Clean Code

# Resources

Code Complete by Steve McConell

Clean Code by Robert C Martin

The pragmatic programmer by Andrew Hunt

# Principles

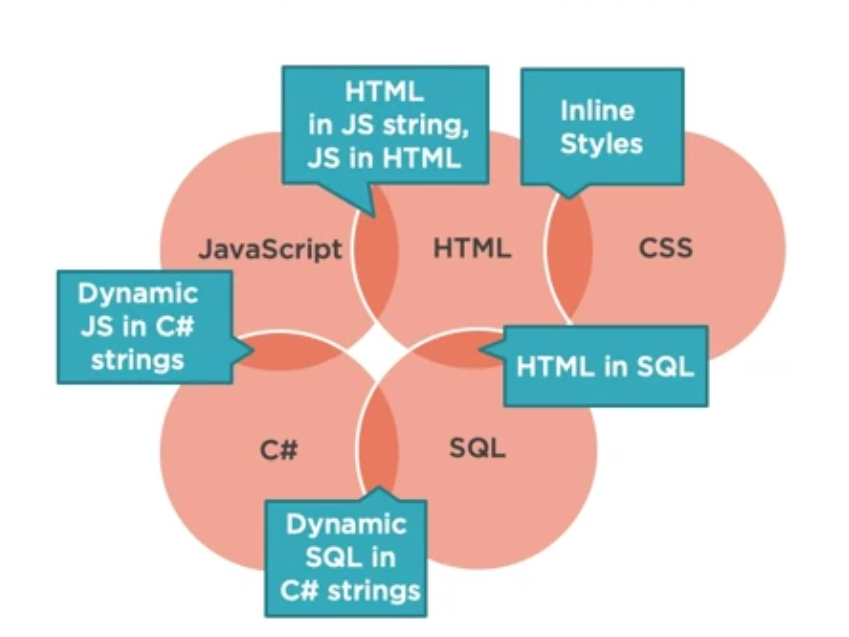
## Choosing the Right tool

An example of using the wrong tool would be having to come up with a solution or a work around to fix the problem. Without focusing on the underlying problem that you are using the wrong tool

For example using a regex for email validation, which can produce a very complex regex

Boundaries matter

Examples of wrong boundaries



* 1. Html in Js and vice versa: eliminates caching and reuse.
  2. Html in Sql: binds presentation and data storage together
  3. Inline style: hard to override and unable to reuse it
  4. Dynamic Sql in C#: using ORM is safer approach and easier to use
  5. Js in C# strings: its better to return Json and use it in Js

Benefits of staying native

(keeping Js in Js file, Html in html file and so on…)

* Code is cached
* Easier to catch errors in syntax checking and pre compiler before running the code
* Separation of concerns
* Reusability
* Avoid string parsing

**Avoid using one language to write another language via strings**

**Every tech has its potential evil**

* + Linq to sql: produces massive queries and outer joins
  + Writing js and html in strings in C#

## High signal to noise ratio

Any logic that follows the TED rule

**Terse**: should be wordy

**Expressive**: clear what the code is doing

**Does one thing**

### The rule of 7

The human brain can hold 7 items in short-term memory

this should impact how the number of parameters in function, number of methods in class, number of variables in scope

## DRY

Don’t repeat yourself

Duplication issues:

1. Create a lot of noise.
2. Increase number of lines of code.
3. Create maintenance problem: either will forget to modify in all repetitions or will have to go and modify each one by one.

## Self Documenting Code

Without needing comments or documentation

The code should have the following properties:

* Clear intent: should know what the writer was trying to accomplish
* Layers of abstraction: provide different levels of details
* Format for readability:
* Favor code over comments: the code should be as expressive as it can be without relying on comments

# Naming

## Naming classes

1. Should be a noun
2. Specific as possible
3. Single responsibility: should be expressive to lead the design of class to be single responsibility
4. Avoid generic suffixes for ex: use Product instead of ProductManager

## Method naming

* Should describe what the method will do

GetRegisteredUsers, SendEmail

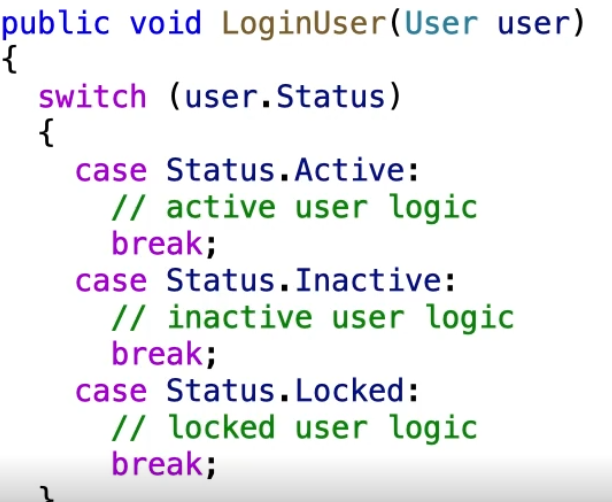
* Shouldn’t be about when the method is called for example Start or Complete
* The method shouldn’t do more than what its name means and shouldn’t have side effects like Login function shouldn’t create user session
* If a method has (And,Or,If) it is probably doing more than one thing
* Avoid non-standard abbreviations

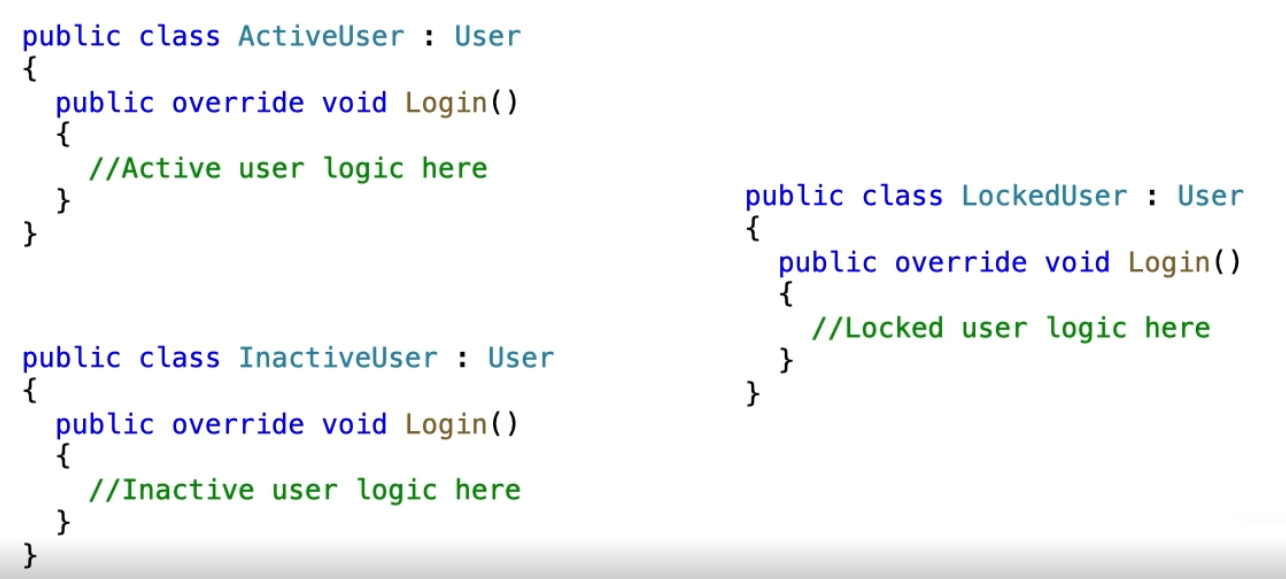
## Naming variables

* Booleans should have a clear meaning for example: loggedIn instead of login. It should be asking a yes/no question
* Be symmetrical in opposites for ex: OpenConnection/CloseConnection, Create/Destroy, Login/Logout

# Conditionals

* Prefer positive conditionals over negative for ex: !loggedIn
* Use ternary operator when needed but not more than one nested ternary
* Be strongly typed; Use constants for strings and numbers and name what they represent
* Encapsulate complex conditionals into separate method
* Use polymorphism instead of switch cases in code

 instead of this we should define an abstract method called login inside User class and each specific user will override this method



## Arrow Code

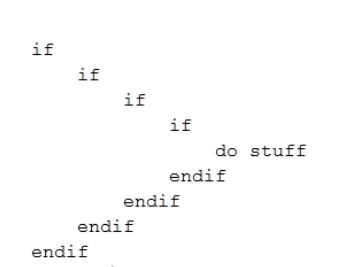
When using a lot of conditionals and else statements it looks like an arrow code

Bec it has high cyclomatic complexity meaning a lot of paths created by if and else statement

# Writing functions

Why to write functions

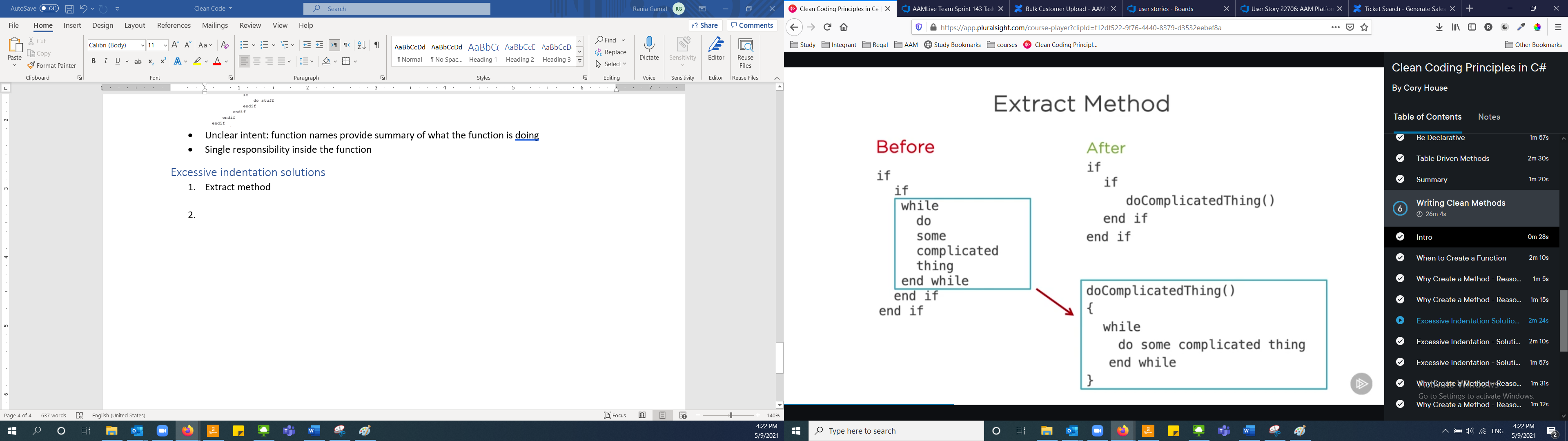
* Duplication
* Indentation: having to write nested blocks like if conditions inside loops and etc. this means code is too complex and needs to be split into function or more



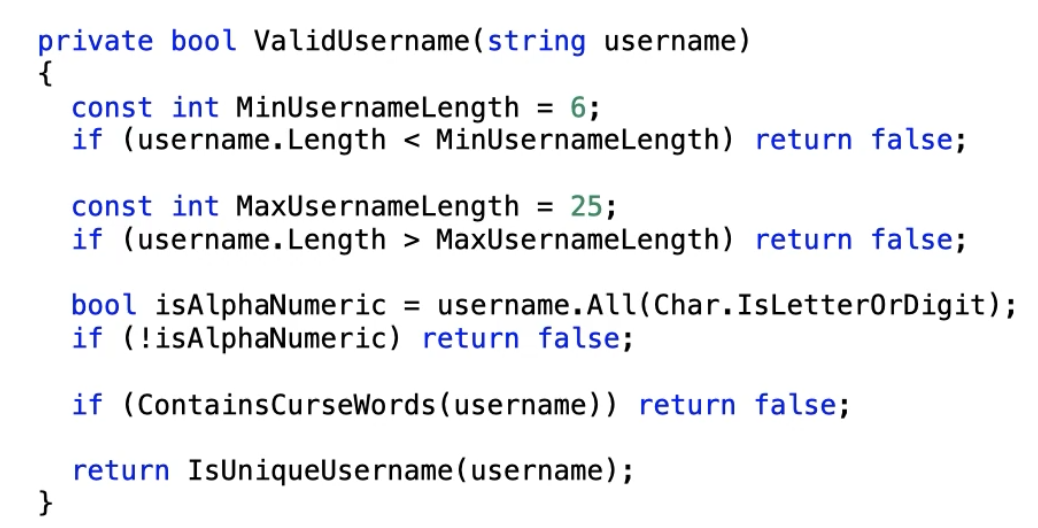
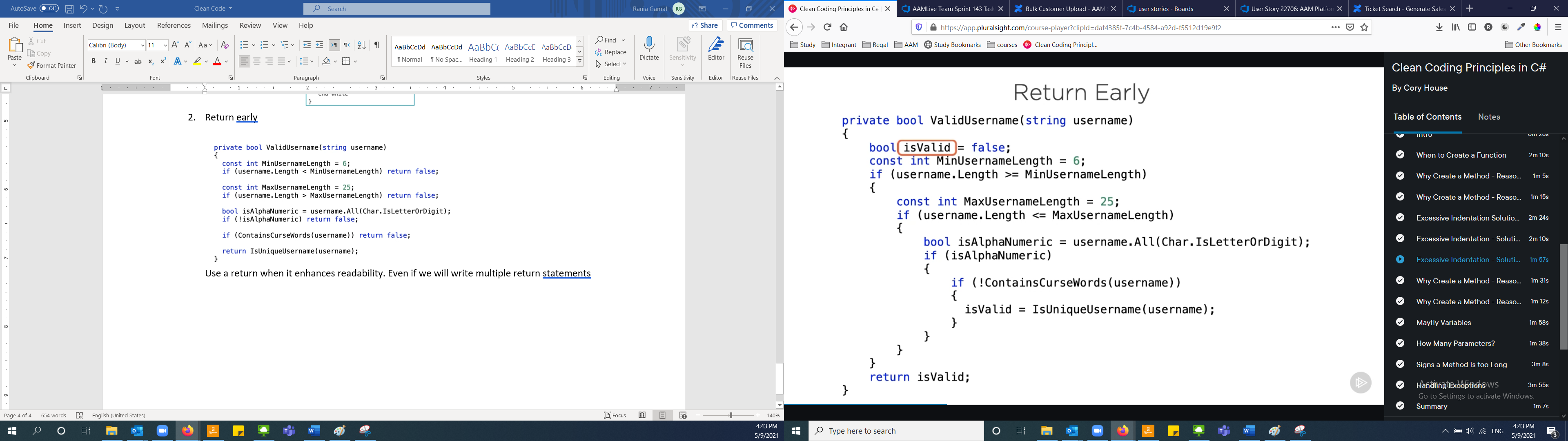
* Unclear intent: function names provide summary of what the function is doing
* Single responsibility inside the function

## Excessive indentation solutions

1. Extract method



1. Return early



Use a return when it enhances readability. Even if we will write multiple return statements

## Mayfly variables

We should limit the number of variables defined in the beginning of function and define variable when its needed inside the function.

Bec it will be harder to keep track of the variables defined at the top.

## Number of parameters

The number of parameters should be between 0-2 params

Flag arguments (isAdvisor) can be replaced by two separate methods (not always)

## Signs to split the function

* A lot of whitespace and comments
* Scrolling required
* Unable to name it
* Having multiple conditionals
* Hard to understand: more than 7 parameters and 7 variables in scope
* Simple functions should be longer, and complex functions should be shorter

# Exceptions

## Types

* Unrecoverable: null-reference, file not found, access denied
* Recoverable: retry to open connection, wait for few second and try again to send
* Ignorable: just log it, should have no impact on the user

If there is an error that you can’t handle, you should let it bubble up (fail fast and loud)

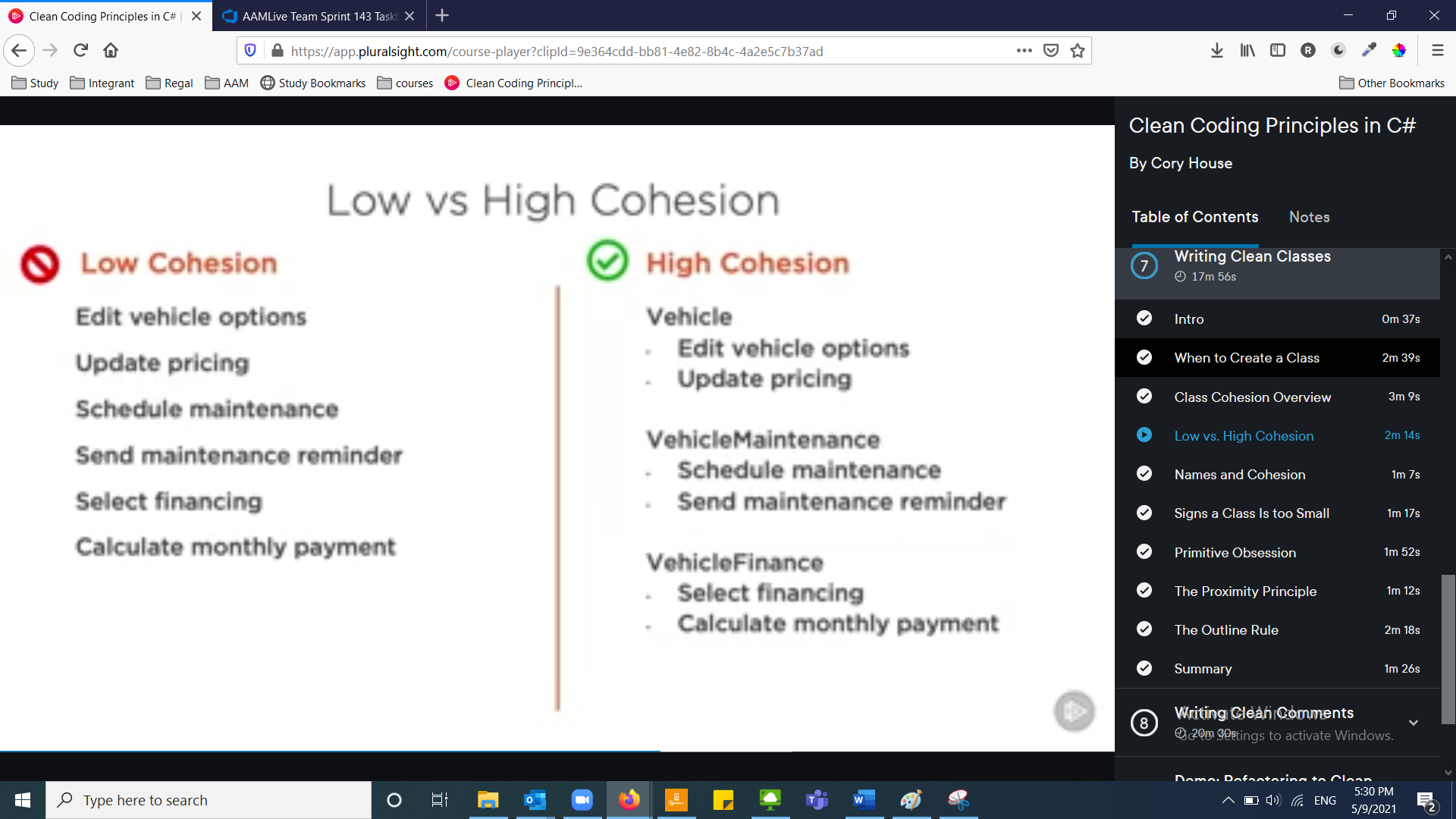
Logging the error sometimes isn’t enough, you need to prevent other steps that are reliant on the try block success

# Classes

When to create a class

* Based on real-life or business entity or model like product or car
* Reuse
* Group related parameters or properties together

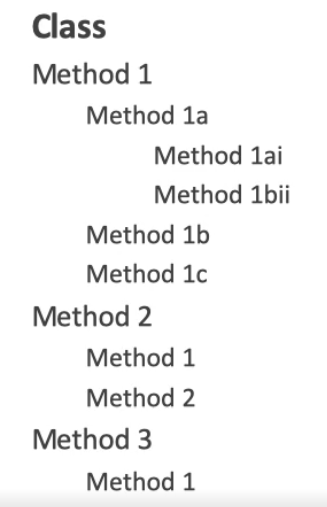
Low cohesion classes: the methods are standalone and don’t interact with the properties inside the class, and less likely to be reused



## The outline rule

Classes and methods calling should be similar to headings in a article. The code should be split Into smaller methods

Which means that reading through a function should read like steps or related method calls. That way makes it easier to read the code at any level of abstraction. If the user wants to read higher level only or dig deeper into one of the levels



For example:

RegisterUser()

ValidateUserData()

ValidateUserName()

ValidateEmail()

CheckUserExists()

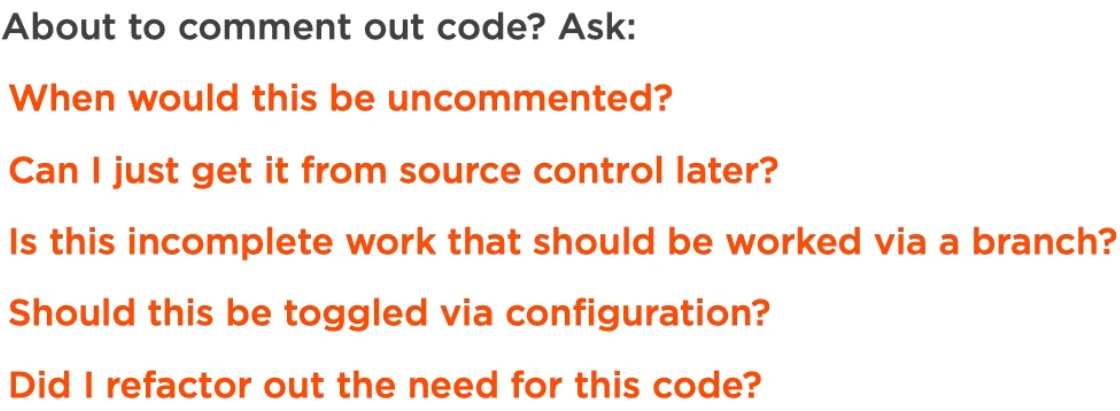
# Comments

Things to avoid:

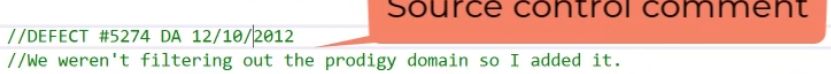
* Redundant comments: useless and already can be understood from context



* Zombie code: commented code



* Source code comments like who changed the line and when



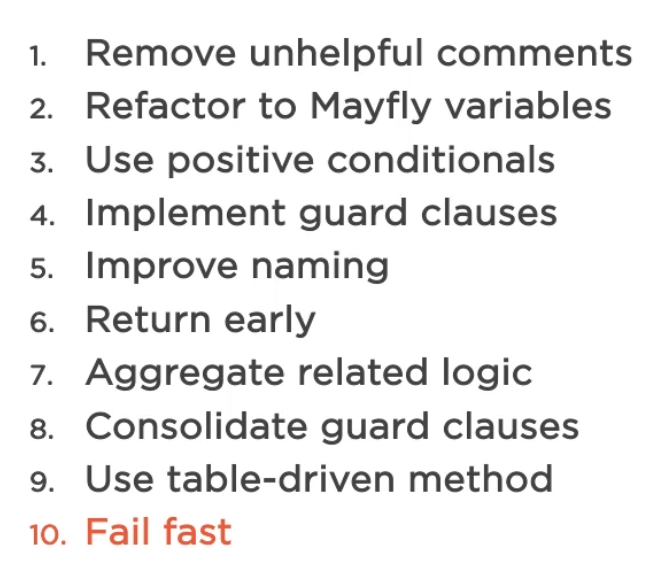
Clean comments:

* TODO
* Summary: describes higher level of what the class or function does

For ex: //generates custom newsletter emails

* Documentation: //for documentation see: http.fdf

# Steps to refactor unclean code



# When to refactor

1. When you need to work with the code:

You shouldn’t refactor code that no one uses or will use, like if you are migrating to a new technology, no need to refactor the old code.

However if you are just changing one line, no need to refactor multiple methods in the code!

1. Difficult to comprehend or change
2. Add tests before refactoring

Leave the code a little better than you found it