWithSuffix(this Type serviceType, string suffix)
    {
        …
    }

    public static ComponentRegistration<T> NamedForDatabase<T, TEnum>(this
ComponentRegistration<T> registration, TEnum database)
        where T : class, IDatabaseConnectionStringProvider
    {
        …
    }
}
```

3. Avoid taking dependencies on static .NET Framework classes whose results are not idempotent or require access to infrastructure (e.g. `System.DateTime`, `System.IO.File` and `System.Configuration.ConfigurationManager`, but not `System.IO.Path`). Instead, define an interface using a naming convention of `I`{*StaticClassName*} and build a minimal (but expanding, as necessary) facade with an implementation named as {*StaticClassName*}`Wrapper`. This will enable the behavior to be controlled for testing purposes.
    a. For `System.IO.File`, this would look like this:

```
public interface IFile
{
    bool Exists(string path);
}

public class FileWrapper : IFile
{
    public bool Exists(string path)
    {
        return File.Exists(path);
    }
}
```

    b. The functionality of the `ConfigurationManager` has been abstracted using a different approach due to legacy code reuse. Review the `IConfigValueProvider`, `IConfigSectionProvider` and `IConfigConnectionStringsProvider` interfaces and associated implementations.
4. When providing an implementation of an interface that does nothing, use the Null Object Pattern.

```
public interface IETagProvider
{
    string GetETag(object value);

    DateTime GetDateTime(string etag);
}

public class NullETagProvider : IETagProvider
{
    public string GetETag(object value)
    {
        return string.Empty;
    }

    public DateTime GetDateTime(string etag)
    {
        return default(DateTime);
    }
}
```

5. The Provider Pattern (also known as the Strategy Pattern) is used heavily in the code base.
    a. If the contract being defined basically allows the caller to *get* something, the `Provider` suffix is preferred.
    b. If the contract allows the caller to *get* and *set* something, consider splitting the methods into `Reader` and `Writer` interfaces (which could still be implemented by the same class). The intent here is to provide more clearly stated intent when a caller intends to modify the underlying data exposed by the provider.

```
// Instead of this...
public interface ICacheProvider
{
    void RemoveCachedObjects(string keyContains);
    void RemoveCachedObject(string keyName);
    bool TryGetCachedObject(string key, out object value);
    void SetCachedObject(string keyName, object obj);
    void Insert(string key, object value, System.Web.Caching.CacheDependency dependencies,
DateTime absoluteExpiration, TimeSpan slidingExpiration);
    void Insert(string key, object value, DateTime absoluteExpiration, TimeSpan slidingExpiration);
}

// Consider decomposing the interfaces ...
public interface ICacheReader
{
    bool TryGetCachedObject(string key, out object value);
}

public interface ICacheWriter
{
    void RemoveCachedObjects(string keyContains);
    void RemoveCachedObject(string keyName);
    void SetCachedObject(string keyName, object obj);
    void Insert(string key, object value, System.Web.Caching.CacheDependency dependencies,
DateTime absoluteExpiration, TimeSpan slidingExpiration);
    void Insert(string key, object value, DateTime absoluteExpiration, TimeSpan slidingExpiration);
}
```

   c. If the contract being defined performs some sort of action, the suffix may be removed entirely.

```
public interface IObjectValidator
{
    ICollection<ValidationResult> ValidateObject(object @object);
}
```

6. Use creational patterns for creating objects that involve significant logic.
   a. When the logic required to create something doesn't fit well into a class constructor, use a creational pattern like the Factory Pattern.
   b. When an item can built in one step, use "Create" semantics (see the Factory Method Pattern).
   c. When an item is built up over multiple steps, using "Build" semantics (see the Builder Pattern).
7. When a class has the concept of an empty instance, implement a static read-only `Empty` property.

```
public class UserLookupResult
{
    public static readonly UserLookupResult Empty = new UserLookupResult();
    …
}
```

# Design Guidelines

1. Apply SOLID principles everywhere. (Highly recommended reading: Chapters 8-12 of Agile Principles, Patterns, and Practices by Robert C. Martin.)
   a. **S**ingle Responsibility Principle - There should only be one reason for a class to change. If you're having to change an existing class, ask yourself, "Is there an abstraction (interface) missing here?"
   b. **O**pen/Closed Principle - The system should be open for extension, but closed to modification. You should be able to change the behavior of the system by adding a new implementation of an existing interface.
   c. **L**iskov Substitution Principle - A derived class should be substitutable for its base class, and should not change the fundamental nature of the abstraction.
   d. **I**nterface Segregation Principle - An interface should be highly focused. Many interfaces in the ODS API have just one or two methods. If you implement some methods of an interface using `new NotImplementedException()`, the interface probably needs to be decomposed.
   e. **D**ependency Inversion Principle - External dependencies are injected into classes, preferably via their constructors. The Ed-Fi ODS API uses Castle Windsor as its Inversion of Control container.
2. Class design
   a. Avoid deep class hierarchies (more than 2 total levels). Prefer composition over inheritance.
   b. Keep logic in constructors simple, primarily focused on capturing the injected dependencies to the field values.

c. Do not use properties to modify class state outside of the property being set. If other state must be changed, prefer the use of a read-only property in combination with a well-named method.
d. Prefer the use of return types that are abstractions rather than concrete types from public members. In other words, prefer `IList<T>` to `List<T>` and `IDictionary<TKey, TValue>` over `Dictionary<TKey, TValue>`.
e. Make sure the return types from public members match the intended semantics. In other words, prefer `IEnumerable<T>` over `IList<T>` and `IReadOnlyDictionary<TKey, TValue>` over `IDictionary<TKey, TValue>`, unless modifications by the caller are intended.

```
// Good
public interface IStudentDataProvider
{
        IEnumerable<Student> GetAll();
    IReadOnlyDictionary<string, Student> GetStudentByNameDictionary();
}

// Potentially problematic for maintenance, due to semantics of returned values
public interface IStudentDataProvider
{
    IList<Student> GetAll();
    IDictionary<string, Student> GetStudentByNameDictionary();
}
```

f. Do not use the `new` inheritance qualifier. Reevaluate the design of the class.
g. Do not make members of a class `public` for the sake of unit testing. Find a better way to test the functionality.

> ⚠ **Inversion of Control**
>
> The guidance for this topic is still under discussion.

a. Prefer constructor injection to property injection.
b. Avoid use of the `IoC.Resolve` static method unless you're not in control of the creation of the class needing the dependencies.
c. Use the `IServiceLocator` interface only for classes that are providing convention-based component creation (e.g. a class processing messages from a bus that is responsible for identifying and instantiating the corresponding message handlers based on naming conventions).

# Unit Testing Style Guidelines

T.B.D.

- Discussion of Test Doubles: Mocks, Fakes, Stubs - Mocks Aren't Stubs

# Git Usage Guidelines

T.B.D.

Here is a brief overview of the recommended git workflow:

1. Create a personal fork of the main repository at Github.
2. Clone your forked repository to your workstation. (`git clone`)
3. Add the main repository as a remote. (`git remote add`)
4. Do all work on a feature branch off the latest main branch.
5. Commits should have one vector of change. Use git's *interactive add* feature to stage individual lines from files. (`git add -i`)
6. Clean up your commit history on the branch before issuing the pull request (PR).
    a. Use git's *interactive rebase* feature to collapse unnecessary noise. (`git rebase -i`)
    b. Commits should not exist in the PR that contain message like "added tests" or "fixed tests" – *squash* these commits.
7. Create the pull request.
    a. Pull the latest from the main repository.
    b. Rebase your branch on the end of the target branch.
    c. Push the branch to your personal fork at Github.
    d. Issue issue the pull request to the main repository using a description in the format of *[ODS-123] Brief issue description here*.

# Acknowledgements

These coding standards are based partly on the following sources:

- IDesign C# Coding Standard 2.4, as published at www.idesign.net. Copyright (C) 2011, IDesign Inc., All Rights Reserved. Used under license.
- The "Contributing" guide for Microsoft's .NET Core github repository at https://github.com/dotnet/corefx/wiki/Contributing.