Takeaway
Do’s and Don'ts

Xavier Morera

Helping .NET developers create amazing applications

@xmorera / www.xaviermorera.com / www.bigdatainc.org

Any fool can write code that a computer can understand

Good programmers write code that humans can understand

Write clean code

Follow best practices

Clean Code Principles

Make your code simple

Avoid unnecessary complexity

KISS

Create only functionality that you need

Don't solve problems you currently don't have

YAGNI

Reduce repetition of code

Create functions for duplicate code

DRY

Clean Code Principles

Design types according to their functionality, rather than nature

Favor composition over inheritance

Separate an application into multiple units

UI, BL, DA...

Separation of concerns

Set of principles

SOLID

SOLID

Single-responsibility

Open-closed

Liskov substitution

Interface segregation

Dependency inversion

SOLID

Each class should have only one responsibility

Single-responsibility

Open to extension but closed to modification

Open-closed

Subclass object should be substitutable for its superclass

Liskov substitution

SOLID

Only declare methods
that are required

Interface segregation

High-level modules should not
depend on lower-level modules

Dependency inversion

Always write self-documenting code!

Coding Conventions and Guidelines

Set of rules used for coding in a programming language

Recommend style, practices, and methods for writing code

Coding convention

Provides general suggestions regarding the coding style to improve understandability and readability of the code

Guideline

Coding Conventions and Guidelines

Naming

Layout

Comments

Naming Conventions

Use representative names for entities

- x → address → trailaddress

Different naming conventions

- PascalCase, camelCase
- Depends on the entity
  - Class, interface, public/private variable...

Naming Conventions

Private/internal fields and method parameters

camelCase

Class, record, struct, interfaces, public members, and positional records

PascalCase

*** Don't rename auto-generated names**

Code Layout

Spend more time reading than writing code

Layout conventions

- Smart indenting
- Four-character indents
- Tabs saved as spaces

Only one statement and declaration per line

Code Layout

If continuation lines are not indented automatically

- Indent them one tab stop (four spaces)

Add at least one blank line between property definitions and methods

Use parentheses to make clauses in an expression apparent

Commenting Conventions

Place the comment on a separate line

- Not at the end of a line of code (maybe)

Begin comment text with an uppercase letter

End comment text with a period

Insert one space between the comment delimiter and comment text

Language Guidelines

String interpolation

StringBuilder

Implicitly typed variables

Arrays

Func and Action

new

Short-circuit operators

using

Object initializers

Static

LINQ Do’s

Meaningful names

Use aliases

Rename properties that might be ambiguous

Use implicit typing

Align query clauses under the from clause

Use where **clauses before other query clauses**

Clean Methods and Classes

Clean Methods and Clean Classes

Code should be understood by humans

- Favor readability

Make it correct, make it clear, make it concise, make it fast

- In that order

Class naming guidelines

PascalCasing

Use nouns

Be specific, single responsibility, avoid abbreviations

One class per file

Ordering

- Fields, properties, and then methods

Methods

Help organize functionality

- Never create “arrow code”
- Use guard clauses instead

Follow naming guidelines

- Use verbs that indicate action
- Avoid generic verbs
- Get only with constant time complexity
- Boolean methods
  - Start with is, are, was, were…

Refactoring

Process of restructuring existing code without changing the external behavior

Intended to improve the design, structure, and/or implementation of the code

Refactoring Objective

Create methods that are

- Concise
- Easy to understand
- Maintainable
- Upgradeable

Refactoring Techniques

Extract method

Extract variable

Inline temp

Composing methods

Move methods and fields

Extract class

Move features between objects

Encapsulate fields and collections

Replace strings for Enums

Replace magic numbers with symbolic constants

Change type field with class

Organize data

Refactoring Techniques

Consolidate conditional expression

Consolidate duplicate conditional fragments

Decompose conditional

Simplify conditional expressions

Preserve whole object

Introduce parameter object

Split and merge methods

Simplify method calls

Refactoring Techniques

Moving functionality along class inheritance hierarchy

Pull up or down fields and methods

Extract interface, subclass, or super class

Dealing with generalization

Rename entities

Use the IDE functionality or manually, but with care

Rename

Creating Testable Code

Writing unit tests
is highly recommended

Even mandatory in some cases

The objective is to make sure functions provide correct output

Unit Test

Unit test is a method

- [TestMethod()] attribute

Follows the AAA pattern

- Arrange, prepare for the test
- Act, invoke method being tested
- Assert, validate the result

Assert class used for verification

- AreEqual, AreNotEqual, Fail...

Characteristics of a Good Unit Test

Fast

Isolated

Repeatable

Self-checking

Timely

Best Practices for Writing Unit Tests

Use the AAA pattern

- Only one act
- Without complex logic

Name that explicitly expresses intent, scenario, and expected behavior

Use simple input

Avoid magic strings

Thanks for watching!

@xmorera

www.xaviermorera.com

“What you learn is yours for life.”