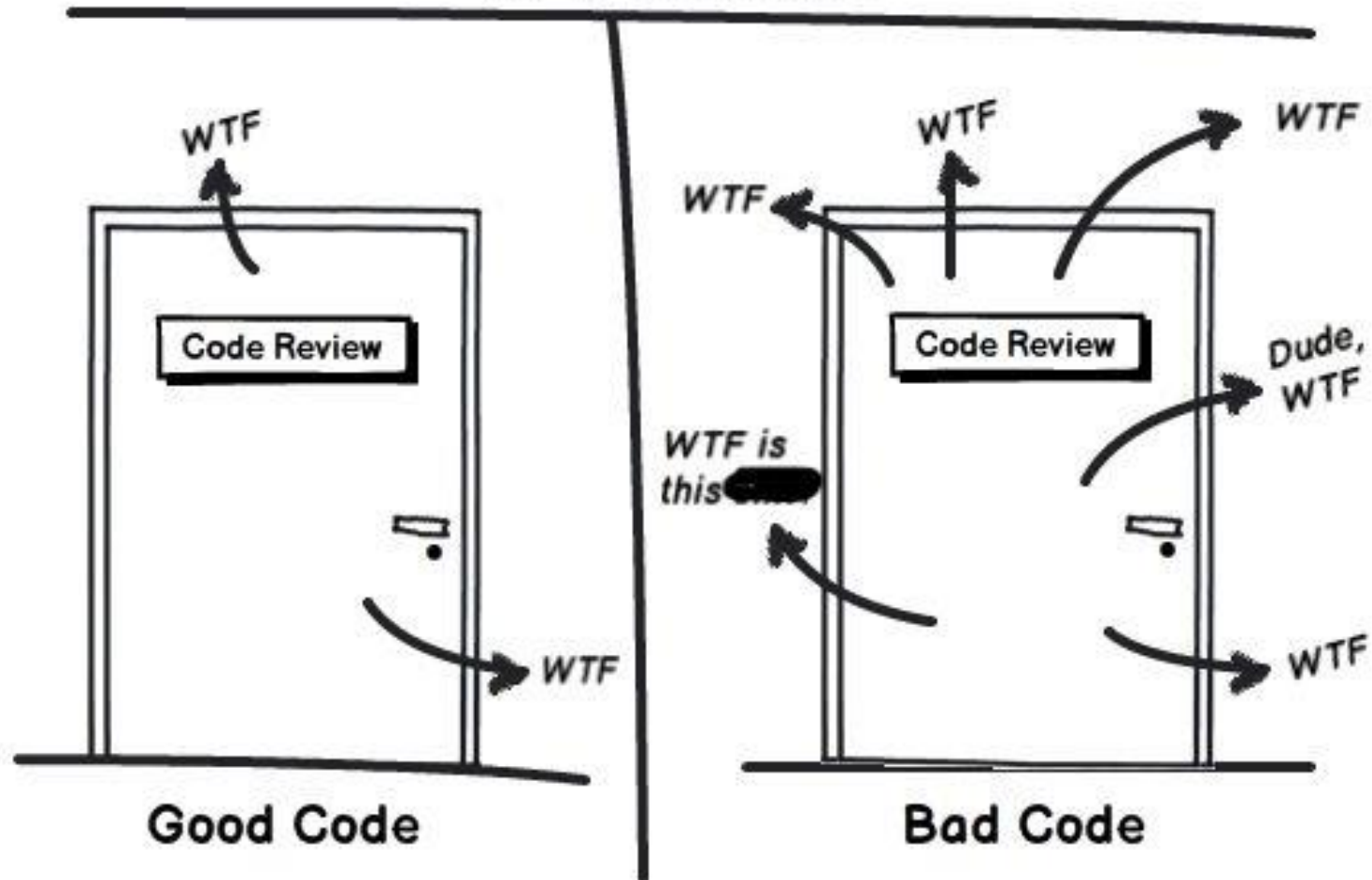


Module 04:

"Code Better C#"



Code Quality Measurement: WTFs/Minute



Agenda

- ▶ Introduction
- ▶ **Clean Code**
- ▶ Exception Do's and Don'ts
- ▶ Code Complexity
- ▶ Curing Primitive Obsession
- ▶ Summary



What is Clean Code?

- ▶ Make code readable
- ▶ Reflect intent
- ▶ Express yourself in code
- ▶ Originally:
 - Published set of rules
- ▶ Modern interpretation:
 - Subjective set of recommendations
 - Broadly accepted as becoming technology-specific

"Even bad code can function. But if code isn't clean, it can bring a development organization to its knees"

- Clean Code by Robert C. Martin



CBC#1: Early Return Principle

- ▶ Turn failed guard clauses into quick returns
- ▶ One of the steps to avoid deeply nested structures



CBC#2: Use Nonnullable Reference Types

- ▶ (Non)nullability of reference types make C# powerfully expressible
 - Use it! Always..! (All the time! No buts... 😊)
- ▶ This will actually remove quite many guard clauses altogether
- ▶ Prefer using the **required** keyword instead of a multiple overload of constructors
 - The **init** accessor is your friend



CBC#3: Name Booleans Positively

- ▶ Always name boolean variables, fields, and methods as
 - **Is**Xxx
 - **Has**Yyy
- ▶ Use positive version of boolean proposition
 - Better for complex boolean expressions
 - Avoids double negatives



CBC#4: Merge Multiple If-statements

- ▶ Enhance readability and conciseness by collapsing
 - If-statements
 - Boolean expressions



CBC#5: Replace Boolean Expressions with Descriptive Methods

- ▶ Introduce more variables and methods if necessary
 - Provide "comments in code"
 - Are searchable
 - Minimize need for actual textual comments



CBC#6: Use LINQ for Conciseness

- ▶ Always prefer LINQ for precision, reusability, and clarity
- ▶ Don't worry about micro-optimizations...!



CBC#7: No Magic Numbers or Strings

- ▶ Replace magic numbers with constants
- ▶ Replace magic strings with enums



CBC#8: Prefer String Interpolation over Concatenation

- ▶ Type-safe and more performant alternative to concatenation
- ▶ Use raw string literals for structured strings



CBC#9: Express Fields and Primary Constructors

- ▶ In general:
 - Use **_xxx** for fields and **xxx** for variables
- ▶ In primary constructors
 - Use **_xxx** when captured
 - Use **xxx** when not captured.



CBC#10: Embrace Pattern Matchings

- ▶ Readability in syntax duality
 - Object initializers create
 - Pattern matchings check



Agenda

- ▶ Introduction
- ▶ Clean Code
- ▶ **Exception Do's and Don'ts**
- ▶ Code Complexity
- ▶ Curing Primitive Obsession
- ▶ Summary



CBC#11: Throw Custom Exceptions

- ▶ Never throw generic Exceptions!
- ▶ At least throw specific exception
- ▶ But prefer throwing custom exceptions
 - Can use contextual information in Exception, e.g. OrderId
 - Can match on such information in
 - Exception when clauses
 - Pattern matching



Lippert's Exception Taxonomy

- ▶ Eric Lippert classifies exceptions into four exception categories
 - ▶ Fatal
 - ▶ Boneheaded
 - ▶ Vexing
 - ▶ Exogenous
- ▶ Stephen Cleary expands of this classification in
 - <https://blog.stephencleary.com/2011/03/exception-types.html>



Fatal Exceptions

- ▶ **Definition:** Fatal exceptions are *not your fault*, and you *cannot sensibly clean up from them*.
- ▶ **Examples:** Out of memory, thread aborted.
- ▶ **Resolution:** Don't catch; let them crash the program.
- ▶ **Design:** Don't ever throw fatal exceptions directly.



Boneheaded Exceptions

- ▶ **Definition:** Boneheaded exceptions are *violations of the API*, and are *bugs in your code*.
- ▶ **Examples:** Argument is null, index out of range.
- ▶ **Resolution:** Don't catch; fix them in the code.
- ▶ **Design:** Use code contracts for boneheaded exceptions; do not document the specific exception type.



Vexing Exceptions

- ▶ **Definition:** Vexing exceptions are due to *bad design decisions*, thrown in *non-exceptional situations*.
- ▶ **Examples:** Parsing errors.
- ▶ **Resolution:** Avoid calling vexing functions; if not possible, catch the vexing exception.
- ▶ **Design:** Don't ever throw vexing exceptions.



Exogenous Exceptions

- ▶ **Definition:** Exogenous exceptions are from *unpredictable, external influences*.
- ▶ **Examples:** File not found, resource already in use.
- ▶ **Resolution:** Always catch and handle.
- ▶ **Design:** Throw exogenous exceptions as necessary; document the specific exception type.



Alternative: Result Object Pattern

- ▶ **Some** people advocate the Result Object Pattern as an alternative to using exceptions
- ▶ Pattern:
 - Create a return type to be returned from method
 - Contains
 - Result type (when processing was success)
 - Error indication (when processing was unsuccessful)



Agenda

- ▶ Introduction
- ▶ Clean Code
- ▶ Exception Do's and Don'ts
- ▶ **Code Complexity**
- ▶ Curing Primitive Obsession
- ▶ Summary



Code Metrics Values in Visual Studio

- ▶ Visual Studio can generate code metrics data to measure complexity and maintainability of your code
- ▶ Main metrics are:
 - Maintainability Index
 - Cyclomatic Complexity
 - Depth of Inheritance
 - Class Coupling
- ▶ Additionally:
 - Lines of Source Code
 - Lines of Executable Code
- ▶ <https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022>



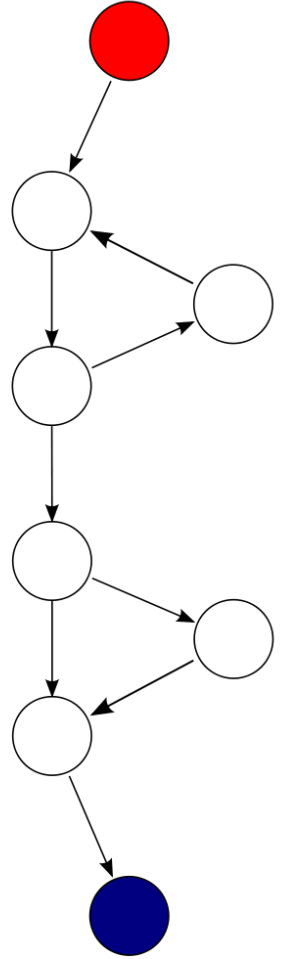
How to Generate Code Metrics Values

- ▶ Generate code metrics in Visual Studio via
 - **Analyze > Calculate Code Metrics**
- ▶ Alternatively;
 - Enable .NET code quality analyzers in Solution Explorer or **EditorConfig** file
 - Command line



Cyclomatic Complexity

- ▶ Measures the amount of decision logic in a source code function
 - ~ number of paths through a method
- ▶ Influenced by boolean operators, **if**, **switch**, **while**, ...
- ▶ Lower is good
- ▶ Higher is bad
 - indicates more tests needed to cover it
 - indicates harder to maintain



Maintainability Index

- ▶ Calculates an index value between 0 and 100 that represents the relative ease of maintaining the code
 - ~ Complexity per program parts

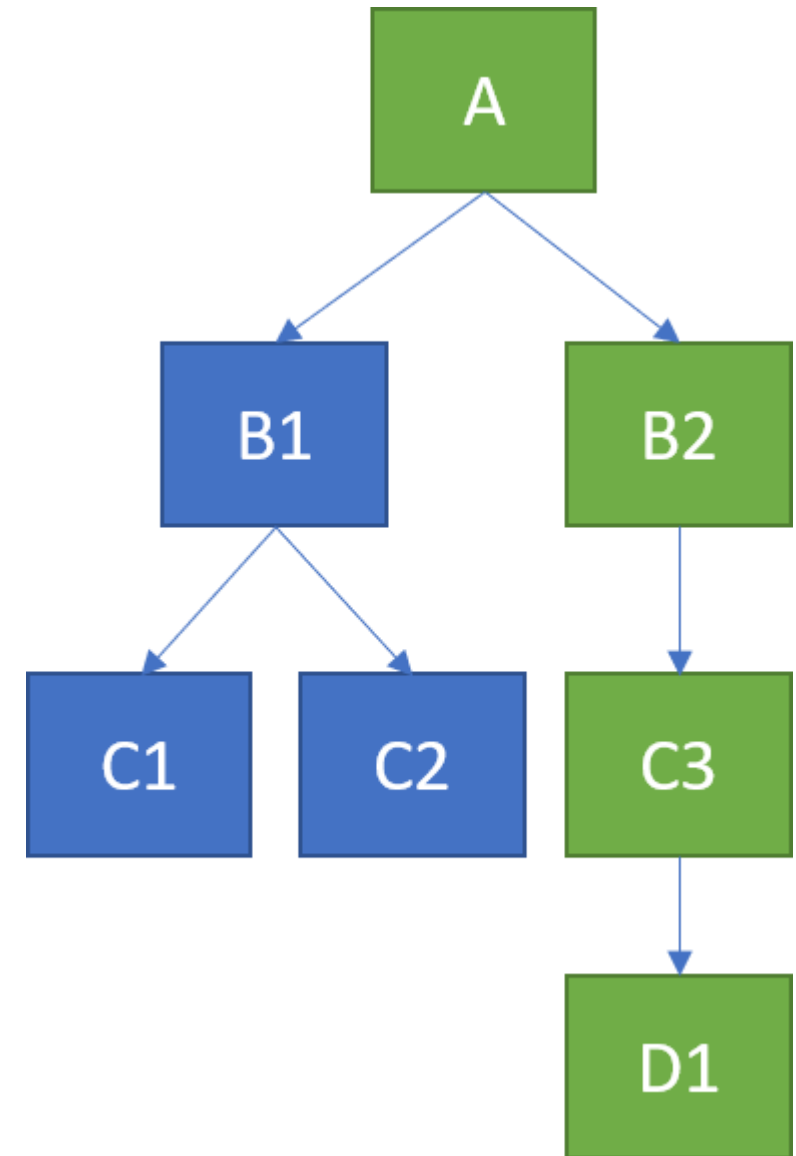
Size of program's size in "bits"

$$\text{Maintainability Index} = \text{Max}(\begin{aligned} &0, \\ &(171 - 5.2 * \ln(\text{Halstead Volume}) \\ &\quad - 0.23 * (\text{Cyclomatic Complexity}) \\ &\quad - 16.2 * \ln(\text{Lines of Code}) \\ &)*100 / 171 \end{aligned})$$

- ▶ Higher is good
 - 0-9 10-19 20-100

Depth of Inheritance

- ▶ Measures the depth of longest inheritance chain from root to leaf classes
- Lower
 - implies less complexity but also the possibility of less code reuse through inheritance.
- Higher
 - implies more potential for code reuse through inheritance but also higher complexity with a higher probability of errors in the code.



Class Coupling

- ▶ Measures the coupling to unique classes through
 - parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decoration
- ▶ ~ how many “custom” types the class uses
- ▶ Lower is good
 - High cohesion and low coupling is preferred
- ▶ Higher is bad
 - Difficult to reuse and maintain due to interdependencies on other types



Agenda

- ▶ Introduction
- ▶ Clean Code
- ▶ Exception Do's and Don'ts
- ▶ Code Complexity
- ▶ **Curing Primitive Obsession**
- ▶ Summary



What is Primitive Obsession?

- ▶ "Primitive Obsession" ~ Code smell where primitive data types only are used for modelling your domain classes and objects
- ▶ String is not your friend!
- ▶ Type-safety is lost
- ▶ Validation logic is scattered and/or duplicated



Doctor's Prescriptions

- ▶ Cure primitive obsession by
 - Create value objects
 - Validation
 - Create strongly-typed id types
 - Type-safety
 - Refactor scattered algorithms into Strategy objects
- ▶ Consider reusability and/or "frameworks" for value objects and types



Summary

- ▶ Introduction
- ▶ Clean Code
- ▶ Exception Do's and Don'ts
- ▶ Code Complexity
- ▶ Curing Primitive Obsession



