# Coding Style, Standards, and Reviews

JD Kilgallin

CPSC:480

10/17/22

Photo: Megan Smith, New York Times
Known for: First M.S. from MIT Media Lab, Product Design Lead at Apple, VP at Google, MIT Board of Trustees, CEO of gay.com, 2028 Olympics committee, cofounder of the Malala Fund, 3rd CTO of the United States

# Coding Style, Standards, and Reviews

JD Kilgallin

CPSC:480

10/17/22

# Notes

- Project 2 Team Participation Survey due tonight. Counts as a quiz.
- Every assignment has said this, but blanket course policy will now be that **document submissions to Brightspace and releases in GitHub must be as PDF**. .docx is poorly standardized and subject to display differently across editors. This is good practice for resumes and other documents distributed professionally too.
- An issue or backlog item is meant for a developer to understand and be able to work on the item, and would need more than the one-sentence user story, which is user-focused rather than system-focused.
- GitHub has built-in support for task lists on issues: start a line with "- [ ] " ("- [x] " for completed tasks) followed by the description of the task.
- Exercise 4: Assignment asked for concern analysis *and* user story, and three files+line numbers per concern location.
- "DTO" stands for "Data Transfer Object", and represents a native object that corresponds closely to the serialized form of an object. These are common for HTTP requests/responses, or SQL records.

# Learning Objectives

- Code quality concepts
- Elements of coding standards and style guidelines
- Reasoning behind coding standards
- Code review concepts and strategies

# Code Quality

- Well-written code makes programs that are easier to develop and maintain, improving product reliability and performance, and lowering development costs.
- Starts with quality of class model and software architecture design, and continues through deployment process.
- Codebase should be easy to navigate, comprehend, and search by concern (core or cross-cutting).
- Individual code snippets should be easy to read and understand.
- It should be easy to modify or extend code with confidence in accuracy of the changes and freedom from unintended side effects, while maintaining performance, testability, and other concerns.

# Code Quality Considerations

- You should assume that code will be read and modified by someone besides yourself (also if you write 100 lines per day, you'll have forgotten everything in your head if you revisit the code yourself in a month or a year).
- Code quality is not directly observable by software consumers, but good code lets developers implement more features and put more focus on quality issues like performance in the same amount of time.
- There is more variation in developers' ability to write good code than in ability to write functional code; this is a major factor in the difference in productivity of a "great" team and an average team.
- Writing good code the first time is slower, but leads to less technical debt and time to rewrite or improve later and ultimately saves time, not to mention developer frustration.

# Maintainability

- The ability to make changes to product code over time.
- Duplicated code makes software changes substantially harder and more prone to break. Consolidate widely-used code into one method.
- *Orphaned code* is code that is no longer used by any part of the program. Large codebases can contain over a million lines of orphaned code, which hinders code comprehension and navigation.
- Scattered code is hard to find and change correctly without unintended side effects.
- Tangled code also makes it difficult to change code, as it requires understanding (and testing) of only tangentially-related functionality.
- Technical debt requires more maintenance work to refactor.

# Readability

- Code should be easy to understand and follow when making modifications to the program.
- The most important thing is that code is not **mis**understood. Lack of curly braces and improper indentation can easily obscure bugs. Inconsistent terminology in variable names can cause confusion too.
- It is also important that code purpose should be easy to determine *and recall*. If it's not clear what a variable represents, it may require consideration each time you read the code, whereas after figuring good code out once, it should be easier to refresh yourself again.
- Breaking long functions up lets a complex method read like a checklist, and breaking long lines up simplifies comprehension and modification of each part. Ideally, all code under consideration should fit on one screen, or even a half-screen.

# Coding style

- Conventions and implementation patterns used across a codebase.
- Encompasses class/method/variable naming, organization of code (by file, by commit, etc), indentation/bracketing/line breaks, commenting and code documentation, function call patterns (recursion, chaining, etc), logging, input validation/exception handling, attribution, & more.
- Consistency is key; it's better for everyone to use the same style, even if it's not always the best style, unless a more important consideration must take priority.
- What's considered good style will depend on the language and framework used, as well as needs of the program and the team's philosophy.

# Coding Standards

- Requirements for code implementation and coding style developed and enforced by the development team or organization.
- Some elements will be common across languages and/or projects, while some will vary. There are industry standards for most languages
- May be formalized in a coding standards document or may be less formal and require identifying conventions directly from prior code.
- *Linting* is the practice of automatically validating, enforcing, or applying coding standards, usually on code checkin or set schedule. Simple issues (e.g. tabs vs spaces) can be automatically standardized. Some (e.g. most compiler warnings) must be rewritten by the developer, and not all can even be automatically checked.

# Example Coding Standards Elements

- Naming conventions, capitalization
- Whitespace, line breaks, brackets, formatting
- Orderings
- Exception handling (input params, nulls, etc)
- String building
- Header files
- Warnings
- Comments
- Logging and levels

# Coding Standards Benefits

- Code that adheres to standards is more readable and maintainable because the standards enforce good practice, and knowledge of how one part of a program functions translates to knowledge of other parts of the codebase.
- Lack of standards causes code style to diverge over time as code evolves organically. This makes it **much** less readable/maintainable.
- Consistent coding style makes it easier to automate code search, data mining, code generation (e.g. logging), and test case generation.
- Coding standards reduce some common bugs, security issues, and performance issues and facilitate troubleshooting.

# Code reviews

- Process of having other developers read proposed code changes for approval and comment before being checked in to source repository.
- Increase the odds of catching bugs and code quality issues that impact productivity (and ultimately customers if they're not caught at a later time) and lead to growth of developers' programming skills.
- Ensure other developers are aware of current state of the codebase.
- Tend to lead developers to write better code up front, to preempt likely objections and improve teammates' perceptions of the author.
- Different teams will have different processes and cases where a code review should be issued; some may require review of every change.
- Whenever code reviews are used, the code being reviewed is expected to be complete, tested, and compliant with standards.

# Recap

- Code quality encompasses many aspects of the source code, like comments and variable names
- A consistent code style increases readability and maintainability of the source, as well as reliability of the product and user perceptions.
- It is important to be able to comprehend and recall program functionality on a quick read and have confidence in code changes; this requires more time up front but saves time in the long run.
- Following separation of concerns and limiting technical debt are two key parts of code quality.
- Many teams adhere to code standards, and may use a linter to enforce compliance.
- Code reviews are a common way to improve code quality (& skills)

# Keyfactor Standards

- Mostly written for C#, but largely also apply for Java and C.
- Mostly written for Keyfactor platform, but apply to other products when there aren't other standards specific to that project.
- Followed by the product development team as well as engineers in other business areas in some cases.
- Do not encompass all requirements for publicly-facing, open-source product code, but apply to open-source code also.

# Using Outside Code

- Content on the Internet is copyrighted by its author and subject to their terms of use or that of the hosting site. Licenses that prohibit commercial/non-personal use without written consent must be followed. If a code excerpt doesn't include its own license notice, a site's "terms of service" or other legal/policy/about pages will govern the code.
- Licenses that require open-sourcing code that makes use of the online code ("viral" or "copyleft" licenses) are incompatible with our business and cannot be used. The primary example of this is the GNU Public License or GPL.
- Licenses that require software be licensed under the same terms as the acquired software are untenable too, such as the Creative Commons "ShareAlike" license.
- Licenses for code that allow full use, modification, and distribution of the code are okay. Some of these require distributing an attribution notice or license terms, which we can do.
- Some attribution requirements, notably including Stack Overflow, require a comment in source code linking to the original post it was copied from. Regardless of license requirements, **include an attribution link in the source as a minimum for all code not authored by Keyfactor.**

# Variable Declaration

- The use of implicit types in declarations is limited to situations where it is necessary because the type cannot be determined.
- Namespace aliases for include/import references are encouraged.
- Use PascalCasing for class, property and method names.
- Use camelCasing for local variable declaration.
- Use an underscore at the beginning of private, class-level variables .
- Be as descriptive as possible for names.
- Begin interface names with a capital ‘I’.

# Formatting

- Always use curly brackets for conditional statements, even if the action could be done on the same line. Avoid single-line conditionals and while loops; braces are free.
- Align matching curly braces to the same level of indentation.
- Use the Visual Studio formatting tool to format code prior to checkin.
- Code that must execute at the end for resource cleanup, etc, regardless of return path or exception condition, should go in a “finally” block.
- Use the “Remove and Organize Using Statements” function. This helps avoid merge conflicts and an excessive number of changes to a file that are simply due to reordering using statements.

# Logging Levels

- Error – Errors and exceptions. For exceptions, always use LogHandler.FlattenException to format the message, stack trace, and any inner exceptions. Any time the server returns a 5xx HTTP response code, the issue should be logged at this level. For egregious authorization violations that indicate malicious access attempts, this should be used too.
  - This should NOT be used to log unexpected input from an agent or API client – log a short message on WARN, and FlattenException on DEBUG.
- Warning – Non-fatal error conditions, brief notes on unexpected third-party input, or when one or more items fail in a batch process.
- Info – Informational messages for events like when a service starts or stops. "understandable" authorization violations can be logged here for auditing purposes.
- Debug – Program flow and important values. Anywhere code comments would be entered are good candidates for debug logging. Remember that anything you'd want to find in a customer log should be logged BELOW this level. However, it should be possible for customers to run at debug level for hours to days without generating unusably large files.
- Trace – Write out larger values, or values within a batch processing loop. Method entry and exit are also logged here for extra control-flow clarity (typically handled with attributes).

# Globalization

- All strings presented to the user are globalized into resource files. There is a resource file associated with each functional area. Strings should be added to the appropriate resource file.
- Strings that appear only in the application log should not be globalized – these should be readable by CSS support.
- Globalized strings are also used within the web console's HTML and Javascript files.
- No string should be referenced without an English representation for accessibility by developers.

# Comments

- Add a comment anywhere the functionality is not obvious by inspection.
- Use “TODO” when adding comments describing incomplete functionality. TODO comments can be tracked using the Task Explorer window, available under the “View” menu.
- DO NOT COMMENT OUT CODE. This leads to long, hard-to-read files, trips up code merges, and rarely is useful for recovering old code. If you remove code that future developers may wish to refer to, add a comment like “Code removed mm/dd/yyyy. See version history for reference”.
- The main exception to this is if there’s a tempting “easy” short implementation that does not work, in which case a note like “The below commented-out implementation does not work because XYZ; don’t try to change this to do that” is appropriate.

# Strings

- Format – Use C# interpolated strings for formatting ($”The value is {myValue} at index {i}”). This feature creates simpler and more readable string formatting compared to String.Format variants, or worse, "The value is" + myValue + " at index " + I.
- Multi-stage string building – Use the StringBuilder class for incremental string building or within loops – C# string objects are immutable, which requires copying the whole buffer for each concatenation, so you should never have a loop that does something like “myString += newClause”

# Null handling

- Cases where a value may be null must always be handled.
  - If it indicates as an error, the error should be logged and handled
  - If a null value is possible in normal operation, without an error, consider the appropriate default.
- Make use of C# nullable types (int?), null-propagation (?.), and null-coalescing (?? ) where appropriate. "myObject?.Value ?? -1" rather than "myObject == null ? -1 : myObject.Value".
- int? syntax is preferred over Nullable<int>
- An object reference error (null pointer) is always a bug.

# Exception Handling

- Methods should generally check their input before operation. Missing or invalid parameters should immediately result in a log message and exception.
- Return values, ESPECIALLY from Entity Framework or any "SingleOrDefault" queries, should be null-checked, and unanticipated nulls should result in a clear error logged and returned. Object reference exceptions are always a bug.
- Methods that may throw exceptions for multiple reasons should throw the most precise Exception type possible. New exception types should be created for easily anticipated exceptions that do not already have a type.
- Exception Filters – Use exception filters instead of re-throwing exceptions.

# Data Structure Access

- Use LINQ method syntax: Collection.Where(item => item.value == criteria).Select(item => item.property) rather than a for loop.
- Do not use SQL-like syntax “item.property FROM collection WHERE criteria”. If you think you have a case where this syntax allows more performant joins or otherwise is preferable to the method syntax, consult your team lead or a SQL Server/Entity Framework expert.
- Avoid using static strings that refer to a schema component. For example, “.Include(“Agents”)” should not be used, and instead written as “.Include(c => c.Agent)”. This ensures that schema changes will cause necessary updates to be caught at compile time.

# Anonymous Functions

- For short event handler callbacks, local data-processing methods, and LINQ parameters.
- Avoid anonymous functions that do not easily fit on one line, as this suggests complex functionality that should be subject to unit tests which cannot easily be written against local anonymous functions.
- Conversely, it is preferable to use anonymous functions inline for very short data transformation tasks that do not require readers to navigate to a separate function definition.
- Common operations for a particular type can be encapsulated as an extension method (e.g. get template by oid) in the Service layer.

# Avoid "else"

Avoid excessive indentation by handling special conditions in an “if” statement and keeping the “expected” path at an outer level.

**Bad:**

```
if (valid) {
        doAllTheUsualStuff();
}
else {
        throw new Exception(…);
}
```

**Good:**

```
if(!valid) {
        throw new Exception(…);
}
// Don't put an else here!
doAllTheUsualStuff();
```

# Return early

- Avoid allowing execution past the point that the return value is known – if a return value is set but the method continues running, it isn't clear whether it can still be modified or re-assigned.
- Avoid assigning a value that will always be re-assigned and will never actually be returned.

**Bad:**

```
int result = -1;
for(...){
      if (condition) {
            result = i;
            break;
      }
}
return result;
```

**Good:**

```
for(...){
      if (condition){
            return i;
      }
}
return -1;
```

# Line length

- Avoid excessively long lines – There is no hard limit to a line length, but it should generally be readable and should fit entirely on screen with typical display settings. Code should be limited to 80 characters when system convention dictates this (e.g. embedded C code).
- Newline does not end a statement, so break long lines at natural points and align subsequent lines to show clear parallelism.

```
int x = MyCollection.Where(item => item.key == parameter)
        .Select(item => item.value)
        .Distinct()
        .SingleOrDefault(v => v > 0) ?? -1
```

# C/C++

- Where applicable, standards equivalent to the corresponding C# standards should be met. Where there are no equivalents, follow industry best practice
- Functions and data types must be specified and described in a .h header file, while the implementations must be done in a corresponding .c file.
- Header files should use the #ifndef #define pattern to avoid conflicts from multiple inclusions.
- Code should compile without warnings on gcc using –Wall.
- For repeatable results, executables must always be compiled with a make rule that is checked in to version control along with the source.
- A segmentation fault is always a bug.
- A memory leak is always a bug.

# HTML/CSS/JavaScript

- An uncaught exception that appears in the debug console is always a bug. Always test with the console open to catch these.
- DO NOT USE THE “debugger” STATEMENT IN CODE. Due to a history of developers failing to remove these, in-browser breakpoints must be used instead. Similarly, do not use “alert” for debugging purposes.
- Network communication should always be done through AJAX with jquery. Application data should not be returned from .html files; these should only be used to transmit the web application itself. This ensures that all data access is available to alternative clients and that all data access methods are fully testable.
- Use of the “position”, “float”, “top”, “left”, and “right” properties are strongly discouraged in favor of “padding” or “margins”. Padding is for space inside the element and margins is outside.
- Do not use tables for formatting; only use them for true tabular display.
- Use of jquery widgets is encouraged for any complex objects used multiple times.

# Developer Testing

- All service methods and API endpoints must have unit tests and integration tests for each significant code path.
- Service unit tests are NOT able to mock out EF extension methods or verify invocation count on them. Use TestHelpers.MakeMockable to make a mock object from a collection.
- API unit tests can create a controller object directly and do not need to make a web request. This improves performance and isolates the cause of failures to the code, rather than network connections.
- Integration tests should not mock out underlying layers.
- A test.runsettings file allows data specific to your environment, such as credentials, to be specified. Local content should not be checked in to source control.

# Code Reviews

- Unless told otherwise under team policy, send a code review if:
  - You are changing something very close to release or that will go to customers imminently.
  - Your code does something exceptional, novel, or unorthodox.
  - You are working in an area you're not familiar with and are not confident in your changes or design
  - Another developer requests to review.
- Send the review to your team lead, internship or new-hire mentor (if applicable), experts in the product area (consult as needed to determine who this is), and any developers who ask to be included

# Code Review Checklist

- Code must have been reasonably tested first, including new automated tests, and all automated tests must pass.
- Code must comply with coding standards and style guidelines.
- You must include all changes you want to check in, and no files that have not actually been modified.
- The changes are applied with the latest build of Main merged in, so that the code under review is the exact code that would appear in Main after the checkin.
- The description clearly states what the change is intended to do, the relevant TFS item numbers, and any concerns you have about your implementation.

# Code Review Responses

- If you are asked to review code, please respond in a respectfully prompt timeframe.
- If you are unable to review the code or don't believe you are qualified, you can decline the review. If you are able, you should promptly accept the review even if you can't complete it immediately.
- Add comments as needed, and finish with one of the following statuses:
  - Looks good – Would be fine as is, possibly with some polish or minor improvements listed in your comments
  - With comments – Some concerns prevent immediate signoff, and you would like responses to your comments and to reach a consensus on these concerns before the code is checked in.
  - Needs work – Significant changes (or many small changes) are needed and you would like to reject these changes and participate in another code review after they have been addressed.

# References


- Megan Smith: "You can Affect Billions of People". Susan Dominus. Oct 2014. New York Times.
- The Elements of Programming Style, 2nd edition. Brian Kernighan and PJ Plauger. 1978.  Mcgraw-Hill.
- Keyfactor Coding Standards. Gary Galehouse, Jonathan Kilgallin, et al. 2019. Keyfactor.
- Avoid Else, Return Early. Tim Oxley. 2013. "Probably Wrong" blog.
- Lectures on High-Performance Computing. Jesus Fernandez-Villaverde and Pablo Guerron. Jan 2022. University of Pennsylvania.
- Google Style Guides. Google. 2008-2022.
- Coding Style. Charlie Wong et al. April 2019. Read the Docs.


- *Bring a laptop to class Wednesday*