SECTION 1 - SCOPE

SCOPE

The scope of this document is to identify .NET coding standards and the code review process that will be followed for all .NET applications developed or maintained by staff or consultants employed by the Office of Information Technology.

Best practices are also included in this document for developers to incorporate in their code.

Updates to this document are made on an as-needed basis by the Application Services Standards and Technical Work Group upon approval by Application Services Management.

SECTION 2 - THE REVIEW PROCESS

WHAT IS A CODE REVIEW AND WHO PERFORMS IT?

A code review is a methodical review of .NET code against the established OIT .NET Coding Standards (located in Section 3 of this document). Two types of .NET code reviews are referenced in this document: Informal and Formal.

The goal of informal code review(s) is to locate and expedite any concerns or questions to the entire project team as early in the coding process as possible. This enables a pro-active approach to resolving potential problems as well as providing knowledge sharing. The informal code review is conducted internally by .NET skilled developers able to provide an objective review of the code. In addition, informal code review(s) must include a review of the SQL by the database technician assigned to the project. The informal reviewer(s) are coordinated by the project manager.

The goal of the formal code review is to provide a final check of the code prior to production implementation to ensure that violation of standards has not been overlooked. The formal code review is conducted by a team of experienced .NET developers provided by the FDOT Enterprise Library Team . The review is documented and results stored in the Code Review Library. Project staff are notified once the review is complete.

WHY IS A CODE REVIEW PERFORMED?

A code review is performed to protect the shared runtime environment from inefficient and unsafe code execution. It provides an improvement in the overall quality and consistency of coding practices as well as development of best coding practices whereby others can benefit. Reviewing the application functionality, behavior, workflow, user interface, or any other application design issues is not part of the review.

WHICH PROJECTS REQUIRE A CODE REVIEW?

All new development requires a .NET Code Review. Additionally, any application that has a maintenance or enhancement release on the Application Services WorkPlan requires a .NET Code Review.

WHEN ARE CODE REVIEWS PERFORMED?

Informal code review(s) must be done periodically during the application life cycles. They should be performed early enough so that feedback provided to the programmer can be incorporated in their future assignments. Informal code reviews should be an iterative process, not fixed to a specific point in the development or maintenance lifecycle.

The Formal Code Review is conducted and documented at the end of the construction / maintenance phases and before final user acceptance.

The project team must build time into the project schedule for the informal and formal .NET code reviews. Formal Code Reviews should allow for a 5 – 10 business day turnaround of the results. All projects must be aware of the time limitations for the Formal Code Review and provide advance notification to allow for its scheduling. Items uncovered in the Formal Code Review must be resolved before production implementation.

THE FORMAL CODE REVIEW

1. The request for a Formal Code Review must be made two weeks prior to when the code will be available for review (at code freeze). The Code Review process takes 5 – 10 days to complete.
2. Formal Code Reviews must be requested by the BSSO Program Manager or Project Manager (with BSSO Program Manager approval).
3. An application cannot move to production if there is a pending code review.
4. The formal code review must be requested by submitting an online form. There are two forms available: one for BSSO Development Staff and one for End User Development Staff: The following details must be submitted via the online form:
   - 4.1. Application Name
   - 4.2. A labeled version of the Source Code which includes the location and version number.
   - 4.3. Instructions and all necessary software and details to create a buildable solution. Have a developer verify that a solution can be downloaded and built, with no errors, using the information provided, prior to requesting the Code Review.
   - 4.4. Version of Visual Studio on which the solution was created.
   - 4.5. Code Freeze Date (Anticipated Date to Start Review).
   - 4.6. Database Platform
   - 4.7. Internet or Intranet
   - 4.8. Web Application, Client, Batch Processing program or a combination of these.
   - 4.9. Authentication Platform.
   - 4.10. IRR Numbers for any Third Party code or assemblies that is included in the application. This would include all code or assemblies directly included in the application itself, as well as any code or assemblies that may be provided via a FDOT built common assembly.
   - 4.11. Contact person for questions and answers prior to the review.
   - 4.12. DA and DBAT Contacts.
   - 4.13. Results of SQL Review.
   - 4.14. Any exception requests that have been granted to the Project (if applicable).
   - 4.15. Additional supporting documentation for the application to facilitate timeliness of the review and to understand the code.
5. Violations identified during a code review must be addressed prior to production implementation. The following must be provided to the review team for re-review.
   - 5.1. Source Code Location and version information, instructions and all necessary software and details to create a buildable solution.

6. It is the responsibility of the Program Manager to notify the Review Team of any changes made after the Code Review has begun, including a summary of any significant changes. The Review Team will identify if an additional Code Review is required.

SECTION 3 - .NET CODING STANDARDS

1. Calling the Garbage Collector is not allowed.

SUPPORTING DETAILS: Calling the garbage collector explicitly is a very resource intensive procedure. It does not need to be called explicitly because it is run by the Common Language Runtime (CLR) during periods of low activity and memory pressure.

2. All objects that implement the IDisposable interface or inherits from a class that implements this interface must have their dispose method called either in a try finally block or by using a 'using' block when the dispose method is not automatically called and where NOT calling Dispose results in a Resource Leak.

SUPPORTING DETAILS: Objects that implement IDisposable typically use unmanaged resources that may not be appropriately cleaned up when the object goes out of scope resulting in a memory leak. This is particularly important for database connections, file streams and socket connections because you could end up locking the resource. Certain objects such as the Data Table do not leak resources if not disposed and do not need to call their Dispose method. If you are in doubt as to whether a particular object needs to be disposed, it is better to err on the side of caution and call the dispose method. Inspecting the disassembled source through Reflector allows verification of the effect of not calling the Dispose.

The easiest way to tell is to wrap the object in question in a *using()* call; Visual Studio will let you know if it doesn’t implement IDisposable.

3. Overriding default settings for the .NET Request.Form validation is not allowed.

SUPPORTING DETAILS: Disabling ASP.NET Request Validation allows potential cross site scripting attacks. IIS will not validate the content of the http request header for any potentially malicious tags or scripts.

4. SQL statements must be implemented using ADO .Net parameterized query objects.

SUPPORTING DETAILS: Parameterized SQL statements are important for both security and application performance. The use of ADO.Net parameterized queries is the standard. Adherence is required whether using handcrafted queries or an ORM tool.

Security:
Parameterized queries are the best defense against SQL injection attacks because they prevent the context of the statement from being changed. In typical SQL injection attacks, string terminators and SQL keywords are entered which if concatenated with a dynamic SQL query cause the execution of SQL code not intended by the developer.

Performance:
Parameterized queries give a significant performance boost to most database operations because compilation by the DBMS is on first execution only. All subsequent calls use the cached execution path until it is removed from the cache. This works because the SQL

statement itself is not changing compared to a completely dynamic statement with embedded values.

5. General Shared Resource Addresses should not be hard coded.

SUPPORTING DETAILS: Hard coding shared resource addresses requires the code to be changed when promoting from Unit Test, through System Test and on to Production. Additionally, system configuration changes cannot be implemented seamlessly when hard coded references are used.

Examples of Shared Resources include, but are not limited to:

- SMTP - available from the FDOT Enterprise Library
- DB2Creator - available through the connection value or the environment variables
- Absolute URLs, use the relative path instead.
- IP addresses
- Server Names

6. Turn Option Strict On for VB.NET (disallows late binding).

6.1. For Projects this must be enabled at the Project Level.
6.2. For Websites this must be set in the root config file on the website.

SUPPORTING DETAILS: Visual Basic allows conversions of many data types to other data types. Data loss can occur when the value of one data type is converted to a data type with less precision or smaller capacity. A run-time error occurs if such a narrowing conversion fails. Option Strict ensures compile-time notification of these narrowing conversions so they can be avoided. In addition to disallowing implicit narrowing conversions, Option Strict generates an error for late binding. An object is late bound when it is assigned to a variable that is declared to be of type Object. Because Option Strict On provides strong typing, prevents unintended type conversions with data loss, disallows late binding, and improves performance, its use is required.

Source: http://msdn2.microsoft.com/en-us/library/zcd4xwzs.aspx

7. The VB.NET Methods and Keywords listed below must not be used because they are redundant, deprecated or potential performance degraders. Use one of the listed alternate constructs instead.

SUPPORTING DETAILS: The use of CLR methods and constructs will prevent the use of VB deprecated code which will not be/is not supported in the future and better supports object-oriented programming (OOP). For more information see http://msdn.microsoft.com/en-us/library/skw8dhdd.aspx.

Outdated VB6 Keyword/Method	**VB.NET Preferred Alternate**
Call	Method calls no longer require the use of the Call keyword.
Err	Use the Exception object or a derivative
iif	This causes unnecessary overhead due to boxing/un-boxing. Use standard If…Then…Else blocks with the short-circuited Boolean operators (OrElse and AndAlso)
GoTo	Refactor your code to use one of the more standard control flow statements (if, select case etc.)
On Error GoTo / Resume	Use Try…Catch…Finally blocks
Randomize	Random Object is automatically randomized when declared
EndIf	Use 'End If' to terminate the 'If Then Else' Block
Variant	Using the 'variant' equivalent, 'object' type is not recommended as it carries a performance cost while boxing and unboxing. Either use type specific methods or use generics.
GoSub	Call procedures with the **Call** statement, and the **GoSub** statement is not supported. Note: As mentioned in the 'Call' section, you can directly call the method with the name without the keyword
Wend	Use **While ... End.**
Let	No alternative. The meaning of the **Let** keyword has changed. **Let** is now used in LINQ queries. 'Let' Computes a value and assigns it to a new variable within the query.

8. Include comments in the code.

SUPPORTING DETAILS: These comments should explain why the code was written in the manner that it was, and should make it easier for programmers working on maintenance in the future. Use these comments to document anything that you think will help future programmers understand the coding method/logic that was used.

9. When using HIS and DB2 stored procedures defined with 'commit on return no', the .NET code must wrap each call to a procedure (or group of calls to a procedure) in a transaction.

SUPPORTING DETAILS: Not wrapping DB2 non-committing stored procedures in a transaction will result in DB2 perpetually locking the resources used in the call. DB2 will eventually close the connection without releasing the lock, and this will lead to the MSDB2 client driver becoming unstable and unrecoverable. This state will force a process re-start to correct. This will allow the transaction to control the unit of work, insuring data integrity, avoiding the risk of partial units of work being committed, and minimizing DB2 locking.

10. Exceptions must be handled appropriately by the project.

10.1. Exceptions must not be swallowed.

10.2. Only catch expected exceptions for the purpose of handling them.

10.2.1. Only mask an exception that is handled within the application.

10.3. All other exceptions must be allowed to bubble-up to the correct tier.

SUPPORTING DETAILS: Exception Handling is critical to the project as well as to the environment.

- Swallowing (ignoring) or masking (replacing) unexpected exceptions will result in lost information. Allow the exception information to bubble-up unchanged, so that complete information is available to support staff.
- The syntax throw works better than throw ex
- Preserve the stack trace information where appropriate.

11. Suppress the display of exception details to the user.

11.1. The web.config file must contain a configuration entry that specifies the use of error pages.

11.1.1. Applications must use the error handling of the FDOT Entperise Library for web applications.

11.1.2. Applications must use the <custom errors, sections to configure the error page and set the MODE property to either '**ON**' or '**Remote Only**'.

SUPPORTING DETAILS: Suppressing error details in an application is a vital step in implementing a comprehensive strategy to prevent a wide variety of malicious attacks. Error messages that reveal details about the application's business objects or database structure can be exploited by malicious users to launch an attack against the application. In web applications, suppressing the exception details can be achieved by using custom error pages.

- Customized, user friendly error pages or messages should always be used to hide information that can aide an attacker seeking to exploit vulnerabilities in our applications.

- Where possible, application errors and exceptions with their associated stack trace should be emailed to the appropriate groups for troubleshooting the underlying problem.

12. Applications must be developed such that there is at least logical separation between presentation logic, business logic and persistence logic.

12.1. The business logic must encapusulate the business rules, state and interactions inherent to the problem domain.

12.2. The presentation logic must handle view rendering, input collection and transformation as necessary before passing data to the business object classes for processing.

12.3. The persistence logic must handle system state persistence and retrieval including transformation necessary to recreate business logic state.

12.4. Additional layers of separation are permissible for additional separation of concerns as well as for coordination and communication between layers.Exceptions must be handled appropriately by the project.

SUPPORTING DETAILS: This standard sets out a bare minimum requirement for separating application concerns to improve maintainability of the system.

Outside of just separating UI/Business logic/persistence logic applications tend to deal with various concerns such as user notifications, document storage and retrieval, workflow management, etc. which are all logically separate and should be identifiable as separate areas of concern for the system. For example, a single logical operation may require a status change on an entity, the storage of a document and an email notification to a user that the action was performed. This should not be implemented as one method that has all the logic to perform these 3 separate tasks but rather there should be a coordination between separate testable areas of functionality within the application to perform the operation.

We strongly discourage code that is not testable in isolation from other dependencies i.e. static classes, code that has side effects within the system, code that is tightly coupled to other code/classes, etc.

We highly recommend developers adhere to the SOLID design principles as much as possible. The following blog article series goes into detail regarding applying the SOLID principles: http://lostechies.com/jimmybogard/2008/06/17/separation-of-concerns-how-not-to-do-it/

The current preferred layer segregation model for development at FDOT is the Onion architecture as described by Jeffery Palermo.

13. Web Applications using client-side storage must not store sensitive information or information which could compromise the application's security.

SUPPORTING DETAILS: Sensitive information includes, but is not limited to, authentication credentials, authorization values (e.g. group membership), personally identifying information that could be exploited for tracking purposes, session identifiers, or data that could be misused in a harmful way. Examples of personally identifying information include name, social security number, address, and telephone number (see http://en.wikipedia.org/wiki/Personally_identifiable_information for a more complete discussion). Note that session identifiers are permitted in cookies, which are already commonly used for that purpose and have a means for mitigating the risk of exploitation. Please refer to https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet#Storage_APIs for more information on the potential vulnerabilities of storage APIs.

Examples of appropriate use might include saving user preferences, preserving UI state, caching a history of previous searches, caching query results so long as the cached results do not expose anything that might compromise the application's security, and supporting partially offline usage scenarios.

14. Email Messaging

14.1. Email addresses must be fully qualified.
14.2. An individual's email address must not be hard coded.
14.3. Email Addressing
 14.3.1. Emails generated using the FDOT email domain must be an existing address in the Department's email system.
 14.3.2. Email addresses generated from User Information or user entry must be validated for format (contains @, etc.) but are not required to be a valid email address.
14.4. Application-generated emails that may handle responses from the recipient must be produced using a valid "From" and/or "Reply To" address.
14.5. Application-generated emails that do not handle responses from the recipient:
 14.5.1. Must be produced using the reply to address of "**DoNotReply-FDOTApp@dot.state.fl.us**".
 14.5.2. Must include a warning to the recipient that the email should not be "replied to" and any responses will not be monitored.

SUPPORTING DETAILS: It is important that any email generated from an enterprise application use a valid and active email address in the “From” or “Reply To” fields.

If the email system cannot deliver the message (for whatever reason), and the “From” or “Reply To” fields are invalid, the delivery failure message has no valid sender address to go back to. This generates unneeded traffic on the email system as the system continues to try and reach the invalid email address.

It is also important that emails generated from applications be sent only to valid recipient addresses. When recipient email addresses are no longer valid, the email system continues to attempt to deliver the message.

SECTION 4 - BEST PRACTICES

General:

1. Use strong types instead of Object whenever possible.
2. Use parenthesis to explicitly define the order of operations in expressions.
3. For classes to be instantiated locally, allow access to fields from outside a class through properties only.
4. For classes to be instantiated remotely, avoid properties and only implement methods.
5. For Code Maintenance and readability, do not use abbreviations unless there is an industry standard abbreviation already in existence. (int for Integer, bln for Boolean)
6. Reuse code wherever possible in order to avoid excessive repeated or duplicated code blocks.
7. Execute an Application Center load test.
8. Use lightest object possible (e.g. data table vs. dataset)
9. Make the code as structured and organized as possible
10. Use StringBuilder for complex string manipulations and when you need to concatenate strings multiple times.
11. Eliminate compiler warnings prior to .NET code review.
12. Utilize test fixtures or automated test cases.
13. Use the XML comment format to automatically insert the comment tags. (Ex: ///) Using this format will allow teams to export comments into a separate document.
14. Set the .NET framework to the same version for all projects in the solution.
15. Organize code in such a way that the solution file is in a parent folder and projects in subfolders beneath the solution folder and avoid the dependencies on specific paths. Use project references instead of assembly references when referencing another project in the solution.
16. Avoid excessive or inappropriate use of caching.
17. Avoid excessive use of SQL command execution per request.
18. Avoid excessive use of session.
19. Constant values and optional parameters should be avoided in public interfaces where possible. The recommendation is to use static readonly instead of const, or overloads instead of optional parameters in the case of methods and constructors.
    Supporting Details: During compilation, constant values are inserted into the generated IL. Optional parameters (for a method, constructor, delegate, or indexer), when not specified by the caller, are also seen as constants. Any assembly with code built against the constant value will emit IL that contains the value of that constant at the time of compilation. If the constant is subsequently changed and its containing assembly rebuilt, any assemblies with code using those constants also need to be rebuilt, as they contain the original value. For non-public constant values, this is not a problem as the constant should only be baked into the assembly that contains it, which is obviously rebuilt with the new constant values.

Code Structure:

20. Avoid the use of Literals within the code. Use constants wherever possible.
21. Nested logic should use parenthesis to explicitly set the order of evaluation.
22. Use verbose function and variable names.
23. Declare variables as close as possible to their usage.
24. Use Compiler Directives to separate test code from release code.

SECTION 5 - REQUESTING AN EXCEPTION OR CHANGE TO THE CODE REVIEW STANDARDS.

Requesting an exception or change to the standards.

1. Project Teams may request exceptions or changes to the standard.
2. The exception or change requests must be provided, in writing, to the BSSO Quality Assurance Specialist by the BSSO Program Manager of the Project. The request must include:
   - 2.1. Standard(s) for which they are requesting the exception or change.
   - 2.2. Business case justifying why the exception or change is needed.
   - 2.3. Technical details of the non-standard implementation or the change being proposed.
   - 2.4. Impact to the Department for the exception or change.
   - 2.5. List alternatives considered with pros and cons of each alternative.
   - 2.6. Provide justification of why requested exception was the chosen alternative.
3. The request for exception or change will be reviewed by a team assembled by the BSSO Quality Assurance Specialist.
4. The review team will provide a written recommendation to the BSSO Manager.
5. Final decision will be determined by the BSSO Manager.

SUPPORTING DETAILS – Requesting Exceptions or Changes

The Application Development arena is constantly changing. The .Net Coding Web Standards must also change to meet the needs of our growing Application Development community. The process for requesting exceptions or changes is the method by which the BSSO Standards and Tech group is made aware of the possible need for changes (temporarily – as an exception, or permanently – as a standards change). Project Teams requesting changes/exceptions are required to provide the listed documentation to assist in the research and understanding of their particular situation.