Cooper Tire & Rubber Co.

Code Review Topics

Originator: Keith DuPont

Issue Date: 01/2015

Revision Number	Revision Date	Comments	Revised By	Approved By
1.0	**12/17/2014**	**Initial Revision**	**Keith D. DuPont**	
1.1	**1/21/2015**	**Edits/additions** **Added info on ABAP and JAVA**	**Keith D. DuPont**	
1.2	**7/31/2015**	**Minor updates**	**Keith D. DuPont**	
1.3	**02/01/2016**	**Multiple additions** **Updated: 2.1, 2.5, 10.15, 10.16** **Added: 2.16, 2.17, 2.18, 2.19, 2.20, 5.16, 5.24, 5.37, 5.38, 5.6.10, 5.6.11, 5.6.12, 8.6** **Added Sections: 5.7, 11**	**Keith D. DuPont**	
1.4	**05/05/2016**	**Updated: 9.6, 9.9** **Added Sections: 2.2.21, 5.1.7, 5.4.4, 5.4.5**	**Keith D. DuPont**	
1.5	**03/24/2017**	**Updated: 2.1** **Added Sections: 2.22, 2.23, 2.24, 2.25, 7.4, 7.5, 10.17, 12.4**	**Keith D. DuPont**	
1.6	**9/6/2018** **----------------** **12/21/2018**	**Updated 5.6.7, 5.7.1** **Added Sections: 5.8. 5.8.1, 12.5, 12.6, 12.7** **----------------------** **Added sections: 1.3, 10.18** **Updated sections: 2.25, 5.2.4, 5.3.5, 5.6.10** **Inserted new section 4.5. Existing sections 4.5 and 4.6 became sections 4.6 and 4.7** **Corrected typographical errors throughout**	**Keith D. DuPont** **---------------** **Rick Marshall**	

Table of Contents

TOC

Overview

This document provides an overview of items that will be evaluated when performing a code review. Note that evaluation of business logic, while a critical component of coding (one would argue the most critical) receives only a bullet point in this document. This is due to business logic requirements being specific to an individual system/application.

NOTE: This document is considered a "living document" and will continually be updated as additional code review items are identified and documented.

Introduction

Our process of formalizing the code review process is continually evolving. This document is to be used when conducting code reviews of any Cooper Tire code. This applies to code created internally or by third party partners. This applies to code running inside the Cooper firewall as well as to code hosted externally. This applies to new code as well as enhancements to existing code.

Many sections of this document are specific to Microsoft Visual Studio .NET while others are technology agnostic and apply to any development tool. In cases where code review checklists and/or coding standards specific to other tools/technologies are identified links to them will be added to this document. See Additional Guidance and Documentation below for information specific to those tools/technologies.

Additional Guidance and Documentation

This document is not intended to replace any existing code review standards and/or checklists. In some cases this document may overlap with pre-existing code review standards and/or checklists specific to a certain tool and/or technology.

Technology	Document	Location
ABAP	ABAP Development Standards	Located in SOLMAN -- \Business Scenarios\Technology Architecture\Business Processes\Development Architecture
ABAP	ABAP Naming Standards	Located in SOLMAN -- \Business Scenarios\Technology Architecture\Business Processes\Development Architecture
ABAP	Code Review Checklist.xls	The code review checklist used in ABAP development can be added to any folder in SOLMAN by clicking the icon and choosing "Code Review Checklist' from the "Documentation Type" drop down list

General Guidelines/Comments

Since we are just beginning to formalize our code review process (December 2014) some general guidelines and comments will be helpful. As we move forward with this initiative our long-term goal will be to make code reviews an inherent part of our development process. This implies that in the future increasingly larger portions of our entire codebase will have been reviewed.

1.1 Identifying what should be reviewed

This is simple. Review everything. Review all objects, configuration files and documents that are included in the solution.

1.2 Why should I review the entire solution if only one project was modified?

For maintenance and enhancement work it is common to update specific portions of the codebase (as opposed to major portions of the codebase). Until we get to the point in which the majority of the codebase for a system has been reviewed, it is strongly recommended that the entire solution be reviewed (even if only a subset of the codebase has been updated).

While this admittedly takes more time to complete it does provide several benefits, such as:

- The reviewer obtains an overview of the overall cleanliness of the code as well as an understanding of the design structure of the solution.
- Variances from the code review standard are identified. These variances identify future opportunities for fixing and/or cleaning-up code.
- Resolving the opportunities identified in the review can result in cleaner, more-stable code.

1.3 Requesting a Code Review

Depending on work loads, code reviewers may need several days' notice that a code review request is being made. In practice, a request for code review should be made five business days prior to the date the code will be ready for review. To request a code review, submit the document RequestForCodeReview.xlsx to the code review team.

For a code review to be initiated, the code must be stored in the applicable source code repository (e.g., TFS).

Review Items at the Solution/Project Level

Various objects at the solution and project level should be reviewed prior to beginning the review of individual objects contained within individual projects. Examples of solution and/or project-level items that are reviewed include (but are not limited to) design considerations, compilation settings, configuration settings etc.

2.1 Ensure separation of concerns principles are not violated

Review the design of the solution/projects and ensure that separation of concern principles are not violated. A primary example of violating a separation of concerns principle is having the Business layer or UI layer directly interact with the database (instead of calling a method in the Data layer). Having the Business layer or UI directly interact with the database is a violation of the separation of concerns and would be flagged as a code review violation.

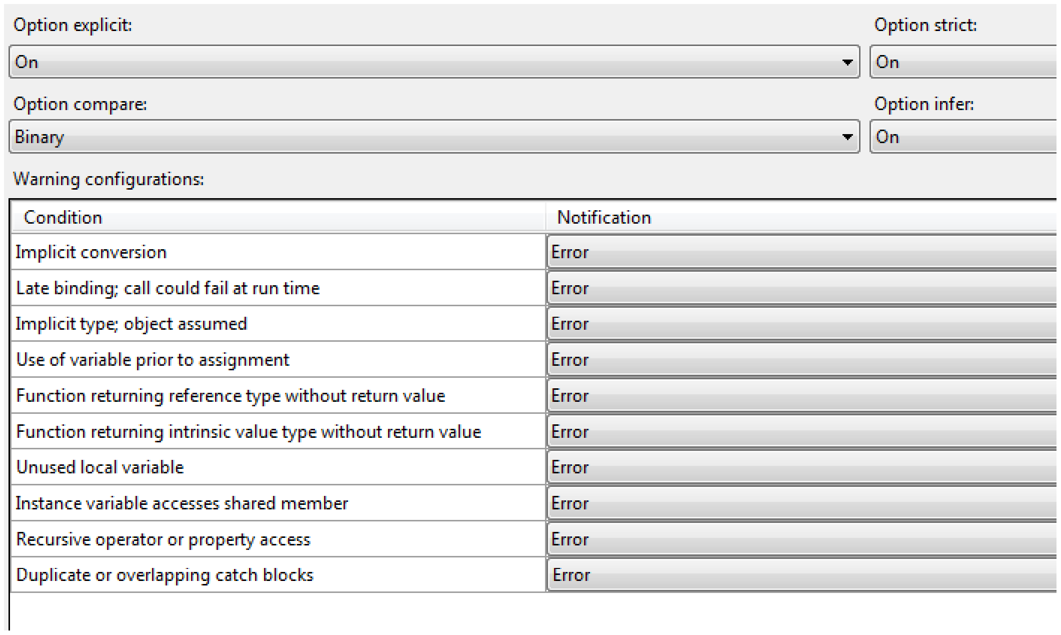
Other examples include: a UI layer or a Business layer knowing the names of specific stored procedures or other database object/parameter information; Not having a Business layer

2.2 Ensure compiler settings are as restrictive as possible

In development tools that support configuration of compilation settings, always ensure that the compilation settings are set to their most restrictive level.

Examples of compilation settings for Visual Studio .NET projects are shown below.

<u>Visual Basic .NET</u>
For Visual Basic .NET projects this means that all settings in the Project Properties – Compile tab should be set to “ERROR”, and that Option Strict and Option Explicit should be set to “ON” (see screen capture below).



Option explicit: On

Option strict: On

Option compare: Binary

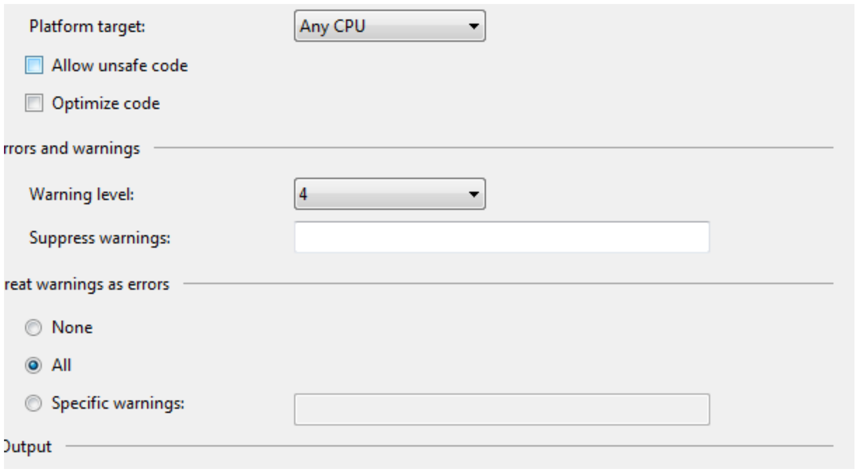
Option infer: On

Warning configurations:

Condition	Notification
Implicit conversion	Error
Late binding; call could fail at run time	Error
Implicit type; object assumed	Error
Use of variable prior to assignment	Error
Function returning reference type without return value	Error
Function returning intrinsic value type without return value	Error
Unused local variable	Error
Instance variable accesses shared member	Error
Recursive operator or property access	Error
Duplicate or overlapping catch blocks	Error

<u>C#</u>
For Visual Studio .NET C# projects this means that the following settings in the Project Properties – Build tab should be set as follows (see screen capture below):

- "Allow unsafe code" should <u>not</u> be checked
- "Warning level" should be set to "4"
- "Treat warnings as errors" should be set to "All"



Platform target: Any CPU

☐ Allow unsafe code

☐ Optimize code

rrors and warnings

Warning level: 4

Suppress warnings:

reat warnings as errors

○ None

● All

○ Specific warnings:

Dutput

2.3 Ensure code compiles

A critical step of the code review is to ensure that the code actually compiles. This applies to individual projects as well as the entire solution (ensuring that all required projects are included in the solution-level build).

After attempting to build the solutions and projects it will be necessary to investigate if any of the following occur:

- Compilation failures
- Compiler warnings are generated during compilation
- Undeclared variables are identified
- Unused variables are identified
- Uninitialized variables are identified
- Use of deprecated methods/functionality is identified

2.4 Ensure file backups (.bak, .old etc.) do not exist in solution/project structures

Inspect the directory/file structure of the solution (including all subdirectories) and look for any backups of files. These could have the extension of .old, .bak, .DoNotDeleteMe etc. These backups could have been intentionally created by a developer or may have been automatically created by an editing tool (Example: Some merge/compare tools create .bak files automatically whenever a file is updated). These types of file backups should not be included in any solution and/or project.

2.5 Ensure solution/project structure is clean and well-structured

Review the structure within the development tool's solution explorer and the file system structure to ensure that the solution and project(s) are clean and well-structured.

Ensure that a common "lib" folder is used at the solution level to maintain/contain common reference assemblies.

Please refer to the following document for additional information on structuring Visual Studio .NET Solutions and Projects.

2.6 Look for missing objects that should be included in the solution/project

Look for obvious signs that something has not been included in the solution/project that should be. Examples could include (but are not limited to):

- Documentation
- Does the code fail to compile due to a missing reference or object?
- Do items appear as "excluded" in the Solution Explorer in Visual Studio .NET?
    - If so, is this intentional or should they be included?
- A setup project for a Windows form application
- Application configuration files
- Web.config files
- If resources files are defined for specific regions/cultures (example: en, en-US, es, es-MX) are entries provided for each setting in each resource file?
    - If not, should they be?
- Does at least one DEBUG and at least one RELEASE configuration exist?
    - See debug/release requirements for more info

2.7 Look for "extra" objects/files that have no value

Review the solution, project(s) and file system structure to identify objects that are present that do not need to be. This includes (but is not limited to) the following:

- Classes that are not being used
- Unused resource files

- Unused projects
- Files that exist in the solution/project directory structure but that are not used to build the code or are not relevant to the project. These extra artifacts could be the result of deleting something from the project/solution but not removing the debris from the directory structure.

2.8 Do not generate .pdb files when building RELEASE configurations

Visual Studio .NET project configurations should not generate .pdb (debug) files when compiling for RELEASE configurations. .pdb files should only be generated for DEBUG configurations. This is also applicable to other development tools that may generate debug objects.

2.9 Ensure the correct version of the Microsoft .NET Framework is targeted

Beginning with Visual Studio .NET 2008 it is possible to target applications/assemblies for specific versions of the Microsoft .NET Framework (hereafter referred to as “FX”).

- Ensure that the correct version of the .NET FX is specified in the project configuration(s).
- Ensure that applications/objects are not configured to target the “.NET Framework 4 Client Profile”. This specific FX configuration does not support the System.Data.OracleClient namespace which is used in many Cooper applications.

2.10 Ensure the correct version of Java/JRE is targeted when developing with IDEs that support Java

Eclipse (and some other IDEs that support Java) support targeting specific versions of the Java JRE. Ensure that the appropriate version of the target JRE is specified in the solution/project configuration.

2.11 Ensure that at least one Debug and at least one Release configuration is defined

Each Visual Studio .NET solution must have at least one (1) DEBUG and at least one (1) RELEASE configuration. This requirement should also apply to other development tools/IDEs that support multiple solution/project configurations.

2.12 If different configurations are required for different locations create a Debug and Release build for each location and environment

Creation of a location/environment-specific configuration simplifies the process of creating location/environment specific builds and minimizes the chances of configuration errors that could occur due to manual misconfiguration. This requirement should also apply to other development tools/IDEs that support multiple solution/project configurations.

2.13 RELEASE configurations should be configured in a Production-ready mode

RELEASE configuration settings should be configured in a production-ready manner. Production-ready settings include (but are not limited to):

- No debugging settings enabled
- .PDB files are not generated (or deployed to production)
- Appropriate application/web configuration settings are used
- Debugging code is either not present or is decorated with appropriate directives to ensure that it is not compiled/included when building in RELEASE mode

2.14 Is any of the code/data/objects in the solution/project a candidate for additional Intellectual property protection (ex: obfuscation) ?

After the code has been reviewed ask the question if the code is a candidate for additional protection of the intellectual property contained within the code. If you are uncertain if this code qualifies for additional protection, follow-up with a Cooper architect or with your Cooper manager to discuss. Cooper has the ability to provide additional protection to assemblies created in Visual Studio .NET and additional steps can be taken to provide additional protection to both Oracle and SQL Server objects. Contact a Cooper application architect to implement this additional protection.

2.15 Ensure only Cooper-approved tools (and versions of tools) are used

Specific development tools, specific versions of development tools and 3rd party components have been approved for use by Cooper licensing administrators. Always check solutions and/or projects to ensure that no new tools have been used without prior authorization and license review/approval by Cooper licensing administrators.

As mentioned above, this also applies to specific versions of tools. Several scenarios have been encountered in which developers/partners have used a newer version of a tool than was approved/supported within the Cooper environment.

When in doubt as to the status of tool and/or component, follow-up with the Cooper licensing administrator or Cooper application architect for confirmation as to the approved status of a specific tool or approved version of a tool.

2.16 Ensure compiler settings are consistent between solutions and between projects within a solution

All compiler settings and options should be consistent betweens solutions and between projects contained within the same solution. The only exceptions to this are:

1. There is a specific, approved technical requirement to vary a compilation setting.
2. Prior approval has been provided by a Cooper architect to vary compiler settings.

2.17 Ensure Code Analysis rule sets are consistent between solutions and between projects within a solution

Rule sets used for Code Analysis should be consistent betweens solutions and between projects contained within the same solution. The only exceptions to this are:

1. There is a specific, approved technical requirement to vary code analysis rule set settings.
2. Prior approval has been provided by a Cooper architect to vary code analysis rule set settings.

2.18 Ensure consistent, approved namespaces are used between solutions and between projects within a solution

Namespaces should be consistent betweens solutions and between projects contained within the same solution. The only exceptions to this are:

1. There is a specific, approved technical requirement to vary a namespace.
2. Prior approval has been provided by a Cooper architect to vary a namespace.

2.19 Ensure solutions and/or projects do not contain references to any specific item on a specific developer’s workstation

Examples include (but are not limited to): a location on a developer’s desktop, a web service running on a developer’s local workstation, a tool that may only be installed on the developer’s workstation

2.20 Do not use IIS Express for web projects

Always use Local IIS instead of IIS Express. Local IIS can and should be configured to match the local Cooper environment

2.21 Always ensure that “Internal Compiler Error Reporting” is set to “None”

If the “Internal Compiler Error Reporting” setting is present ensure that it is set to “None” in all solutions/projects.

2.22 Always use the Appropriate pragma settings in C# to disable decprecation warnings related to use of the System.Data.OracleClient (and other System.Data.Oracle*) Namespaces

Required warnings to disable at the beginning of a class and to restore at the end of a class are: 612, 618

NOTE: Use of the above pragma settings will disable these warnings from being generated when building C# code that uses the System.Data.Oracle* namespaces.

2.23 Ensure that the "Disable all warnings" checkbox is <u>not</u> checked in VB .NET project compilation settings

The "Disable all warnings" checkbox should <u>never</u> be checked in VB .NET project compilation settings. There are <u>no</u> exceptions to this requirement!

2.24 Ensure the "Treat all warnings as errors" checkbox <u>is</u> checked in VB .NET project compilation settings

The "Treat all warnings as errors" checkbox should be checked in VB .NET project compilation settings. See 2.25 (immediately below) for the only recognized exception to this requirement.

2.25 Acceptable warnings in VB .NET code.

In VB code from version 2010 and prior, deprecation warnings for use of System.Data.Oracle* namepaces are allowed due to the inability to suppress those warnings via the use of pragma directives.

In VB code from version 2013 and later, follow the instructions at the following link to suppress deprecation warnings for the use of System.Data.Oracle.* namespaces.

https://docs.microsoft.com/en-us/visualstudio/ide/how-to-suppress-compiler-warnings?view=vs-2015#suppressing-warnings-for-visual-basic

No other warnings should be generated during compile.

Web Services

3.1 Ensure valid namespace Uniform Resource Identifier (URI) is used

When reviewing code that contains web services check to ensure that the default namespace URI that is created when adding a web service to a project has been changed to reflect the system/application. Accepting the default value of http://tempuri.org is not acceptable.

Example of unacceptable namespace URI:

C#: `[WebService(Namespace = "http://tempuri.org/")]`

VB: `<System.Web.Services.WebService(Namespace:="http://tempuri.org/")>`

Example of an acceptable namespace URI:

```
[WebService(Namespace =
"http://CooperTire.CooperServiceBroker.WebServices.Public.NonSAP")]
```

AssemblyInfo and Metadata

Review the AssemblyInfo.vb or AssemblyInfo.cs file for every project in a Visual Studio.NET solution. It is very important to review the specific items discussed below.

4.1 Ensure Assembly description is provided

An assembly description must be provided. Ensure that the description accurately describes the function of the assembly at a high level.

4.2 Ensure copyright is assigned to Cooper

Ensure that the Copyright is assigned to Cooper Tire & Rubber Company.

4.3 Ensure assembly versions are explicitly defined

Assembly version number should be explicitly assigned. Do not allow any assembly version settings in which an asterisk (“*”) is used.

4.4 Ensure trademark is assigned

Ensure the trademark is assigned to Cooper Tire & Rubber Company.

4.5 Ensure unused assembly items are removed.

Remove the AssemblyConfiguration and AssemblyCulture lines.

4.6 If a strong-name key is referenced ensure that it exists and is part of the .NET solution

Inspect the AssemblyInfo settings to determine if a strong-name key is used to sign the assembly. If a strong-name key is specified ensure that the strong-name key file is included in the project/solution.

4.7 If a strong-name key is referenced ensure that the path to the key is part of the solution structure and not a path to a location outside the solution structure

Inspect the AssemblyInfo settings to determine if a strong-name key is used to sign the assembly. If a strong-name key is specified ensure that the path referencing the strong-name key file is to a location that exists in the project/solution directory path.

Code Specifics

5.1 Variables

5.1.1 Ensure all variables are initialized

Ensure all variables are initialized. Never allow the compiler to initialize variables for you.

5.1.2 Ensure all variables are initialized to a proper value

Ensure all variables are initialized to a valid value. If strings are not initialized to a specific value initialize them with String.Empty()

5.1.3 Ensure meaningful variable names are used

Ensure that meaningful, descriptive and accurate variable names are used. Do not use extremely short variable names (example: “i” … or “a” etc.).

Ensure that variable names from “copy & pasted” code have been changed to accurately reflect the context of how they are being used.

5.1.4 Variable scope is appropriate

Ensure that the scope of a variable is appropriate to its use. Do not use global variables when local variables are a better fit.

5.1.5 Ensure variable/parameter types (and sizes) are defined appropriately

Check to ensure that parameter types (and sizes, where applicable) are defined correctly. This ensures that the code is more stable and also prevents unintended side effects. The following two (2) examples from real Cooper code illustrate issues that should be flagged in a review.

1. Code was encountered in which a variable was declared to store the value of a sequence retrieved from a database. The maximum value that the sequence would generate was significantly larger than the size of the .NET variable that was defined to store the sequence.

2. A .NET OracleParameter was defined to hold a string that would be passed to a stored procedure which would store the value in the database. Shortly after deployment issues were reported indicating that the string being stored in the database was always 22 characters in length. This occurred even if the client entered more than 22 characters of text. Subsequent debugging/investigation identified the problem as follows: Instead of setting the .Size value of the parameter to the actual size of the text entered by the client an enumeration was incorrectly used to set the .Size value of the parameter. (see below):

```
test.Size = OracleType.VarChar;
```
//the value of this enumeration is 22

This coding error caused every value to be truncated to 22 characters in length when it was stored in the database. If this wasn't bad enough, the code was copied & pasted more than **200 times** (!!) across multiple applications. This drastically compounded the impact of the original error as well as the time required for several Cooper developers to track down and fix the issue.

5.1.6 Ensure application settings related to Cooper network/environment are appropriate and approved

All application settings related to Cooper network/environment should be reviewed and approved. Examples include but are not limited to: SMTP setttings, Server names, DFS settings, DNS settings etc.

5.1.7 Use consistent variable naming conventions throughout the codebase

Variable naming conventions should be consistent throughout the codebase. Do not use Pascal casing in one class/method and then use camel casing in another class/method.

5.2 Localization

Cooper has started requesting that all new applications (along with significant changes to existing applications) be localized to support potential deployments to locations outside of the United States.

5.2.1 Break user-facing messages out into resource files where appropriate

There should be no user-facing messages (prompts, messages, warning etc.) that are hard-coded.

All user-facing message verbiage should be defined in resource files to allow them to be changed without re-compilation of the main application code/objects. Resource files also make it easier to support multiple languages with no code changes.

5.2.2 Ensure code is localized with en-US as the first language supported

Code should be localized with en-US as the first language. This should be implemented even if there are no current plans to take the application to a non-US location.

5.2.3 Use String.Format with resource settings to construct user-facing messages

Code should not utilize manual concatenation of strings to construct user-facing messages. String.Format should be used in conjunction with verbiage stored in resource files.

5.2.4 Ensure all colors are defined with resource settings

This ensures that colors can be changed easily with no changes to code. Name the resource setting for the condition being represented, not the color being used. For example, if text for an error message is to be set to red, name the resource setting ColorErrorText, not ColorRed.

5.3 Strings

5.3.1 Ensure String.Empty is Used Instead of “”

Do not use “”when initializing strings or setting string values. Always use String.Empty. This especially true when performing logical comparisons.

5.3.2 Ensure String.IsNullOrEmpty is used for string tests

Ensure that String.IsNullOrEmpty() is used when determining if a string contains non-null or non-empty data. This is more efficient than performing other tests.

5.3.3 Use String.Format() when creating/concatenating strings

Always use String.Format() when concatenating strings. This is much more efficient than using the “+” or “&” operators.

5.3.4 Minimal use of hard-coded, duplicated strings

Ensure the use of hard-coded, duplicated strings is minimized. If the same string token is used in multiple places it is a candidate to be moved to a constant or a resource setting. This will minimize the effort for future maintenance work if the string token needs changed.

5.3.5 Ensure that the code does not contain "magic numbers"

A "magic" number is the direct usage of a number in code. "Magic numbers" should be refactored into constants so they can be easily changed in the future. Name the constant for the type of value being stored, not the actual value.

An example of a magic number would be the boiling point of water. The code should not contain a test that looks like the following:

```
If (WaterTemp > 100)
```

Rather, the value should be stored in a constant and the constant used in the comparison.

```
const constBoilingPointCelcius as integer = 100
.
.
.
If (WaterTemp > constBoilingPointCelcius)
```

Another example of magic numbers is indexes used to define datagrid cells. Code such as "Cell(3)" is encountered. In this case it would be more beneficial to have the "3' defined as a constant with a name that explains the purpose of the cell. For example: If Cell(3) refers to "UserName" then it would be good to define an integer constant named CELL_USERNAME that is set to 3; then the code now looks like Cell(CELL_USERNAME). Someone looking at the code 6 months later will not need to spend time determining what the contents of the cell contain.

5.3.6 File system paths should not be hard-coded

Ensure that hard-coded paths are not used. File system paths should be broken out to constants or moved to resource/configuration files to minimize the effort for future changes.

5.3.7 Break user-facing messages out into resource files

There should be no user-facing messages (prompts, messages, warning etc.) that are hard-coded.

5.3.8 Use String.Format with resource settings to construct user-facing messages

Code should not utilize manual concatenation of strings to construct user-facing messages. String.Format should be used in conjunction with verbiage stored in resource files.

5.4 Comments

5.4.1 All classes, methods, enums etc. must have comment headers supplied

Comment headers must be supplied. Comment headers not only assist developers that must maintain the code, they also can be used in some scenarios to generate documentation via the use of 3rd party tools (such as Sandcastle). Comment headers must include the full name of the developer who created the code, the date the code was created, and a log of dates the code was revised and the developer who made the revision.

5.4.2 Ensure comment headers are appropriate

Check to ensure that the comment headers/comments provided actually describe the code/objects that are commented. Look for cut & paste scenarios in which comment headers are copied from another object and not subsequently updated for the code/objects to which they are applied.

5.4.3 Ensure critical sections of code are commented

Ensure that critical sections of code are commented. This is especially critical if the code is very complex or very brittle. This is very important if the actions being performed are non-obvious to a competent developer.

5.4.4 Ensure comment headers are updated to reflect subsequent code changes

Ensure comments related to change history are updated during subsequent code changes.

5.4.5 Ensure comments in code are updated to reflect subsequent code changes

Ensure any comments specific to sections of code are updated if the code that the comments describe are updated. For example: If a comment indicates that the maximum size of an array is 100 and subsequent changes to code increase the maximum size to 150 ensure that the comment indicating that the maximum array size is updated to match the change in the code.

5.5 Objects/Classes etc…

5.5.1 Ensure constructors are defined

Always define constructors for classes (and any other object that requires a constructor). Never allow the compiler/language to create your default constructor for you.

5.5.2 Ensure IDisposable is implemented when needed

Ensure IDisposable is implemented when necessary. Ensure that IDisposable is implemented in the proper manner.

5.5.3 Ensure that sensitive/critical classes are decorated with NotInheritable/Sealed etc. if appropriate

Review the code and determine if there are any classes that should not be inherited. If any such classes are identified check to see if they have been decorated with appropriate modifiers such as NotInheritable (Visual Basic) or Sealed (C#).

Examples of such classes include (but are not limited to):

- Classes in the data layer
- Classes the provide administrative functionality
- Classes that implement security functionality

5.6 Structuring Code/Dead Code

5.6.1 No dead code should exist

Ensure the code does not contain methods / objects / variables that are never used.

5.6.2 No commented-out code should exist

Ensure that commented-out code is not present. This includes individual lines of code as well as entire sections of code.

5.6.3 No unreachable code should exist

Ensure that all code is reachable. There should not be any sections of code that will not ever be executed in the logical flow of the code.

For example, the following block of VB.NET code contains a line of code (highlighted in yellow) that will never, ever be executed:

```
        Dim UserName As String = String.Empty

        'return an empty string or the Name property of the WindowsIdentity
object
        If (String.IsNullOrEmpty(ident.Name)) Then
            Return String.Empty
        Else
            Return ident.Name
        End If

        Return UserName
```

5.6.4 Use Regions to structure code

Ensure the code uses REGION(when available) to structure code into logically or functionality related sections. Examples of candidates for structuring in REGIONS include (but are not limited to):

- Properties
- Constructors/destructors
- Member variables
- Public methods
- Private methods
- Imports of non-managed code

5.6.5 No empty regions

Ensure that the code does not contain any REGIONs that do not contain anything. An example of an empty region in VB.NET is:

```
#Region " Properties "
#End Region
```

In this example a Region named "Properties" is defined but it does not contain anything. This could possibly lead a developer performing maintenance on the code to think that the code is incomplete….or that the previous developer meant to create properties but never got around to doing so.

5.6.6 Ensure Regions only contain code that functionally belongs in the region

For example: If there is a Region named "Public Methods" ensure that only public methods belong to the region. Do not put Private methods in the region name "Public Methods".

5.6.7 Ensure all functions definitions explicitly return a type

Do not allow the compiler to decide the type of what is returned from the function. Ensure that the type of the values returned from the function are explicitly defined.

Do NOT have the function return its own name. Always return an explicit type.

Example of an incorrectly defined function (VB.NET)

```
Private Function ThisFunctionDefinitionIsVeryBad()
    Dim userMessage As String = String.Empty

    'do some work
    Return userMessage

End Function
```

Example of a correctly defined function (correction highlighted in yellow)

```
    Private Function ThisFunctionDefinitionIsVeryBad() As String

        Dim userMessage As String = String.Empty

        'do some work
        Return userMessage
   End Function
```

5.6.8 Ensure access attributes/modifiers are used appropriately

Ensure that access modifiers (Public, Private, Friend, Protected Friend, internal etc.) are used appropriately. Ensure that access modifiers are used in a manner that accurately reflect the design intent and visibility requirements of the particular class, object, section of code etc.

5.6.9 Break values that may change out into configuration files (ex: log file names, timer values, etc…)

Do not hard-code values that may frequently change in application code. Move them out to configuration files and/or settings. Examples include (but are not limited to):

- Log file names
- Timer values
- E-mail addresses
- Flags used to enable/disable logging/e-mail functionality
- Environment values
- Cooper Tire Security Application Settings

5.6.10 Ensure all colors are defined with resource settings

This ensures that colors can be changed easily with no changes to code. Name the resource setting for the condition being represented, not the color being used. For example, if text for an error message is to be set to red, name the resource setting ColorErrorText, not ColorRed.

5.6.11 Ensure efficient "return" logic is used in code

Only have one place in a method that an explicit "return" call is made. This promotes easier to read and maintain code

5.6.12 Ensure order of evaluation topics are handled correctly

Do not allow poorly coded order of evaluation constructs to increase the likelihood of exceptions or unintended effects. For example: A check to see if a DataTable contains rows occurs before ensuring that the DataTable is not null is a poorly constructed order of evaluation.

5.7 Code Quality

5.7.1 Ensure that code (including markup, CSS etc.) does not contain/generate validation warnings

Validation warnings should not exist in code, markup, CSS, XML or other entities

Contact the Cooper Technical Lead for the specific version of CSS, XML etc. that code should be validated against.

5.8 Code Style

5.8.1 Ensure that very long method argument/parameters lists are not all defined on one line in the IDE.

Ensure that method argument/parameters lists do not force a developer to scroll to see the arguments/parameters. Break long argument/parameters lists into multiple lines to enhance readability.

Security

6.1 Ensure administration functionality is highly-secured

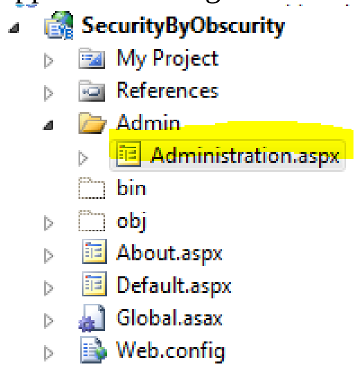
Any application / system that exposes administrative functionality must ensure that this functionality is tightly controlled. This frequently involves additional levels of security, frequently added to the application level, active directory level and database level where appropriate. NOTE: This also includes database code that supports administrative functionality.

6.2 No security by obscurity

Do not allow any code or application structure that implements security by obscurity. “Security by obscurity” essentially means that security is implemented by hiding something and hoping that it is not found.

Examples of this are frequently encountered in web applications in which Administrative folders and pages are deployed with the application but there are no visible links within the application to access the pages.

In the example below a folder named “Admin” has been added to the project. This folder contains a web page named “Administration.aspx”. If the developer hoped that just hiding the Adminstration.aspx web page in an Admin folder would protect it they would be very wrong. A user (possibly a potentially malicious user) could easily find this page just by manipulating the URL that was used to access the “non-admin” portion of the application and gain access to the Administration.aspx page.



This would then be exacerbated if no additional level of protection was used (such as protecting with HTTPS, requiring admin login, validating a user via active directory group membership, etc.)

6.3 Use secure coding methodologies

Always use secure coding methodologies when coding. Look for violations of Cooper’s secure coding standard. Many secure coding best practices are related to a simple concept: “Do not trust any input entered by a human”. Always validate user input. Validation includes ensuring required data was entered, entered in the correct format and does not contain any potential malicious input. Please note that a non-malicious user input could be very damaging in a poorly coded application. If an application takes user input and directly executes it as part of a SQL command in a database the potential exists to do much damage (in this case unintentional). A poorly coded application could be very dangerous to a user with malicious intent.

6.3.1 Do not expose stack dumps to clients

Exception stack traces are extremely valuable to a developer. This information can be used to quickly find and resolve the issue that caused the exception.
That being said, it is important to ensure that stack trace information is not displayed to the client. This is especially true for applications that are exposed outside of Cooper’s firewall (such as web applications). Exposing stack trace information to clients and/or outside consumers can increase the attach surface that a malicious user could then leverage in attempts to compromise a Cooper system.

Refer to Cooper's secure coding standard for more information on exposing stack traces to clients.

6.3.2 Ensure appropriate fields lengths used in Windows and web forms

Ensure that controls used to accept user input are configured to only allow the maximum number of characters required to fulfill the business logic/requirements.

In Visual Studio .NET 2010 the default maximum length of a textbox in a Windows forms app is 32767 characters. Default textbox length in a web form is also significantly large. In most cases default values as large as those mentioned above are not required for our applications.

Failure to set (and enforce) maximum lengths can cause exceptions in applications. Failure to set (and enforce) maximum lengths can also make the applications more vulnerable to malicious actions.

6.3.3 Crystal Reports (or other reports) should not contain hard-coded database connection information

Do not embed database connection information into Crystal Reports or other reporting solutions. Embedding database connection information into reports creates additional maintenance tasks when the underlying database information (username, password, location, server, etc.) changes. Embedding database connection information into reports also exposes this information in a manner that is not acceptable.

Most robust reporting solutions support the ability to bind datasets and/or data tables to the reports at runtime. These reporting solutions can also consume .xml for the purpose of designing the reports. In a Visual Studio .NET environment, strongly-typed datasets can be used to both design the report while it is disconnected from the database as well as bind the data to the report at runtime.

6.3.4 Is Cooper Tire Security (CTS) being used in the application? If not, should it be used?

If the system/application was built with Visual Studio .NET and interacts with an Oracle database is Cooper Tire Security being used? If it is not determine why it is not being used.

NOTE: It has been proven/demonstrated the CTS can be consumed from within VBA code. Therefore CTS can be used with Excel and other applications that use VBA. Therefore Excel and other VBA applications should be reviewed with this in mind.

6.3.5 Is CooperTire Security (CTS) being used correctly in the application?

If Cooper Tire Security is being used in the system/application ensure that it is being used correctly. Contact a Cooper architect if you have questions as to whether CTS is being used correctly and/or efficiently.

Exception Handling

7.1 Appropriate use of last-chance exceptions

Ensure that “last-chance” exception-handling strategies and functionality are used. Most (but unfortunately not all) applications implement some type of exception handling. Many developers are very good at handling typical or expected exceptions. That being said, there are some type of exceptions that cannot always be anticipated or handled in some places in code. These are frequently referred to as “last-chance” exceptions.

Visual Studio .NET provides several tools/methodologies to handle some of these last-chance exceptions.

Web Applications
Last-chance exception handling can be implemented in web applications via use of global.asax in both Visual Basic and C#. Ensure that web applications implement code in the **Application_Error** method in the global.asax (and the corresponding code-behind files) to handle these unexpected/unhandled exceptions.

Windows Form Applications
Last-chance exception handling can be implemented in Windows forms applications via use of the application framework/application events tab in the project properties in both Visual Basic and C#. Ensure that these applications implement code in the **UnhandledException** handler method to handle these unexpected/unhandled exceptions.

NOTE: This handler is implemented differently in Visual Basic than it is in C#.

7.2 Use try/catch/finally where appropriate

Ensure that Try/Catch/Finally blocks are used where needed. Ensure that Try/Catch/Finally blocks are not used in scenarios where they are not needed. Note that this can vary between systems and even between layers within the system. Always use Try/Catch/Finally blocks when opening connections with databases. This provides the ability to close the database connection within the Finally block when an exception occurs.

7.3 Return meaningful exceptions

If exceptions are being generated in code ensure that they are meaningful and provide some useful information that will help in troubleshooting the code. For example, the following code was found (and flagged) in a recent code review:

```
throw new Exception()
```

This is close to useless. Other than the stack trace that would be generated when the exception is thrown it provides no meaningful information that will help a maintenance developer troubleshoot the exception.

7.4 Do not have empty Catch blocks or empty Finally blocks

Empty Catch blocks and empty Finally blocks are not allowed.

7.5 Use of the System.Diagnostics.Debug.* methods cannot be the only code that exists in a Catch block

Since System.Diagnostics.Debug.* methods are not compiled into Release builds, any such code will not exist in the final assembly when compiled for QA or Release. Such code can exist to facilitate debugging however it cannot be the only code in a Catch block.

E-mail

E-mail functionality exists in many applications/systems. The following items should be checked during a code review

8.1 No hard-coded e-mail addresses in application code

E-mail addresses (including distribution list names) should not be defined in code/objects that must be compiled.

E-mail addresses (including distribution list names) should be defined in configuration files and/or settings to allow them to be changed without re-compilation of code/objects.

8.2 No hard-coded SMTP server names

SMTP server names should not be defined in code/objects that must be compiled.

SMTP server names should be defined in configuration files and/or settings to allow them to be changed without re-compilation of code/objects.

8.3 SMTP server IP Addresses Should not be used

SMTP server IP addresses should never be used in code/objects/configuration files unless absolutely necessary.

SMTP server names should be used instead of server IP addresses. The SMTP server name should be defined in configuration files and/or settings to allow them to be changed without re-compilation of code/objects.

8.4 No hard-coded Subject Verbiage

E-mail subject verbiage should not be defined in code/objects that must be compiled.

E-mail subject verbiage should be defined in resource files to allow them to be changed without re-compilation of the main application code/objects. Resource files also make it easier to support multiple languages with no code changes.

8.5 No hard-coded Message Body verbiage

E-mail message body verbiage should not be defined in code/objects that must be compiled.

E-mail message body verbiage should be defined in resource files to allow them to be changed without re-compilation of the main application code/objects. Resource files also make it easier to support multiple languages with no code changes.

8.6 Review SMTP settings with a Cooper architect

SMTP server settings should be reviewed by a Cooper architect to wnsure they are current.

General Coding Topics

9.1 Look for "Copy & Paste" Code

Code re-use can be very beneficial and can save time when working on projects. That being said, several occurrences of copy & paste code have been encountered where code was copied but not updated to reflect the context of the current project. Examples of some things to look for include (but are not limited to):

- Pasting in code but not changing variable names
  - This can make maintaining code difficult and/or confusing for the maintenance developer.
    - Example: A variable named Employee Number that is used for a machine ID or machine part number
- Pasting in code but not changing values
  - Examples include pasting in an OracleParameter but not changing the OracleType and/or size to match the requirements of the current code
- Pasting in code retrieved from another project but not reviewing if the code is correct and/or efficient
- Pasting in code retrieved from a search of the Web but not checking to see if the code is correct, efficient and/or safe
- Pasting in code retrieved from a search of the Web but not checking to see if the code is copyrighted to another company

- Pasting in “open source” code without reviewing the open source license. Not all “open source” licenses allow use in a corporate environment

9.2 Look for empty or missing conditional statements/blocks

- Ensure that all conditional logic blocks are complete
- Look for if/then blocks where nothing happens in the "then" block
- Look for if/then/else blocks in which nothing happens in the "then" or "else" block

9.3 Look for inefficient code

Review the code to identify inefficient coding. Making code work is one thing, making code work efficiently (or avoiding inefficiencies) is optimal.

Examples of inefficient coding include (but are not limited to):

- Poor looping logic
- Iterating through entire dictionaries instead of retrieving items by the dictionary key
- Inefficient string handling/concatenation

9.4 Look for non-resolved TODOs

Microsoft Visual Studio .NET (and other tools) support tags/labels for identifying sections of code that require additional follow-up tasks. In Visual Studio .NET the following tags are supplied out-of-the-box:

- TODO
- HACK
- UNDONE
- UnresolvedMergeConflict

In addition to the out-of-the-box tags/labels developers have the ability to create their own tags/labels.

During the code review look for any of these tags/labels that still exist in the code. A "TODO" tag/label (or any other label) that is still in the code should be flagged for additional review and follow-up.

9.5 Look for comments indicating that a section of code is a hack/kludge etc.

Look for comments and/or tags that indicate that a section of code is a hack, kludge or work-in-progress. If a comment such as "This barely works and needs cleaned-up and made more efficient" is found, then it should be flagged during the review. The same applies for comments such as "This should not work but it does" or "Hard-coded to keep moving forward with development", etc.

9.6 Are critical parameters being validated?

If a method / stored procedure accepts parameters that are critical, are they being validated? Validation includes (but is not limited to):

- Ensuring the parameter is not null or empty
- Ensuring the parameter contains a value within an acceptable range
- Use of Debug.Assert along with a test condition is acceptable only in the following scenarios:
  - It is used in non-Release configurations (in other words, only allowed in debug configurations)
  - It is used in nUnit and/or test projects

9.7 No assumptions are made when working with data in DataSets and DataTables

Ensure that the code does not contain implicit assumptions that data is always present in a DataSet and/or DataTable. Frequently code is found in which a DataSet is returned from a call to a database and then the next line of code is an attempt to retrieve something from .Table(0) or .Table[0]. What happens if no data was returned and the DataSet does not contain any tables? The same applies to always assuming that a DataTable contains at least one Row.

9.8 Look for hard-coded retrieval syntax in Collections and other objects

Review the code to identify any areas in which collections are accessed using a hard-coded index. If these are found determine if there is a more efficient, more easily readable option.

For example: If Item(0) is used in the code, it is more readable and easy to understand to use Item("SAP_PART_NUM"). This not only tells the developer what the value of the Item is it also serves as a visual indication that if a subsequent change is made to the field named SAP_PART_NUM that it will impact the code that is using it.

Another example was recently found while troubleshooting an issue. A link was being created on a web page based on the results of a call to a database. Multiple records were returned from the call to the database, however the code creating the link was only retrieving the first record that was returned. The code was supposed to determine which of the multiple records returned was the correct record and build the link using the correct record. Instead the code always used the first record which resulted in incorrect links being created in some scenarios.

9.9 Look for the appropriate use of properties (or lack thereof)

Review the code to determine if properties are being used correctly. If properties are not being used, should they be used? Ensure that a property that is meant to be a read-only property is truly read-only and cannot be updated. Ensure that an updateable property can be updated. Ensure that the scope/visibility of properties is appropriate as per design intent. Getters and setters should be fully implemented. Do not use default implementations of getters and setters.

9.10 GOTO Statements are Not Used

Ensure that code does not include "GOTO". In most modern languages this type of syntax / logic is very poor coding technique and can make troubleshooting/ maintaining the code extremely painful.

That being said, in some very old, out-of-date/out-of-support languages (such as VB6 or COBOL) a GOTO statement may be the only option (and / or best practice).

9.10.1 Ensure the length of methods, classes etc. is appropriate

Ensure that the number of lines of code in classes, methods, functions, stored procedures, etc. is not too large. If any of the above contain many lines of code, then that may be a very strong indicator that the code is a candidate for refactoring and or improvement.

Database/Data Layer

10.1 No hard-coded connection strings

Connection strings should never be hard-coded in application code. If Cooper Tire Security (CTS) is being used it will handle all connection string activities. If Cooper Tire Security is not being used then connection strings should be stored in an encrypted format in the configuration file used by the application/system. Always use Cooper-created encryption/decryption utilities to encrypt/decrypt the connection strings (see 10.3 below)

10.2 No plaintext connection strings

Never store connection strings in plaintext…anywhere. If Cooper Tire Security (CTS) is being used it will handle all connection string activities. If Cooper Tire Security is not being used then connection strings should be stored in an encrypted format in the configuration file used by the application/system. Always use Cooper-created encryption/decryption utilities to encrypt/decrypt the connection strings (see 10.3 below)

10.3 Do not create custom encryption/decryption code

Never create custom encryption/decryption code in an application (unless approval has been given by a Cooper Application Architect). Always use the Cooper-created encryption/decryption utilities that are stored in Serena and Team Foundation Server.

10.4 Ensure security bypass code does not exist in the system

Any code and/or configurations implemented to bypass security are not acceptable.

Exception

The only exception to this rule is when a Cooper application architect has given permission to bypass Cooper Tire Security (CTS) code (in DEVELOPMENT environments only) in order to facilitate development by partners who are developing on workstations that are not members of the Cooper global.ctb domain.

In this scenario the developer must:

- Use the Cooper-supplied "starter" solution that contains Cooper-approved CTS bypass methodology.
- This is only allowed in the development environment.
- All bypass code and configuration settings must be removed prior to deployment to QA (or production if no QA environment exists).
  - Code and configuration settings must be removed – commenting them out is not acceptable.

10.5 Ensure connections are closed in the Finally block of a Try/Catch/Finally construct

Database connections should always be closed within the Finally portion of a try/catch/finally block. This ensures that the connection will be closed even if an exception occurs after the database connection has been opened.

10.6 Ensure usernames and/or passwords are not hard-coded in application code

Usernames and/or passwords should not be stored in application code. If code requires a username and/or password to be specified, the username and/or password should be stored in the application configuration file.

In addition to the above, any passwords stored in the application configuration file should be encrypted using the approved, Cooper-created encryption/decryption utilities. Refer to section 10.3 for more info.

10.7 Protect Against SQL Injection

Ensure all code is written in a manner the prevents SQL Injection attacks. For example, never take user input (or anything that can be manipulated by a user …such as a URL) and use it as a direct command to a database call. Refer to Cooper's Secure Coding Standards for additional information.

10.8 Ensure Code conforms to Secure Coding Standards/Methodologies

Ensure the code conforms to the requirements outlined in the Cooper Secure Coding Standards document.

10.9 Ensure Oracle Stored Procedures and Functions are Structured into Packages Based on Functionality

Ensure that Oracle procedures and functions are grouped into packages to correctly reflect the various functional roles that the code supports. Do not "mix-and-match" different types of functionality in the same package. For example:

- Do not have update commands in a QUERY package
- Do not have delete commands in a QUERY package
- Do not have code that performs Administrator-level functionality exposed in a package that will be accessed by a non-Administrator level user/method

10.10 Look for execute immediates that may be dangerous or not needed

The use of EXECUTE IMMEDIATE in database code can be very dangerous. Therefore ensure that the use of EXECUTE IMMEDIATE is justified and that appropriate measures have been taken to prevent any malicious code from being executed in the EXECUTE IMMEDIATE statement.

10.10.1 Ensure Database Schemas have Appropriate Privileges

Check permissions on database schemas (and objects contained within the schemas) to ensure that unusual or unnecessary privileges have not been granted to other schemas and/or users. For example, it would be highly-unusual for a data schema to directly update a table in a procs schema.

10.11 Are Oracle Parameters defined to reference the underlying type of the target field in the table they reference using the %TYPE syntax?

Oracle provides the ability to link the type of a parameter in PL/SQL to the underlying type of a field in a table. This is called "anchoring". This tightly binds the PL/SQL parameter to the type of the field that it interacts with. This provides several benefits. Examples include (but are not limited to):

- Any changes to the data type of the table field may break the PL/SQL code that references it if %TYPE is not used. While this sound bad it is actually helpful. This ensures that table fields are not changed that could impact the PL/SQL at runtime. It is better to have your code break while coding than have it break while a client is executing it.

10.12 Is SELECT INTO --- WHEN_NO_DATA_FOUND handled?

Review code to ensure that any PL/SQL code that uses SELECT INTO a variable has been coded to handle the scenario when no data is found. Ensure that the WHEN_NO_DATA_FOUND exception is handled (or prevented from occurring in the first place).

10.13 No INLINE SQL

SQL syntax should not be defined in application code. There are very, very few cases in which this is acceptable. SQL code should always be defined in the database using stored procedures, functions or packages (packages exist only in Oracle at this time).

10.14 Ensure Cooper Stored Procedure Development Standard is Followed

Ensure the Stored Procedure Development standard is followed.

10.15 Ensure Database Object Naming Standards are Followed

Ensure database objects conform to the Oracle Object Naming Standard.

10.16 Ensure Database Code that References Data in other Database instances follows standards

It is fairly common for database code to interact with data in schemas defined in other instances/schemas. In those scenarios a standard has been defined that specifies how such data will be made available to the consuming code. Refer to the Oracle Interface Standard and ensure that the database code/structure conforms to this standard.

10.17 Always use the appropriate pragma settings in C# to disable decprecation warnings related to use of the System.Data.OracleClient (and other System.Data.Oracle*) Namespaces

Required warnings to disable at the beginning of a class and to restore at the end of a class are: 612, 618

NOTE: Use of the above pragma settings will disable these warnings from being generated when building C# code that uses the System.Data.Oracle* namespaces

10.18 In VS 2013 and later, always set the NoWarn element in the .vbproj file in VB appropriately to disable decprecation warnings related to use of the System.Data.OracleClient (and other System.Data.Oracle*) Namespaces

In VB code from version 2010 and prior, deprecation warnings for use of System.Data.Oracle* namepaces are allowed due to the inability to suppress those warnings via the use of pragma directives.

In VB code from version 2013 and later, follow the instructions at the following link to suppress deprecation warnings for the use of System.Data.Oracle.* namespaces.

https://docs.microsoft.com/en-us/visualstudio/ide/how-to-suppress-compiler-warnings?view=vs-2015#suppressing-warnings-for-visual-basic

Web Application Configuration

11.1 Ensure recommended settings to enhance security are included in web.config files

The Cooper Secure Coding standards define common web.config settings that should be added to each web.config file. These settings provided enhanced security and must always be included in web.config files

11.2 QA and Production web.config settings <u>must</u> include the following settings

- customErrors must be set to “RemoteOnly”
- Debug must be set to “false”
- Trace enbled must be set to “false”

Potential Concerns during a Code Review

When being brought in to perform a code review there are a few items that may become apparent that could be a cause for concern. A few of these items are discussed below. Do not read too much into what is discussed below….just note that several of these items have already been encountered during code reviews.

12.1 The same issues are encountered repeatedly

If the same issues are flagged over and over again in code created/updated by the same developer or development team then there is an obvious issue that needs addressed. This could be due to one or more of the following:

- A need for additional training
- Conflicting project scope
- A basic failure to learn from past mistakes
- Sacrificing code quality to meet a deadline

12.2 Lack of understanding of why an issue was flagged

If, after completing the code review and returning the findings to the developer or development team they do not understand why something was flagged in the review, this could be an issue.

Asking questions about why something was flagged is acceptable in some scenarios. Following-up with the review team to explain why something was done a certain way in the code is also acceptable.

That being said, obvious lack of understanding of solid coding technique is not acceptable. For example, if a developer does not understand why database connections should be closed in a Finally block or why OPTION STRICT should be required or why all compiler warnings should be investigated or why accepting default web service URIs is <u>not</u> good programming, then there are more serious issues which need resolved that are outside the scope of the code review.

12.3 Simple Changes to Enhance Overall Code Quality/Stability are deferred to Later Phases of the Project

In some cases changes to enhance the overall code quality are in-scope for the project and there is an extremely high probability that these changes take place. In other cases these types of changes are deferred until later phases of the project and one may get the impression that there is very little chance of these later phases actually occurring. Code quality/stability should never be sacrificed for the sake of meeting a schedule.

12.4 Attacking the Code Review Process

The purpose of the code review process is to improve code quality (and hence stability for our clients). In the world of increasingly short timelines, constrained resources and varying skillsets, it could be easy to attack the code review process itself as the cause for missed deadlines, cost overruns, missed features, etc.

We must all be aware that the code review process is still evolving and being enhanced as we perform more reviews and gather feedback from development and code review teams.

It <u>is</u> acceptable to suggest changes to the code review process to make it more efficient, comprehensive and beneficial to creating/maintaining high-quality code . It is <u>not</u> acceptable to attack the code review process itself if the root cause of problems is not following guidelines/standards and/or project timelines forcing the acceptance of poor quality/non-compliant code.

12.5 Not allocating sufficient time for code reviews in the Project Planning Process

The project planning process MUST include sufficient time for code reviews. Project managers/leads do NOT have a clear understanding of how long a code review takes. The definition of "sufficient time" will be provided by the code review team.

12.6 Not allocating sufficient time for the development team to address issues identified during code reviews in the Project Planning Process

The project planning process MUST include sufficient time for the development team to address issues identified during code reviews. The definition of "sufficient time" will be provided by the code review team and development team. Project managers/leads do NOT have a clear understanding of how long it takes to address findings identified during a code review.

12.7 Using time allocated for Code Reviews to make up for poor planning/estimation

It is becoming an unfortunate "standard practice" to use time allocated for code reviews to allow the development team to catch-up on tasks that they are running behind schedule on (due to scope creep, poor planning/estimation or other reasons). When this happens it is <u>extremely rare</u> for the code review team to be given the amount of time that they require for the code review. They are usually told to "hurry-up" or that they are a "roadblock" to a real or artificial timeline.

Forward

13.1 Ensure that new information identified in reviews is added to this document and is included in the next code review cycle

If we identify and accept the same types of items over and over again then there are issues that need to be resolved. These issues could be related to training, process, and/or other factors outside the control of the code reviewers.

13.2 Set consistent expectations!

Set the expectation at the beginning of coding that the code will be held to these standards. Do not be willing to accept "it is OK to ignore code quality this time" on a project unless there are extraordinary circumstances. The same applies to an "out-of-scope" justification. Code quality should never be out-of-scope.

13.3 Solicit and Encourage Responses from the Coding Team

There may actually be a valid reason why one or more of the items outlined in this document was violated. Discuss the review findings with the team. Try to get an understanding of why something was coded in a certain manner.

References

The following documents are either referenced above and/or would be beneficial to review for further information and detailed examples.

Document	Link
Cooper Tire Secure Coding Standards	Secure Coding Standards
Oracle Object Naming Standards	Oracle Object Naming Standards
TOAD Oracle PL/SQL Format Standard	PL/SQL Format Standard
Oracle Stored Procedure Development Standard	Oracle Stored Procedure Development Standard
Oracle Exception Handling Standard	Oracle Exception Handling Standard
Oracle Interface Standard	Oracle Interface Standard