AGENDA

—

About Clean Code

Coding Standards

Comments

Good names

Classes

Methods

cegeka

WHAT DO OTHER PEOPLE THINK A PROGRAMMER DOES?

WHAT DOES A PROGRAMMER ACTUALLY DO?

cegeka

MYTH VS REALITY

WHAT PEOPLE THINK PROGRAMING MEANS?

WHAT PROGRAMMING ACTUALLY MEANS?

PROGRAMMERS SPEND THEIR TIME WRITING CODE

PROGRAMMERS:
- READ CODE
- WRITE CODE
- DELETE CODE

NO

YES

cegeka

CLEAN CODE

WHY SHOULD WE USE IT AND WHY WE DON'T USE IT ENOUGH?

WHAT IS A CLEAN CODE?

CLEAN CODE

imgflip.com

@coolprobn

„Any fool can write code
that a computer can understand.

Good programmers write code
that humans can understand"

Martin Fowler

cegeka

CLEAN CODE – WHY?

The way a software developer writes code is his business card.

RESOURSES

- Clean Code A Handbook of Agile Software Craftsmanship (Robert C. Martin)
- Code Complete: A Practical Handbook of Software Construction (Steve McConnell)
- The Pragmatic Programmer (Andrew Hunt, David Thomas)
- https://app.pluralsight.com/library/courses/writing clean code humans/

cegeka

WHY IT'S IMPORTANT TO WRITE CLEAN CODE?

WHY IT'S IMPORTANT TO WRITE CLEAN CODE?

- **Code is easier to read and understand**
  - By you
  - By others
  - After months or years

- **Code is easier to change**
  - Less possible bugs from misunderstanding code
  - Easier maintenance

- **Easier to integrate new team members**

CLEAN CODE

THE FAST WAY

imgflip.com

cegeka
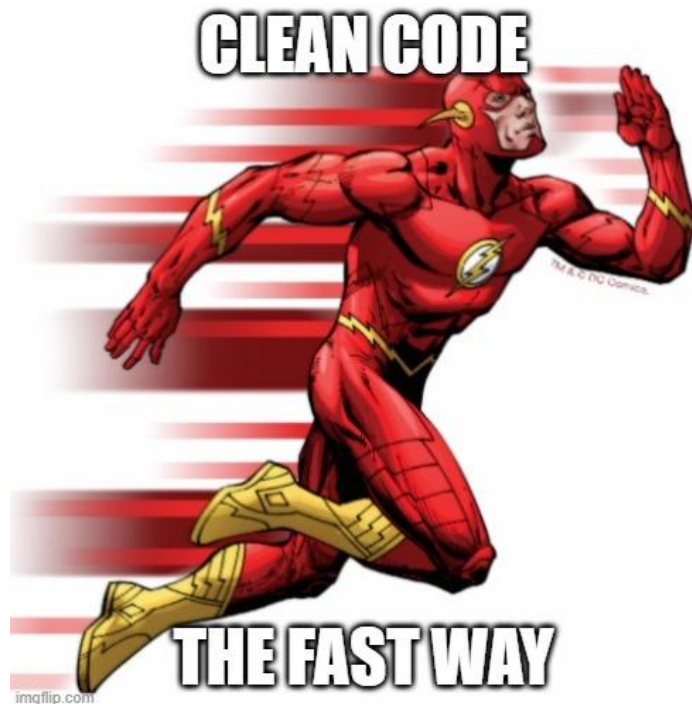
WHY DON'T WE WRITE CLEAN CODE?

BEGINNER,
NOT ENOUGH EXPERIENCE,

HURRY,
DEADLINES,
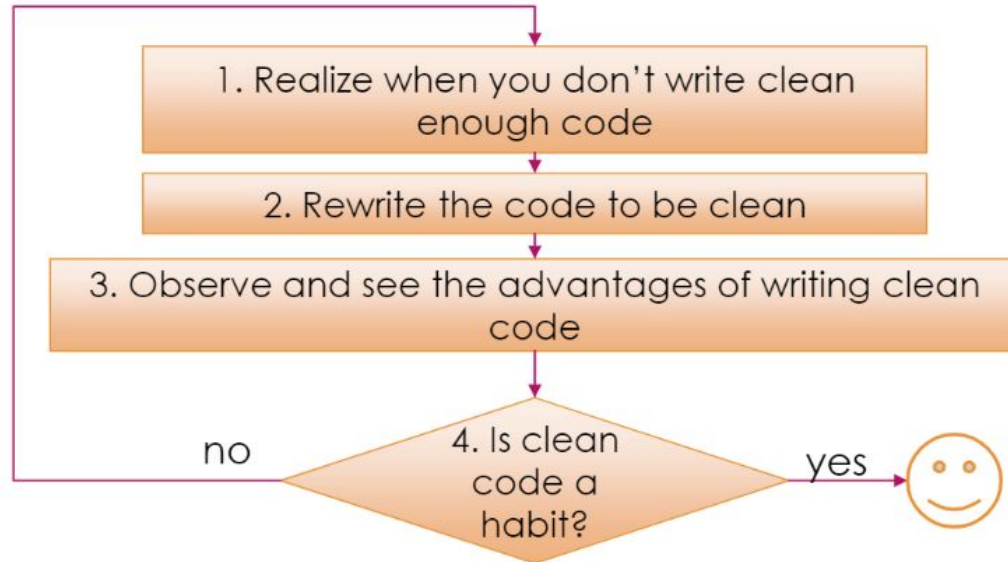PRESSURE

TEST CODE,
FREQUENT CHANGES
NO LONG-TERM VISION

LAZYNESS,
INDIFERENCE,
UNPROFESSIONALISM

LATER = NEVER

cegeka

HOW DO WE LEARN TO WRITE CLEAN CODE?

It's not enough to know how clean code looks like: It's like tasting food – we can say if it's good or not, but that doesn't mean we know how to prepare it to be good.

1. Realize when you don't write clean enough code

2. Rewrite the code to be clean

3. Observe and see the advantages of writing clean code

4. Is clean code a habit?

no

yes

cegeka

HOW CAN YOU MAKE YOUR THE CODE CLEANER?

CODE IMPROVEMENT TECHNIQUES

- Read code and compare clean code with dirty code
- Verbalize code
- Code Review
- Pair Programming
- Feedback
- Try to unit test it

SELF DOCUMENTING CODE

- Ideally, code should be written as a book and should be easily read as a book
- Should easily answer "What did the author want to say here?" = INTENT

- Well written code is self documented code
- Express intent clear
- Abstractization = shows only the important aspects and hides/ ignores unnecessary details
- Good formatting readability
- Favour good code over comments

cegeka

CODING STANDARDS

Coding Standards

NOT SURE IF I WRITE CONFUSING CODE OR MY CO-WORKERS ARE FREQUENTLY CONFUSED
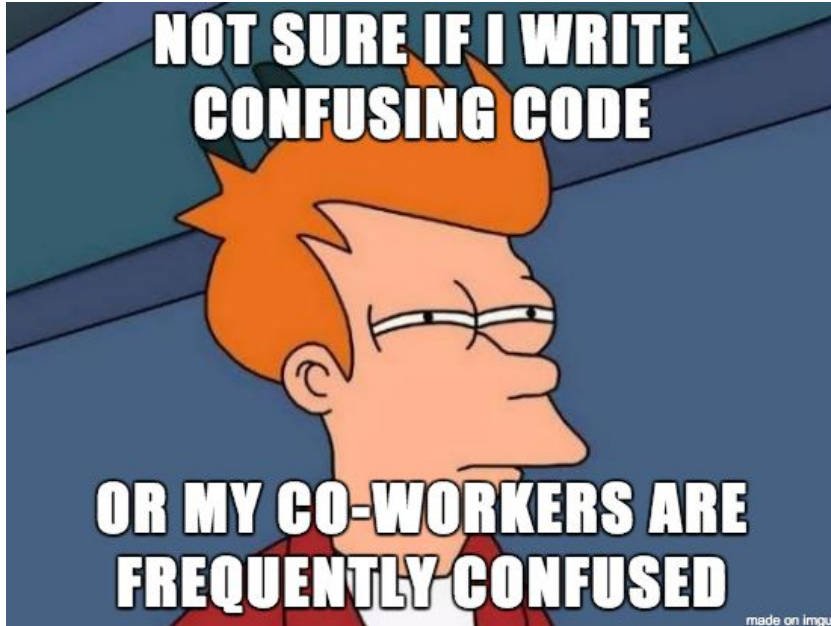
- A coding standard is a set of recommendations about how to write code and what are the naming and organizing conventions of code.
- Each programming language has its own coding standards, usually recommended by the authors of language
- Microsoft recommendations: https://msdn.Microsoft.com/en-us/library/ff926074

- Coding standards can change in time and depend on the language

  Hungarian notation is obsolete

CODING STANDARDS FOR TEAMS

☐ Each team should adopt a standard that best fits their needs and is appropriate to the programming language they use

☐ Why?
- ✔ Better code quality
- ✔ Faster development and maintenance
- ✔ Easier and faster integration for new team members
- ✔ Easier to work on someone else's piece of code
- ✔ Same writing style for the entire team – proof of professionalism

Tools> ReSharper, FXCop, Sonar, etc

cegeka

CAMEL CASE

- C# uses Camel Case for naming variables, methods, classes, etc. and Microsoft recommends the same coding standards for developers that use C#

- Camel Case means that every different word in a name should start with the first letter capitalized. This makes reading easier for the brain, because the words are easier to identify and separate inside the long sequence of letters

Casing type	Upper Camel Case	Lower Camel Case
Used for naming	- Classes, Interfaces, Structs - Methods - Properties - Public Fields	- Private/protected fields - Local variables

Examples	`public void GetStudentById(int id)`
	`public class Student`
	`public string FirstName {get; private set;}`
		`private List<Student> students;`
		`string nextIndex;`

COMMENTS

COMMENTS SHOULD OPTIMIZE READING AND UNDERSTANDING CODE,
BUT SHOULD NOT COMPENSATE FOR BAD CODE.

WHEN TO USE COMMENT?

- They have to be useful and written only when they are needed because the code itself is not enough to understand
- Up to date and reflecting what the code does
- Brief and explicative
- Clear, not misleading
- Visual Studio – Task list (todo, hack, undone)

cegeka

WHEN NOT TO USE COMMENT?

- Comments should not compensate for bad code. You should just rewrite that code
- Redundant – not DRY
- Zombie code
- Too much text that can be easily understood from code

WHERE TO PLACE COMMENTS?

- Always before the code block
- Indented at the same level as the code
- Space after // or /*

Good
```
// list must contain exactly 3 elements
bool isValid = (cases.size() == 3);
```

Evil
```
bool isValid = (cases.size() == 3);//list must contain 3 elements
```

cegeka

COMMENTS - ENCAPSULATION

- Easier to read and understand
- Each class should be responsible for it's own internal logic
- Encapsulation – Don't expose then needed

Evil
```
// checks if the employee is eligible to get extra vacation days
if ((employee.Flags == WorkSchedule.FullTime) && employee.YearsOfExperience > 10)
```

Good
```
if (employee.IsEligibleForExtraVacationDays)
```

COMMENTS - USELESS

- ❑ Funny or apology, but useless
- ❑ Don't answer " What did the author want to say?"

```
Evil

// if you made it this far without having a nervous breakdown
// congratulations! Go get yourself a beer!

// Magic. Do not touch

// When I wrote this, only God and I understood what I was doing.
// Now, only God knows

// Bug #1234

// Sorry, this crashes and I don't know why

// Here starts the hack - See John
```

cegeka

COMMENTS - REDUNDANT

☐ Comments that state the same as the line of code itself
☐ Methods that have good names, don't need description that is actually the same as the method name (DRY)

```
Evil

// returns true
return true;

/// <summary>
/// Counts the females
/// </summary>
/// <returns></returns>
public int CountFemales()
```

cegeka

COMMENTS - MISLEADING

- Compensates bad code
- Does not reflect reality

Evil

```
public int GetRandomNumber()
{
    // chosen by rolling die
    return 4;
}
```

cegeka

COMMENTS – ZOMBIE CODE

- Unused code should be commented – but removed
- You have the code under source control
- Makes reading more difficult and distracts attention from the active code
  Clutters reading code with "noise code"
  Search results might return zombie code as well
- Creates ambiguity – was the code commented by mistake?

cegeka

COMMENTS – USEFUL

Good

```
string searchString = "test";
foreach (Patient patient in this.patient)
{
    // the same string
    if (patient.Name.CompareTo(searchString) == 0)
    {
        return patient;
    }
}
```

Good

```
// Pattern explanation:
// - "^(?:[\w]\:|\\)" -- Begin with x:\ or \\
// - "[a-z_\-\s0-9\.]" -- valid characters are a-z| 0-9|-|.|_
// - "(txt|gif|pdf|doc|docx|xls|xlsx)" -- Valid extension
// Matches:
// \\192.168.0.1\folder\file.pdf
// c:\myfolder\abc abc.docx
// Non-Matches:
// \\192.168.0.1\folder\\file.pdf
// c:\myfolder\another_folder\ab*c.v2.docx
// file.xls
string filePattern = @"^(?:[\w]\:|\\)(\\[a-z_\-\s0-
9\.]+)+\.(txt|gif|pdf|doc|docx|xls|xlsx)$";
```

cegeka

BRACES – BLOCKS OF CODE

☐ The blocks of code in C# are marked with curly braces {and}
☐ The braces should be each on a separate line, without any other command/comment
☐ The braces should be specified even it there is only 1 instruction in that code block

```
Evil
public Student GetStudentById(int id) {
    return students.SingleOrDefault(s => s.Id == id);
}

if (x > 0)
    Console.WriteLine("Positive");
else Console.WriteLine("Negative or zero");
```

```
public Student GetStudentById(int id)
{
    return students.SingleOrDefault(s => s.Id == id);
}

if (x > 0)
{
    Console.WriteLine("Positive");
}
else
{
    Console.WriteLine("Negative or zero");
}
```

cegeka

BRACES – CONDITTIONS

Specify the condition between round parenthesis ()


```
Evil
if x > 0
{
    Console.WriteLine("Positive");
}
```

```
Good
if (x > 0)
{
    Console.WriteLine("Positive");
}
```

GOOD NAMES

"WHAT'S IN A NAME? THAT WHICH WE CALL A ROSE"

GOOD NAMES

- The names you give to classes, variables, methods etc. are essential for understanding code. Even if it takes some time to "baptize" properly a variable, this will eventually save time in the future because it will be easier to understand what the variable us/does just by ranking a glance at it.

- Let's suppose you get a pet. What name would you give him?
  - Dog1
  - myDog
  - D
- Each name should represent what the variable/method/class does

cegeka

MEANINGFUL NAMES

- Give good names instead of using comments to compensate for the bad code

```
Evil
// elapsed time in days
int d;
```

```
Good
int daysSinceModification;
int fileAgeInDays;
```

MEANINGFUL NAMES - DEMO

☐ Give good names to understand easier the code

Evil

```csharp
public double Compute(List<int> a)
{
    double x = 0;
    int y = 0;
    foreach (int nr in a)
    {
        x += nr;
        y++;
    }
    if (y == 0)
    {
        return 0;
    }
    else
    {
        return x / y;
    }
}
```

Good

```csharp
public double ComputeAverage(List<int> numbers)
{
    double sum = 0;
    int howManyNumbers = 0;

    foreach (int number in numbers)
    {
        sum += number;
        howManyNumbers++;
    }

    if (howManyNumbers == 0)
    {
        return 0;
    }
    else
    {
        return sum / howManyNumbers;
    }
}
```

Good

```csharp
public double ComputeAverage(List<int> numbers)
{
    return numbers.Count == 0 ? 0 : numbers.Average();
}
```

cegeka

BAD NAMES

- If the objects represent different things, then their names should reflect the difference
- Don't add prefixes like: the, my, a
- Don't add numeric suffixes

```
Evil

string theString;
string myString;
string aString;

string string1;
string string2;
string string3;
```

- Avoid too general names or prefixes

```
Evil

Utility
Common
...Manager
...Processor
...Info
...Data
```

cegeka

BAD NAMES - REDUNDACIES

- Avoid redundant prefixes/suffixes

Evil
```
string stringName;
decimal moneyAmount;
```

Good
```
string name;
decimal money;
```

GOOD NAMES – PREFIXES & SUFFIXES

Avoid prefixes/suffixes that don't help you make clear differences

Evil
```
class StudentInfo
class StudentData
```

Good
```
class StudentMedicalData
class StudentSchoolData
```

Evil
```
List<Account> theList
List<Account> aList
```

Good
```
List<Account> usedAccounts
List<Account> newAccounts
```

cegeka

GOOD NAMES – SYMMETRY

—

☐ Use symmetrical naming for opposing states or prefixes

Evil	Good
on/disable	on/off enable/disable
quick/slow	fast/slow
lock/open	lock/unlock close/open
slow/max	min/max

cegeka

GOOD NAMES

 Avoid very similar names: itemlnList/itemslnList

 Be consistent when giving names (use GetList all the times, not GetList sometimes and other times RetrieveList)

 Don't be a cheapstake with names – give variable names longer than a letter, because each variable has a significance

 You can use "I" and "j" for iterating through a list, but it the iterator spreads over a few tens of lines, maybe it would be better you rename the "i" and "j" so that you don't have to scroll back and see what each does.

cegeka

VERBALIZING CODE

- Avoid names that cannot be pronounced
- Avoid abbreviations
- It will be verry difficult to explain to someone else what it does

Evil

```
class DtaRcrd102
{
    private DateTime genymdhms;
    private DateTime modymdhms;
    private String pszqint = "102";
}
```

Good

```
class Customer
{
    private DateTime generationTimestamp;
    private DateTime modificationTimestamp;
    private String recordId = "102";
}
```

cegeka

GOOD NAMES IN C#

- Avoid names that have underscore separators between the words
- Exception: the event handlers, that are generated automatically have controlName_eventName


```
Evil

int next_Index;
void get_student_by_id(int id)
```

```
Good

int nextIndex;
void GetStudentById(int id)

void btnSave_Click(object sender, RoutedEventArgs e)
```

CLASSES

WHAT IS A CLASS?

 A class is the pattern (template) that defines a type of objects and an instance (object) is one element created using that pattern (template)

 A class contains:
State = fields/properties = nouns
Behavior = methods = verbs

cegeka

CLASSES VS OBJECTS

A class

The gingerbread cutter is the pattern for a gingerbread man.

It defines the common elements for all gingerbread men:

- 1 head, 1 mouth, 2 eyes
- 2 hands, 2 feet, 2 buttons

But you can't eat it

Objects (instances) of a class

Each gingerbread man is created using tha cutter and all gingerbread men have the same elements defined by the pattern:

- 1 head, 1 mouth, 2 eyes
- 2 hands, 2 feet, 2 buttons

They can have some different properties

- Coloring
- Taste

cegeka

CLASS NAMES

- NOUNS at Singular
- As specific as possible
- Upper Camel Case
- Same for structs

Evil
```
class Students
class student
class ReadStudents
```

Good
```
class Student
class StudentsFileReader
```
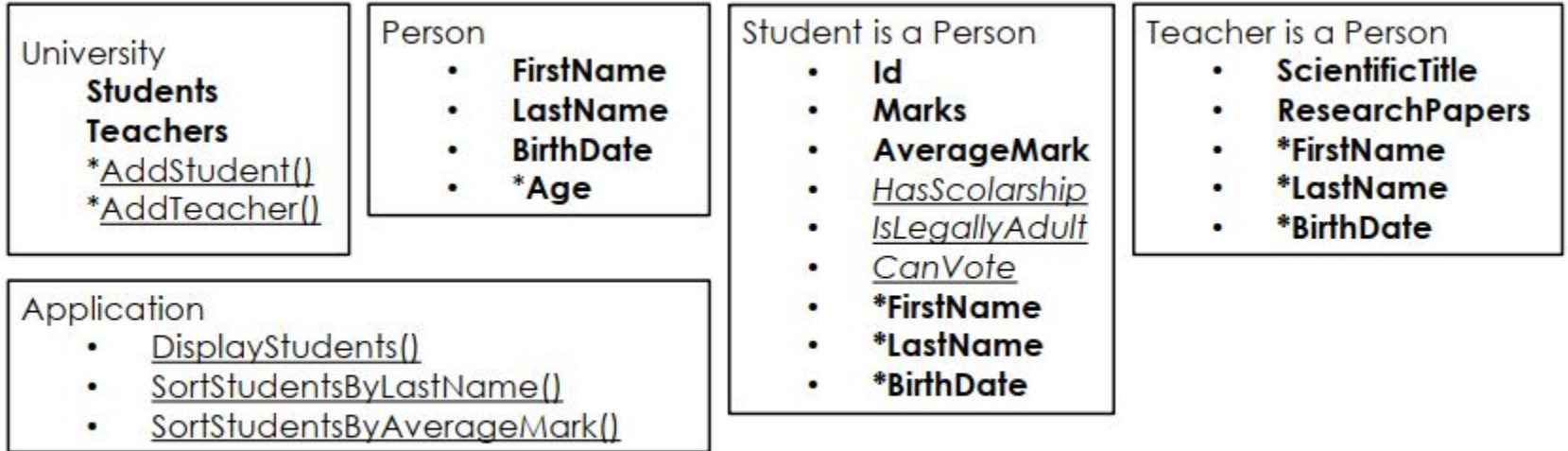
cegeka

EXTRACTING CLASSES FROM TEXT - EXAMPLE

- A university has two types of persons: students and teachers.
- Each person has a first name, a last name and a birthdate.
- Each student has an identifier and some marks
- The application should display for each student the average of the marks, it he has a scholarship, if he is legally an adult and if he can vote.
- The application shoul allow to sort the students by last name or average mark
- In a similar manner, each teacher has a scientific title and can publish research papers.

IDENTIFY STATE AND BEHAVIOUR

- A **university** has two types of **persons**: **students** and **teachers**.
- Each **person** has a **first name**, a **last name** and a **birthdate**.
- Each **student** has an **identifier** and some **marks.**
- The **application** should display for each **student** the **average** of the **marks**, it he has a scholarship, if he is legally an adult and if he can vote.
- The **application** should allow to sort the **students** by **last name** or **average mark.**
- In a similar manner, each **teacher** has a **scientific title** and can publish **research papers.**

cegeka

CLASSES AFTER ANALYSIS

University
Students
Teachers
*AddStudent()
*AddTeacher()

Person
- **FirstName**
- **LastName**
- **BirthDate**
- ***Age**

Student is a Person
- **Id**
- **Marks**
- **AverageMark**
- *HasScolarship*
- *IsLegallyAdult*
- *CanVote*
- ***FirstName**
- ***LastName**
- ***BirthDate**

Teacher is a Person
- **ScientificTitle**
- **ResearchPapers**
- ***FirstName**
- ***LastName**
- ***BirthDate**

Application
- DisplayStudents()
- SortStudentsByLastName()
- SortStudentsByAverageMark()

cegeka

ENTITIES AND FILES

- Each class/struct/interface/enum should be in a separate file
- The file should have the same name as the class/struct/interface/enum
- The namespace should match the containing folder and should be updated if you move a file from a folder to another.

cegeka

INTERFACE NAMES

⬜ NOUNS at Singular, prefixed with "I"

⬜ As specific as possible

⬜ Upper Camel Case

Evil
```
interface Students
interface iReadStudents
interface iStudent
```

Good
```
interface IStudent
interface IStudentsFileReader
```

cegeka

METHODS

METHOD NAMES

- Have to contain VERBS
- Each method should do one thing and one thing only
- Method names should reflect what the method does
- Method names should be descriptive, so that whoever looks at the name, understand what the method does without having to look at the implementation
- Ask yourself (or a colleague or a rubber duck)
  - What does this method do?
  - Does it do a single thing?
  - Watch out for words like: AND, OR, IF
  - It is does more, then you should split it
  - Is it the right place?
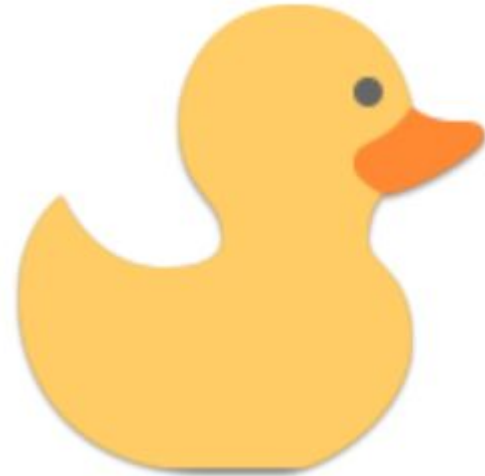
METHOD – SHOULD I SPLIT IT?

☐ Does it fit on your screen?

  Small methods that can be reused

  Ideally not over 20 lines. Never over 100 lines

  Simpler functions can be longer. Complex functions should be shorter.

☐ Do you use comments or white spaces to separate code inside a method?

  You should split the code into smaller methods

☐ Is this method or part of it similar to another method?

  Avoid Duplication – don't reinvent the wheel, just reuse it

☐ Does it have a lot of indentation?

☐ Cyclomatic complexity = there are many distinct paths through a method

☐ Difficult to read and test, more possible bugs

cegeka

METHOD PARAMETERS

—

- Minimize parameters
- 0-2 parameters – Rule of 7
- Group them in a struct/class

Evil

```
private void SendEmail(string username, string password, string email, string message)
{
    // send email
}
```

Good

```
private void SendEmail(User user, string message)
{
    // send email
}
```

cegeka

STRATEGIES FOR METHODS - DEMO

- Return early
  If you know that certain conditions are not valid, return as fast as possible

- Fail fast – Guard clauses
  Guard clauses – conditions at the beginning of the method that assure you have all needed data before continuing
  Guards reduce complexity and indentation
  Guards state from the beginning of the code the conditions that have to be met for the data to be valid

- Variables should have minimum lifespan
  Declare variables only when they are needed, not from the beginning of the method

cegeka

STRATEGIES FOR METHODS

Evil

```csharp
private void btnAddNumbers_Click(object sender, RoutedEventArgs e)
{
    try
    {
        int firstNumber;
        int secondNumber;
        int sum;

        if (string.IsNullOrEmpty(txtFirstNumber.Text) == false)
        {
            firstNumber = int.Parse(txtFirstNumber.Text);
            if (string.IsNullOrEmpty(txtSecondNumber.Text) == false)
            {
                secondNumber = int.Parse(txtSecondNumber.Text);
                sum = firstNumber + secondNumber;
                textBlockSum.Text = sum.ToString();
            }
            else
            {
                throw new Exception("You must specify a value for the input numbers");
            }
        }
        else
        {
            throw new Exception("You must specify a value for the input numbers");
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "An error has occured", MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
```

Good

```csharp
private bool AreInputFieldsMissing()
{
    return (string.IsNullOrEmpty(txtFirstNumber.Text) == false
        || string.IsNullOrEmpty(txtSecondNumber.Text) == false);
}

private int ComputeSum()
{
    int firstNumber = int.Parse(txtFirstNumber.Text);
    int secondNumber = int.Parse(txtSecondNumber.Text);
    return firstNumber + secondNumber;
}

private void btnAddNumbers_Click(object sender, RoutedEventArgs e)
{
    try
    {
        if (AreInputFieldsMissing())
        {
            throw new Exception("You must specify a value for the input numbers");
        }

        var sum = ComputeSum();
        textBlockSum.Text = sum.ToString();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "An error has occured", MessageBoxButton.OK,
MessageBoxImage.Error);
    }
}
```

cegeka

AVOID CODE DUPLICATION - DEMO

☐ DRY Principle = Don't repeat yourself

☐ Avoid code duplication
   Whenever you are tempted to copy-paste code, think if it would not be better to extract a method and use it in a multiple places
   See the common pattern for similar code and extract a method with different parameters for the differences

☐ Disadvantages of duplication:
   More code to maintain
   If you forget to change in all places -> Inconsistencies -> Bugs

cegeka

CODE DUPLICATION

Evil

```csharp
public int CountFemales()
{
    int count = 0;
    foreach (var person in this.persons)
    {
        if (person.Gender == Gender.Female)
        {
            count++;
        }
    }
    return count;
}

public int CountMales()
{
    int count = 0;
    foreach (var person in this.persons)
    {
        if (person.Gender == Gender.Male)
        {
            count++;
        }
    }
    return count;
}
```

Good

```csharp
private int CountByGender(Gender gender)
{
    int count = 0;
    foreach (var person in this.persons)
    {
        if (person.Gender == gender)
        {
            count++;
        }
    }
    return count;
}

public int CountFemales()
{
    return CountByGender(Gender.Female);
}

public int CountMales()
{
    return CountByGender(Gender.Male);
}
```

Better

```csharp
private int CountByGender(Gender gender)
{
    return persons.Count(p => p.Gender == gender);
}

public int CountFemales()
{
    return CountByGender(Gender.Female);
}

public int CountMales()
{
    return CountByGender(Gender.Male);
}
```

cegeka

FIELD NAMES

- Fields represent any type of variables inside a class
- You should make the fields private or protected and make a getter/setter for it if it needs to be visible from outside the class
- According to the access modifier, the casing should be:

Public/internal – UpperCamelCase

Private/protected - lowerCamelCase

Good

```
public class RegistrationBook
{
    private int nextId;
    private List<Person> persons;
}

public class Person
{
    private int id;
}
```

Evil

```
public class StudentList
{
    private List<Student> l;
    private int _next_Id;
}
```

cegeka

STATIC FIELDS

☐ The static fields belong to the class, not to the instances
Static fields are shared between all instances of a class

You don't need an instance to access a static field
If you want to use them from outside the class, you should prefix the static field name with the class name

```
Good

public class Person
{
    public static int MinimumAgeForVoting = 18;

    //....
}

Console.WriteLine(Person.MinimumAgeForVoting);
```

cegeka

CONSTANT FIELDS

☐ A constant is a variable that has a known value at compile time and doesn't change its values during runtime.

☐ The constant fields belong to the class, not to the instances
   constant fields are shared between all instances of a class
   there is no need to use the "static" keyword when defining them

☐ Constants are used similar to static fields, by prefixing with the class name

```
Good

public class Person
{
    public const int MinimumAgeForId = 14;

    //....
}

Console.WriteLine(Person.MinimumAgeForId);
```

cegeka

READONLY FIELDS

- A readonly field is a variable that doesn't have a known value at compile time. The value is set at runtime and doesn't change its values during the lifetime of the application.
- You should assign it when you defined it or in the constructor
- The readonly fields belong to the instances, not to the class
- You need a instance to access it and can have different values for each instance

```
Good

public class Person
{
    public readonly int MinimumAgeForRetirement;
    // ...
    private Gender gender;

    public Person(string firstName, string lastName, Gender gender)
    {
        this.MinimumAgeForRetirement = (gender == Gender.Male) ? 65 : 63;
    }
}
```

cegeka

PROPERTY NAMES

- A property is a member that allows: accessing/ modifying a field or computing a value
- A property must have at least a get or a set, each with different access modifiers

- Properties are called similar to fields (without parenthesis, like methods)

```
Good

public class Person
{
    public int Id
    {
        get
        {
            return this.id;
        }
    }

    public string FirstName { get; private set; }
    public string LastName { get; private set; }

    public string FullName
    {
        get
        {
            return $"{FirstName} {LastName}";
        }
    }
}
```

cegeka

BOOLEAN METHOD/PROPERTY NAMES

- Shoud sound like it is asking a true/false question
- Use "Is/Are/Can/Has" for a property returns a Boolean value

```
Evil
bool valid
bool start
bool open
public bool Voted
public bool GetIdCardValid()
public bool Adult
```

```
Good
bool isValid
bool hasStarted
bool isOpen
public bool CanVote
public bool HasIdCard()
public bool IsLegallyAdult
```

COMPARE BOOLEAN

⬛ Assign and compare implicitly

Evil

```
if (age > retirementAge)
{
    isRetired = true;
}
else
{
    isRetired = false;
}
```

Good

```
bool isRetired = (age > retirementAge);
```

- Fewer lines
- More human readable
- Less duplication
- No need to initialize before condition

Evil

```
if (isRetired == true)
```

Good

```
if (isRetired)
```

cegeka

USE POSITIVE CONDITIONALS

☐ It is difficult to understand double negation

Evil
```
if (!isNotValid)
```

Good
```
if (isValid)
```

cegeka

TERNARY OPERATOR

☐ Avoid using multiple ternaries in the same condition, it will be difficult to understand

Evil

```
int retirementAge = (gender == Gender.Female) ? ((age >= 55) ? 55 : 61) : 65;
```

Good

```
int retirementAge = (gender == Gender.Female) ? 61 : 65;
```

USE ENUMS INSTEAD OF STRINGS/MAGIS NUMBER - DEMO

Evil

```
public void LogUser(User user)
{
    if (user.Role == "manager")
    {
        GiveAdminRights();
    }
    else
    {
        if (user.Role == "employee")
        {
            GiveEditRights();
        }
        else
        {
            GiveReadOnlyRights();
        }
    }
}
```

Good

```
public enum UserRoles
{
    Manager,
    Employee
}

public void LogUser(User user)
{
    if (user.Role == UserRoles.Manager)
    {
        GiveAdminRights();
    }
    else
    {
        if (user.Role == UserRoles.Employee)
        {
            GiveEditRights();
        }
        else
        {
            GiveReadOnlyRights();
        }
    }
}
```

Good

```
public enum Roles
{
    Manager,
    Employee
}

public void LogUser(User user)
{
    switch (user.Role)
    {
        case UserRoles.Manager:
            GiveAdminRights();
            break;
        case UserRoles.Employee:
            GiveEditRights();
            break;
        default:
            GiveReadOnlyRights();
            break;
    }
}
```

cegeka

COMPLEX CONDITIONS & ENUMS - DEMO

- Use enums instead of magic numbers
- Simplify the understanding of complex conditions by

Intermediate variables
Encapsulate in function

Evil

```
public void PlayVideo(string fileName)
{
    string fileExtension = Path.GetExtension(fileName);

    if ((fileExtension == ".mp4" || fileExtension == ".avi")
        && (CurrentState != 2 || CurrentState == 3))
    {
        LoadVideo(fileName);
        Play();
    }
}
```

Good

```
private bool IsExtensionValid(string fileName)
{
    string fileExtension = Path.GetExtension(fileName);
    return (fileExtension == ".mp4" || fileExtension == ".avi");
}

private bool CanPlayNewVideo()
{
    return (CurrentState != PlayerStates.Playing || CurrentState == PlayerStates.Paused);
}

public void PlayVideo(string fileName)
{
    if (IsExtensionValid(fileName) && CanPlayNewVideo())
    {
        LoadVideo(fileName);
        Play();
    }
}
```

cegeka

POLYMORFISM OVER ENUMS - DEMO

Evil

```
public enum AttendanceTypes
{
    Daily,
    Weekend,
    Evening
}

public class Student
{
    public AttendanceTypes AttendanceType { get; private set; }

    public int GetMinimumCreditPointsToPromote()
    {
        int minCreditPoints = 0;
        switch (this.AttendanceType)
        {
            case AttendanceTypes.Daily:
                // compute and return credits for daily courses
                minCreditPoints = 100;
                break;

            case AttendanceTypes.Evening:
                // compute and return credits for evening courses
                minCreditPoints = 50;
                break;

            case AttendanceTypes.Weekend:
                // compute and return credits for weekend courses
                minCreditPoints = 35;
                break;
        }
        return minCreditPoints;
    }
}
```

Good

```
public class DailyStudent : Student
{
    public override int GetMinimumCreditPointsToPromote()
    {
        // compute and return credits for daily courses
        return 100;
    }
}

public class EveningStudent : Student
{
    public override int GetMinimumCreditPointsToPromote()
    {
        // compute and return credits for evening courses
        return 50;
    }
}

public class WeekendStudent : Student
{
    public override int GetMinimumCreditPointsToPromote()
    {
        // compute and return credits for weekend courses
        return 35;
    }
}

public abstract class Student
{
    public abstract int GetMinimumCreditPointsToPromote();
}
```

cegeka

THE POWER OF ENUMS

- Strongly type check, typos can easily generate bugs

- You can use intellisense

- You have a finite known list of all possible values

- If you use strings and change in one place, but not all places, you will get inconsistencies (bug)

- Switch should always have a default clause

cegeka

CONSTANTS INSTEAD OF MAGIC NUMBERS

- Magic numbers are hardcoded values whose value are not obvious for someone looking the first time at the code
- They can generate bugs because:
  Someone that doesn't know what the value represents, can change it incorrectly
  If it is used in multiple places and the dev forgets to replace it in all the places, there will be bugs

Evil
```
return (DateTime.Now - DateOfBirth).TotalDays / 365.2425;
```

Good
```
double daysInAYear = 365.2425
return (DateTime.Now - DateOfBirth).TotalDays / daysInAYear;
```

cegeka

CONSTANTS INSTEAD OF MAGIC NUMBERS

⬚ If it is used in multiple places and the developer forgets to replace it in all the places, there will be bugs


```
Evil

public class Person
{
    public bool CanVote
    {
        get
        {
            return Age > 18;
        }
    }

    public bool IsLegallyAdult
    {
        get
        {
            return Age > 21; //18;
        }
    }
}
```

```
Good

public class Person
{
    public static int MinimumAgeForVoting = 18;

    public bool CanVote
    {
        get
        {
            return Age > MinimumAgeForVoting;
        }
    }

    public bool IsLegallyAdult
    {
        get
        {
            return Age > MinimumAgeForVoting;
        }
    }
}
```

cegeka

REMOVE HARD – CODED VALUES

- Instead of hardcoding values in the code, you might use values from the database or a configuration file
- This way, if something changes, you don't have to rebuild and install the application, just update the database – dynamic logic

Evil
```
if (yearsOfExperience > 20)
{
    return 28;
}
else if (yearsOfExperience > 10)
{
    return 23;
}
else
{
    return 21;
}
```

Id	MinYearsOfExperience	DaysOffPerYear
1	0	21
2	10	23
3	20	28

Good
GetDaysOffPerYear()

Thank you

cegeka