Kagan Yilmaz - Coding Standards

These are the coding standards and naming conventions that I use for the projects that I manage. Most of these standards are based on industry-accepted standard naming conventions for C#.NET development ASP.NET Web applications and MVC applications. They are also consistent with the .NET Framework and Microsoft guidelines.

In summary; I try my code to have following characteristics:

- easy to write , modify and extend
- clean and readable
- has value and cares about quality
- follows SOLID and DRY principles

1. Code Layout & Naming Conventions

1.1. Namespaces: Namespaces should correspond with the sub-folders within the project root folder. Namespace names should have their first letter capitalized followed by lowercase letters. Pattern should be; <company name>.<product name>.<top level module>.<bottom level module>

1.2. Interfaces: Use the prefix “I” with Camel casing and capitalize the letter following the "I". (e.g. IProduct)

1.3. Classes and Structs: Use Pascal casing (e.g. BankStatement) File names should match with class name.

1.4. Special Classes such as Collections, Delegates, Controllers, Managers, Exceptions, Utilities etc. Follow class naming conventions, but add Special Class Type name to the end of the name. (e.g. InvalidTransactionException)

1.5. Event Handler Methods:

- Use <ControlName> _<Event>
- The event handler should not contain the code to perform the required action. Rather call another method from the event handler.

1.6. Methods: Use Pascal casing (e.g. CalculateRatio)

- A method name should represent a clearly definable process.
- Getting and setting internal private member variables should not be a method with a method but via a public property get/set construct.

1.7. Variables:

- Use Camel casing (e.g. totalCount)
- Always use nouns.
- Do not use Hungarian notation.

- Prefix Boolean variables, properties and methods with “is” or similar prefixes. (e.g. isFinished)
- Do not use abbreviations.
- Do not use underscores (_).
- Do not use single character variable names like i, n, s etc. Use names like index, temp (Exception is iterations in loops)

1.8. Member Variables:

- Avoid using member variables. Declare local variables wherever necessary and pass it to other methods instead of sharing a member variable between methods. If you share a member variable between methods, it will be difficult to track which method changed the value and when.
- If it is necessary to use them; prefix them with underscore (_) so that they can be identified from other local variables.
- Do not make the member variables public or protected. Keep them private and expose public/protected Properties.
- Do declare all member variables at the top of a class, with static variables at the very top.

1.9. UI Elements: Use the abbreviations below for the UI elements:
Label=lbl, TextBox=txt, DataGrid=dtg, Button=btn, ImageButton=ibtn, Hyperlink=hlk, DropDownList=ddl, ListBox=lst, DataList=dtl, Repeater=rep, Checkbox=chk, CheckBoxList=cbl, RadioButton=rbtn, Image=img, Panel=pnl

1.10. Constants: Use all caps and place underscores between keywords. Try to be as specific as possible. (e.g. PI_NUMBER)

1.11. File Names: Use Pascal Case (e.g. LanguageDefinitions.xml)

1.12. Other:

- Avoid writing very long methods. A method should typically have 1-25 lines of code. If a method has more than 25 lines of code, you must consider refactoring into separate methods.
- Use #region to group related pieces of code together. The order should be:
- -Private member variables
  - -Private Properties
  - -Private Methods
  - -Constructors
  - -Public Properties
  - -Public Methods
- Method name should tell what it does. Do not use misleading names. If the method name is obvious, there is no need of documentation explaining what the method does.
- A method should do only 'one job'. Do not combine more than one job in a single method, even if those jobs are very small.
- Always use meaningful, descriptive words to name classes and class elements. Do

not use variable names that resemble keywords.
- Each line should contain at most only one argument.
- One declaration should be placed per line as this encourages commenting.

2.Indentation and Spacing

- Use TAB for indentation. Do not use SPACES. Define the Tab size as 4. Use standard tab settings.
- Comments should be in the same level as the code (use the same level of indentation).
- Curly braces ({}) should be in the same level as the code outside the braces.
- Use one blank line to separate logical groups of code.
- There should be one and only one single blank line between each method inside the class.
- The curly braces should be on a separate line and not in the same line as if, for etc.
- Use a single space before and after each operator and brackets.

3.Architecture

- Always use multi layer (N-Tier) and MVC architecture.
- Make decisions on the design pattern(s) that will be used before you start the development. Use Model View Controller pattern if possible.
- Never access database from the UI pages. Always have a data layer class which performs all the database related tasks. This will help you support or migrate to another database back end easily.
- Separate your application into multiple assemblies. Group all independent utility classes into a separate class library. All your database related files can be in another class library.

4.Project Planning

- I prefer the "Iterative and Incremental Development" software development model, which is a cyclic software development process developed in response to the weaknesses of the waterfall model. It starts with an initial planning and ends with deployment with the cyclic interaction in between. Depending on the project Agile Methodologies can also be practiced.
- After the initial requirements project meeting, use case diagrams are being created. Then, by utilizing these use cases, I create UI design documents. Upon approval of the use cases and the design documents, I analyze the business model and create the Application Design documents including ERD Diagram and Class Diagram.

Kagan Yilmaz - Coding Standards

5. Comments

- All source code files shall contain a “page-level” documentation block at the top of each file and “class-level” documentation block(s) immediately above each class.Block-level comments should be liberally used throughout the code where needed.
- Do not write comments for every line of code and every variable declared. Write comments wherever required. But good readable code will require very less comments. If all variables and method names are meaningful, that would make the code very readable and will not need many comments.
- Do not write comments if the code is easily understandable without comment. The drawback of having lot of comments is, if you change the code and forget to change the comment, it will lead to more confusion.
- Use the triple-whack XML comments "///" for all functions and classes. This will facilitate generating a document containing all the code documentation for your project.
- If you have to use some complex or weird logic for any reason, document it very well with sufficient comments.
- If you initialize a numeric variable to a special number other than 0, -1etc, document the reason for choosing that value.
- Perform spelling check on comments and also make sure proper grammar and punctuation is used.

6. Error Handling

- Never do a 'catch exception and do nothing'. If you hide an exception, you will never know if the exception happened or not. Lot of developers uses this handy method to ignore non significant errors. You should always try to avoid exceptions by checking all the error conditions programmatically. In any case, catching an exception and doing nothing is not allowed. In the worst case, you should log the exception and proceed.
- Make sure you have a good logging class which can be configured to log errors, warning or traces. If you configure to log errors, it should only log errors. But if you configure to log traces, it should record all (errors, warnings and trace). Your log class should be written such a way that in future you can change it easily to log to Windows Event Log, SQL Server, or Email to administrator or to a File etc without any change in any other part of the application. Use the log class extensively throughout the code to record errors, warning and even trace messages that can help you troubleshoot a problem.
- Use try-catch in your data layer to catch all database exceptions. This exception handler should record all exceptions from the database. The details recorded should include the name of the command being executed, stored proc name, parameters, connection string used etc. After recording the exception, it could be re thrown so that another layer in the application can catch it and take appropriate action.
- In case of exceptions, give a friendly message to the user, but log the actual error with

all possible details about the error, including the time it occurred, method and class name etc

- Always catch only the specific exception, not generic exception. No need to catch the general exception in all your methods. Leave it open and let the application crash. This will help you find most of the errors during development cycle. You can have an application level (thread level) error handler where you can handle all general exceptions. In case of an 'unexpected general error', this error handler should catch the exception and should log the error in addition to giving a friendly message to the user before closing the application, or allowing the user to 'ignore and proceed'.
- When you throw an exception, use the throw statement without specifying the original exception. This way, the original call stack is preserved. (e.g. use throw; not throw ex;)
- Do not write try-catch in all your methods. Use it only if there is a possibility that a specific exception may occur and it cannot be prevented by any other means. For example, if you want to insert a record if it does not already exists in database, you should try to select record using the key. Some developers try to insert a record without checking if it already exists. If an exception occurs, they will assume that the record already exists. This is strictly not allowed. You should always explicitly check for errors rather than waiting for exceptions to occur. On the other hand, you should always use exception handlers while you communicate with external systems like network, hardware devices etc. Such systems are subject to failure anytime and error checking is not usually reliable. In those cases, you should use exception handlers and try to recover from error.
- Do not write very large try-catch blocks. If required, write separate try-catch for each task you perform and enclose only the specific piece of code inside the try- catch. This will help you find which piece of code generated the exception and you can give specific error message to the user.
- Write your own custom exception classes if required in your application. Do not derive your custom exceptions from the base class SystemException. Instead, inherit from ApplicationException.
- Error messages should help the user to solve the problem. Never give error messages like "Error in Application", "There is an error" etc. Instead give specific messages like "Failed to update database. Please make sure the login id and password are correct."
- When displaying error messages, in addition to telling what is wrong, the message should also tell what the user should do to solve the problem. Instead of message like "Failed to update database.", suggest what should the user do: "Failed to update database. Please make sure the login id and password are correct."
- Show short and friendly message to the user. But log the actual error with all possible information. This will help a lot in diagnosing problems.

7. Javascript:

- Always try to use a CDN to include jQuery on your page.
- Always cache your jQuery selector returned objects in variables for reuse.
- Use camel case for naming variables.

- Use ID selector whenever possible. It is faster.
- When using class selectors, don't use the element type in your selector.
- Use only one Document Ready handler per page. It makes it easier to debug and keep track of the behavior flow.
- Avoid using .getJson() or .get(), simply use the $.ajax() as that's what gets called internally.

8. General Tips:

- Use the ASP.NET specific types (aliases), rather than the types defined in System namespace. (e.g. use int , not Int16)
- Always watch for unexpected values. For example, if you are using a parameter with 2 possible values, never assume that if one is not matching then the only possibility is the other value. Use “else” or “default”.
- Do not hardcode numbers. Use constants instead. Declare all the constants in a separate “Constant” class. However, using constants are also not recommended. You should use the constants in the config file or database so that you can change it later. Declare them as constants only if you are sure this value will never need to be changed.
- Do not hardcode strings. Use xml/ resource files.
- Convert strings to lowercase or upper case before comparing. This will ensure the string will match even if the string being compared has a different case.
- Use String.Empty instead of “”
- Use enum wherever required. Do not use numbers or strings to indicate discrete values. Do use singular names for enums.
- Do not programmatically click a button to execute the same action you have written in the button click event. Rather, call the same method which is called by the button click event handler.
- Never hardcode a path or drive name in code. Get the application path programmatically and use relative path.
- Never assume that your code will run from drive "C:". You may never know, some users may run it from network or from a "Z:".
- In the application start up, do some kind of "self check" and ensure all required files and dependencies are available in the expected locations. Check for database connection in start up, if required. Give a friendly message to the user in case of any problems.
- If the required configuration file is not found, application should be able to create one with default values.
- If a wrong value found in the configuration file, application should throw an error or give a message and also should tell the user what are the correct values.
- Do not have more than one class in a single file.
- Have your own templates for each of the file types in Visual Studio. You can include your company name, copyright information etc in the template. You can view or edit the Visual Studio file templates in the folder C:\Program Files\Microsoft Visual Studio

8\Common7\IDE\ItemTemplatesCache\CSharp\1033. (This folder has the templates for C#, but you can easily find the corresponding folders or any other language)

- Avoid having very large files. If a single file has more than 1000 lines of code, it is a good candidate for refactoring. Split them logically into two or more classes.
- Avoid public methods and properties, unless they really need to be accessed from outside the class. Use “internal” if they are accessed only within the same assembly.
- Avoid passing too many parameters to a method. If you have more than 4~5 parameters, it is a good candidate to define a class or structure.
- If you have a method returning a collection, return an empty collection instead of null, if you have no data to return. For example, if you have a method returning an ArrayList, always return a valid ArrayList. If you have no items to return, then return a valid ArrayList with 0 items. This will make it easy for the calling application to just check for the “count” rather than doing an additional check for “null”.
- Use the AssemblyInfo file to fill information like version number, description, company name, copyright notice etc.
- Logically organize all your files within appropriate folders. Use 2 level folder hierarchies. You can have up to 10 folders in the root folder and each folder can have up to 5 sub folders. If you have too many folders than cannot be accommodated with the above mentioned 2 level hierarchy, you may need re factoring into multiple assemblies.
- If you are opening database connections, sockets, file stream etc, always close them in the finally block. This will ensure that even if an exception occurs after opening the connection, it will be safely closed in the finally block.
- Declare variables as close as possible to where it is first used. Use one variable declaration per line.
- Use StringBuilder class instead of String when you have to manipulate string objects in a loop. The String object works in weird way in .NET. Each time you append a string, it is actually discarding the old string object and recreating a new object, which is a relatively expensive operations.
- Do not use session variables throughout the code. Use session variables only within the classes and expose methods to access the value stored in the session variables. A class can access the session using System.Web.HttpCOntext.Current.Session
- For the logout page; use: 1- Session.Clear(); 2- Session.Abandon(); 3-//Then Redirect to logout page.
- Do not store large objects in session. Storing large objects in session may consume lot of server memory depending on the number of users.
- Always use style sheet to control the look and feel of the pages. Never specify font name and font size in any of the pages. Use appropriate style class. This will help you to change the UI of your application easily in future. Also, if you like to support customizing the UI for each customer, it is just a matter of developing another style sheet for them
- Create classes and properties to get application settings
- Utilize page life cycle events; Init, Load, PreRender, Save, Render, Unload. Place your code accordingly.
- Utilize nested master pages in order to increase maintainability and decrease

development time. Master pages make it easy to create pages that have a consistent look. Do not define more than enough number of master pages. Try to keep them simple and to the point.

- Utilize CSS and themes. Group all CSS elements on a separate file. Do not use inline style elements. This will keep your site more accessible both to disabled people and to search engines bots. In order to test this, take out the css file from a page and see if the page still functions as it should.

9. Database

- All scripts to update related database systems shall be stored is a project subfolder named “SQL”. Under this sub-folder, a subfolder shall be created for the particular build version.
- All transactions has to be ACID. ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably.
- Do not use abbreviations.
- Do not use underscores (_).
- Always use meaningful, descriptive words to name database elements.

9.1.Columns:

- Use Pascal casing (e.g. LastName)
- Do not attach the table name to a column. Keep them short and simple.
- Do not use reserved keywords.

9.2.Tables:

- Use all caps with no underscores.
- Use singular nouns. (e.g. SALESTAX)
- Always have at least one Primary Key in each table and use <Table Name><ID> (e.g. ProductID)

9.3.Stored Procedures

- Do not use _sp for performance considerations. When you call a stored procedure that starts with sp_, SQL Server always checks the master database first, even if the stored procedure is qualified with the database name. After not being able to locate it, SQL Server assumes the procedure isn't in cache (and thus must be recompiled) and acquires an exclusive compile lock on the stored procedure for a short time. However, the short time that the lock exists is enough to cause performance problems.
- Use Pascal casing (e.g. GetTotalCount)

9.3.1. Insert / Update / Delete (CRUD) Procedures

Use action keyword following with the singular noun.(e.g. UpdateCustomer, DeleteCustomer, InsertCustomer)

9.3.2.Select Procedures

- Use prefix "Get" with singular nouns to retrieve a single row along with "By" keyword which indicates the input parameter (e.g. GetCustomerById)
- Use plural nouns to retrieve a list of records (e.g. GetCustomers)

9.3.3.Stored Procedure Parameters & Variables: Use Pascal casing (e.g. LastName)
9.3.4.Report Stored Procedures: Use prefix "Rpt" with a name representing the actual report explicitly.

9.4.Custom Functions
Use prefix "Get" with singular nouns to retrieve a single row along with "By" keyword which indicates the input parameter (e.g. GetCustomerNameById)
Use prefix "Is" to return a Boolean value (e.g. IsCustomer)

9.5.Triggers: Use prefix "Trg" along with the table name and role. (e.g. TrgProductValidateSalesTax)

10. WEB API

I use MVC Controllers when I intend to respond with a View(), and I'll use a Web API for anything that isn't dependent on a particular view. Important points include:

10.1. Stability and Consistency : Choose method names that make sense from the perspective of the API consumer using well defined nouns(not verbs). Do not use GET for state changes. Only use plural nouns.Examples:
GET /tickets - Retrieves a list of tickets , GET /tickets/12/messages - Retrieves list of messages for ticket #12
GET /tickets/12 - Retrieves a specific ticket , GET /tickets/12/messages/5 - Retrieves message #5 for ticket #12
POST /tickets - Creates a new ticket
PUT /tickets/12 - Updates ticket #12
PATCH /tickets/12 - Partially updates ticket #12
DELETE /tickets/12 - Deletes ticket #12
exception : partial responses - /tickets?fields=id,name,description
10.2. Documentation : An API is only as good as its documentation. The docs should be easy to find and publicly accessible.
10.3. Security : Always use SSL. Don't expose your domain model in the API.
10.4. Content Negotiation: Use HTTP headers for serialization formats. Content-Type defines the request format. Accept defines a list of acceptable response formats.
10.5. Flexibility : Keep it simple and usable. Provide filtering, sorting, field selection and paging for collections.
GET /cars?color=red Returns a list of red cars
GET /cars?seats<=2 Returns a list of cars with a maximum of 2 seats

GET /cars?sort=-manufactorer,+model
GET /cars?page=3&pageSize=10
10.6. Error Handling : Create a well defined error format using appropriate HTTP status codes. (Don't limit your choice of error codes to 200 and 500) Example:

```
{
        "errorCode": "401",
        "description" : "Authentication needed",
        "detailsUrl" : http:// ekagan.com/api/products/errors/123
}
```

10.7. Performance : Utilize caching when possible. Distributed caching approaches such as memcached(between the app server and db), cache API responses (using Web API caching library-nuget) and utilize CDN solutions.

Finally, an API is a user interface for developers. I put the effort in to ensure it's not only functional but pleasant to use.

11. Deployment

There are several factors to consider when choosing a deployment strategy: the type of application, the type and location of users, the frequency of application updates, and the installation requirements.

11.1. Versioning

- I use a sequence-based software versioning. In sequence-based software versioning schemes, each software release is assigned a unique identifier that consists of one or more sequences of numbers or letters. You can also utilize the AssemblyVersionAttribute .Net class. I follow Microsoft style versioning:
  major.minor[.build[.revision]]
- When specifying a version, you have to at least specify major. If you specify major and minor, you can specify an asterisk (*) for build. This will cause build to be equal to the number of days since January 1, 2000 local time, and for revision to be equal to the number of seconds since midnight local time, divided by 2.
- If you specify major, minor, and build, you can specify an asterisk for revision. This will cause revision to be equal to the number of seconds since midnight local time, divided by 2.

11.2. Web Server Configuration

The steps below will increase the performance of your applications.
Machine.config file:
Add/Modify:
<connectionManagement>

<add address="*" maxconnection="100" /> </connectionManagement>
<processModel autoConfig="true"/> according to list below:
maxconnection 12 * #CPUs maxIoThreads 100 maxWorkerThreads 100 minFreeThreads 88 * #CPUs
minLocalRequestFreeThreads 76 * #CPUs
Note: The recommendations that are provided in this section are not rules. They are a starting point. Test to determine the appropriate settings for your scenario. If you move your application to a new computer, ensure that you recalculate and reconfigure the settings based on the number of CPUs in the new computer.
Modify:
<section name="deployment" type="System.Web.Configuration.DeploymentSection, System.Web, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" allowDefinition="MachineOnly"/>
The change makes this section to be configured only in the Machine.config file, which is located in %SystemRoot%\Microsoft.NET\Framework\versionNumber\CONFIG.
·Web.config file:
Modify:
<compilation debug="false" strict="false" explicit="true"> Huge impact on performance.

Add:
<system.web>
<deployment retail="true"/> </system.web>
This change enforces the 'debug' flag to false in every web.config on the machine.
Also it disables page output tracing and it forces the custom error pages to be shown to remote users instead of the real error messages.

IIS Setup
Utilize caching. Select your website; Right click any static folder inside your website- example images. Do following:
Select HTTP Headers Enable content expiration
Set a value for "Expire after" say 10 (in days)
Do this for all the static folders- i.e. whose content does not change very frequently.

12. Documentation

- Create a help / faq / knowledge base section for the backend systems in order to provide enough information to explain features.
- Technical documentation should be based on application comments and should include the database elements along with application infrastructure and design patterns used.
- Create detailed user manuals using screenshots in Adobe PDF format.
- Deploy all application related design and technical documents including Database ERD, Class UML, Use Cases, Database and Application Documentation files, training

documents.

13. Multithreading:

Do not use Multithreading. Use the async/await functionality in C# 5 to represent asynchronous workflows.
At all costs avoid shared memory. Most threading bugs are caused by a failure to understand real-world shared memory semantics. If you must make threads, treat them as though they were processes: give them everything they need to do their work and let them work without modifying the memory associated with any other thread. Just like a process doesn't get to modify the memory of any other process.
If you use Thread.Sleep with an argument other than zero or one in any production code, you are possibly doing something wrong. Threads are expensive; you don't pay a worker to sleep, so don't pay a thread to sleep either. If you are using sleeps to solve a correctness issue by avoiding a timing problem -- as you appear to be in your code -- then you definitely have done something deeply wrong. Multithreaded code needs to be correct irrespective of accidents of timing.