Качествен код и преработване

Коректност на кода, четимост, възможност за поддръжка и тестване, преработка

SoftUni Foundation

Проект "Отворено учебно съдържание по програмиране и ИТ", СофтУни Фондация
https://github.com/BG-IT-Edu

Курс "Структури от данни и алгоритми"

Софтуерни и хардуерни науки

Съдържание

SoftUni Foundation

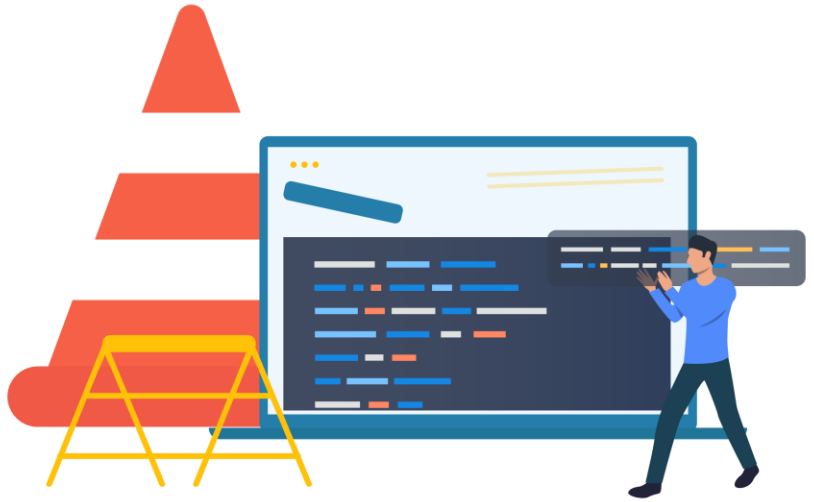
1. **Качествен код**
   - Конвенции за **именуване**
   - **Форматиране** на **код**
   - **Коментари** и **документация** на код
   - Правилно **организиране на данните**
   - Използване на **променливи**, **изрази**, **блокови оператори** и **константи**
   - Качествени **методи** и **класове**
2. **Преработване**
   - Принципи, процеси, шаблони

Какво е качествен код?

Коректност, четимост, възможност за поддръжка и тестване

Характеристики на качествения код (1)

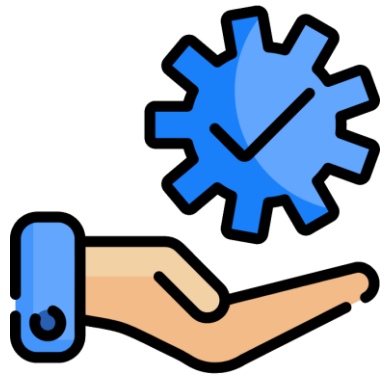
- Лесен за **четене** и **разбиране**
- Лесен за **модифициране** и **поддържане**
- Добре **тестван**
- Добре разработена **архитектура** и **дизайн**
- Добре **документиран**
  - Самодокументиращ се код
- Добре **форматиран**



Характеристики на качествения код (2)

- **Силна свързаност** във всички нива: **модули**, **класове**, **методи**
  - **Един елемент** е отговорен за **една задача**
- **Слабо функционално обвързване** между **модули**, **класове**, **методи** и т.н.
  - Всички елементи са **независими** един от друг
- **Описателни имена** на класовете, методите, променливите и т.н.

Характеристики на качествения код (3)



- Чисто написани класове, интерфейси и класова йерархия
  - Добра **абстракция**, **капсулиране**, **наследяване** и **полиморфизъм**
  - Простота, многократна употреба, минимална сложност
- Променливи, данни, изрази и константи
  - Минимална **продължителност** и **живот** на променливата, прости **изрази**, добре използвани **константи**

Характеристики на качествения код (4)

- **Добро поведение**
  - В съответствие с **изискванията**
  - **Стабилен**, няма увисвания и грешки
  - **Без грешки** – работи както трябва
  - **Правилен отговор** при неправилна употреба
- Лесен за **поддръжка** и **промяна**

Защо качеството е толкова важно? (1)

- Какво прави този код? **Работи** ли правилно? Имали **грешки**?
  Трудно е да се **разбере**:

Не е лесен за четене

Лошо е форматиран

Трудно се разбира

Не може да се тества

Не е документиран

```
static void Main()
{
  int value=010, i=5, w;
  switch(value){case 10:w=5;Console.WriteLine(w);break;case 9:i=0;break;
          case 8:Console.WriteLine("8 ");break;
     default:Console.WriteLine("def ");{
               Console.WriteLine("hoho "); }
     for (int k = 0; k < i; k++, Console.WriteLine(k - 'f'));break;} {
Console.WriteLine("loop!"); }
}
```

Защо качеството е толкова важно? (2)

- Сега **кодът е форматиран**, но все още не е **изчистен**

- Ако кодът е написан по този начин, ще бъде **невъзможно** да се създаде голям и сериозен **софтуерен проект**

- Така че **кодът с добро качество** може да се счита за **основно свойство на софтуера**

```
static void Main()
{
  int value = 010, i = 5, w;
  switch (value)
  {
    case 10: w = 5; Console.WriteLine(w); break;
    case 9: i = 0; break;
    case 8: Console.WriteLine("8 "); break;
    default:
      Console.WriteLine("def ");
      Console.WriteLine("hoho ");
      for (int k = 0; k < i; k++,
       Console.WriteLine(k - 'f')) ;
      break;
  }
  Console.WriteLine("loop!");
}
```

Защо качеството е толкова важно? (3)

Така трябва да изглежда **предишния код**, когато е **форматиран** правилно

```
static void Main()
{
    int value = 10;
    int row = 5;
    int column = 0;

    switch (value)
    {
        case 10:
            column = 5;
            Console.WriteLine(column);
            break;
        case 9:
            row = 0;
            break;
        case 8:
            Console.WriteLine("8 ");
            break; …
    }
    Console.WriteLine("loop!");
}
```

Наименуване на идентификатори

Описателни имена на променливи, методи, класове на идентификатори

Основни правила при наименуване

- **Винаги** използвайте **английски** – всички програмисти говорят **английски**
- Избягвайте **съкращения** и **трудни** за **произнасяне** имена
  - Пример: `scriptsCount`, не `scrpCnt`
  - Пример: `dateTimeBulgarianRegExPattern`, не `dtbgRegExPtrn`
- Винаги използвайте **смислени имена**
  - Дали едно име е **смислено** или **не**, зависи от неговия **контекст**
- Използвайте **последователно наименуване** в целия
- Името трябва да бъде дълго **толкова колкото трябва**

Наименуване на класове и структури

- За **класове** и **структури** използвайте следните формати:
  - **[Съществително]**
  - **[Прилагателно] + [Съществително]**
- Примери: `Student`, `FileSystem`, `BinaryTreeNode`
- Грешни примери: `Move`, `FindUsers`, `Fast`, `Optimize`
- За **интерфейси** използвайте следния формат:
  - 'I' + **[Съществително]**
  - 'I' + **[Прилагателно]** + **[Съществително]**
  - 'I' + **[Глагол]** + 'able'
- Примери: `IEnumerable`, `IFormattable`, `IDataReader`
- Грешни примери: `List`, `iFindUsers`, `IMemoryOptimize`

Наименуване на специални класове

Използвайте тази класова наставка

- Атрибути
  - `WebServiceAttribute`, не `WebService`
- Колекция от класове
  - `StringsCollection`, не `ListOfStrings`
- Грешки
  - `FileNotFoundException`, не `FileNotFoundError`
- Делегатни класове
  - `DownloadFinishedDelegate`, не `WakeUpNotification`
  - `ClickedEventHandler`, не `ClickedButton`

Наименуване на методи

- Използвайте **`PascalCase`** със следния формат:
  - **[Глагол]**, **[Глагол] + [Съществително]** или **[Глагол] + [Прилагателно] + [Съществително]**
  - Примери: **`Show`**, **`LoadSettingsFile`**
  - Грешни примери: `Student`, `Generator`, `counter`
- Методите, връщащи стойности, трябва да описват **върнатата стойност**
  - Примери: **`ConvertMetersToInches`**, не `MetersInches` или `Convert` или `ConvertUnit`

Наименуване на методи с параметри и променливи

- Имена на **параметрите** на **методи**
  - Предпочитана форма: **[Съществително]** или **[Прилагателно] + [Съществително]**
  - Примери: `firstName`, `report`
  - Грешни примери: `p`, `p1`, `p2`, `populate`
- **Имена** на **променливи**
  - Предпочитана форма : **[Съществително]** или **[Прилагателно] + [Съществително]**
  - Трябва да обяснява **дейността на променливата**
  - Примери: `firstName`, `report`, `config`, `usersList`
  - Грешни примери: `foo`, `bar`, `p`, `p1`, `p2`, `populate`

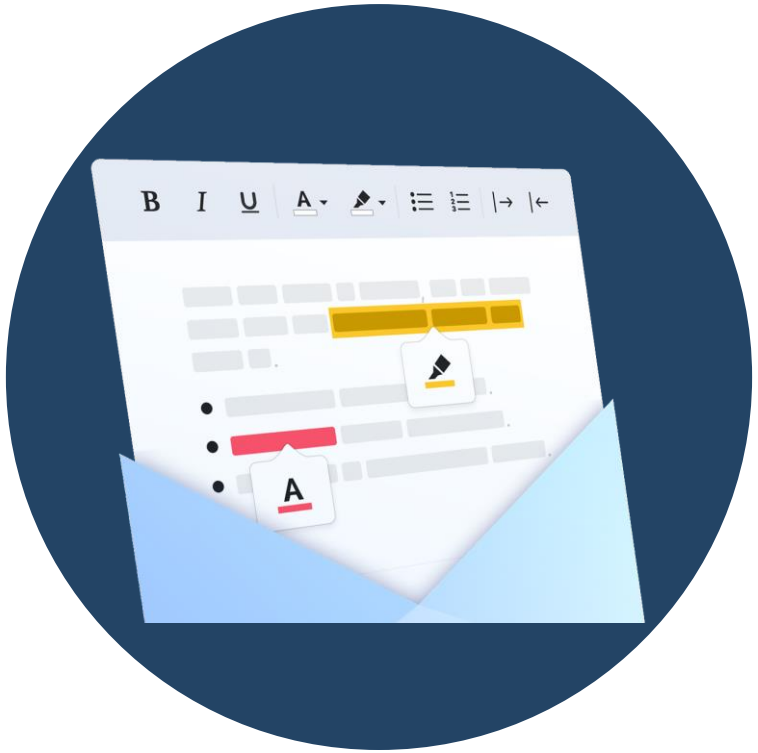
Наименуване на Namespaces и папки

SoftUni Foundation

- Наименуване на **namespace**
  - Използвайте **PascalCase** със следния формат:
    - **Company.Product.Component.…**
  - Примери: **Microsoft.WinControls.GridView**
  - Грешни примери: **Microsoft_WinControlsGridView**
- Наименуване на **папките** на **проекта**
  - **Имената на папките** трябва да **следват namespace-совете на проекта**
  - Примери: **System.Collections.Generic…**
  - Грешни примери: **generic.src**, **system_collections_generic**

Наименуване на файлове и приложения

- Наименуване на **файлове**
  - Файловете трябва да имат имена, съответстващи на тяхното съдържание
  - Примери: `Constants.cs`, `CryptographyAlgorithms.cs`
  - Грешни примери: `Program.cs`, `SourceCode.cs` , `Page1.aspx`
- Наименуване на **приложения**
  - Използвайте `PascalCase` със следният формат:
    - **[Съществително]** или **[Прилагателно] + [Съществително]**
  - Примери: `BlogEngine`, `NewsAggregatorService`
  - Грешни примери: `ConsoleApplication4`, `WebSite2`, `zadacha_14`, `online_shop_temp2`

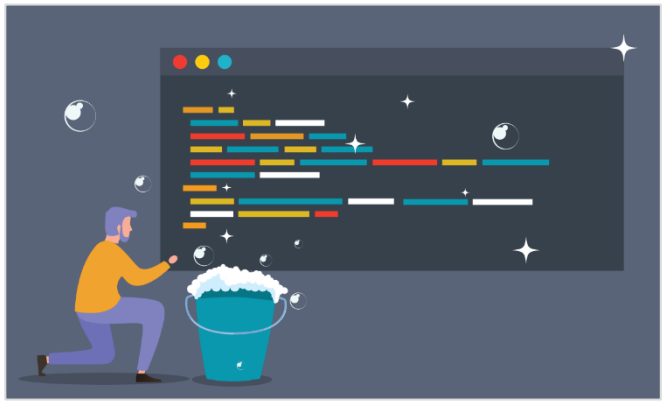


Форматиране на код

Правилно форматиране на сорс кода

Форматиране на кода

- **Форматирането** има две цели:
  - Да подобри **четимостта** на кода
  - Да подобри **поддържимостта** на кода



- Форматирането трябва да **следва логическата структура на програмата**
  - Всеки стил на форматиране, който **следва** горните **принципи**, **е добър**
  - Всеки друг стил на форматиране **не е добър**

Форматиране на условни конструкции и цикли

SoftUni Foundation

- Форматирането на **условни оператори** и **цикли**
  - Винаги използвайте **{ }** след **if** / **for** / **while**, дори когато следва само един ред код
  - Отстъп на тялото на блока след **if** / **for** / **while**
  - Винаги слагайте нов ред след блока **if** / **for** / **while**
  - Винаги слагайте **{** на следващият ред
  - Никога не отстъпвайте **повече от един [Tab]**

Символът { трябва да бъде на следващия ред

```
for (int i=0; i<10; i++) {
   Console.WriteLine("i={0}", i);
}
```

Символите { и } ги няма

```
for (int i=0; i<10; i++)
Console.WriteLine("i={0}",i);
```

```
for (int i = 0; i < 10; i++)
{
   Console.WriteLine("i = {0}", i);
}
```

Форматиране на методи (1)

- **Методите** трябва да бъдат с **един отстъп [Tab]** навътре от **тялото на класа**
  - **Телата на методите** също трябва да бъдат с **един отстъп [Tab]** навътре
- **Скобите** в декларациите на **методите** трябва да бъдат **форматирани** по следния начин:

```
public class IndentationExample
{
    private int Zero()
    {
        return 0;
    }
}
```

Целия метод трябва да има отстъп **[Tab]**

Тялото на метода също **трябва да има отстъп**

```
private static ulong CalculateFactorial(uint num)
```

- **Не слагайте интервал** между **скобите**:

```
private static ulong CalculateFactorial ( uint num )
```

```
private static ulong CalculateFactorial (uint num)
```

Форматиране на методи (2)

- **Разделете параметрите** на метода със **запетая** и **интервал**

```
private void RegisterUser(string username ,string password)
```

```
private void RegisterUser(string username, string password)
```

- Използвайте празен ред, за да разделяте логически свързани **последователности** от **редове** :

```
private List<Report> PrepareReports()
{
    List<Report> reports = new List<Report>();

    // Create incomes reports
    Report incomesSalesReport = PrepareIncomesSalesReport();
    reports.Add(incomesSalesReport);
    Report incomesSupportReport = PrepareIncomesSupportReport();
    reports.Add(incomesSupportReport);

    return reports;
}
```

Празен ред

Празен ред

Коментари и документация на кода

Самодокументиращ се код

Ефективни коментари

- **Ефективните коментари** не **повтарят кода**
  - Те обясняват на **по-високо ниво** и разкриват **неочевидни** детайли
- Най-добрата **софтуерна документация** е **сорс кода** - поддържайте го **чист** и **четим**!
- **Самодокументиращият код** сам се обяснява и **не се нуждае** от **коментари**
  - Прост дизайн, малки методи, добре **наименувани променливи** и т.н.

Лошо документиран код – Пример

```
public static List<int> FndPrimes(int start, int end)
{
    List<int> primesList = new List<int>(); // Създаване на списък от числа

    // Започваме цикъл от start до end
    for (int num = start; num <= end; num++)
    {
        bool prime = true; // Декларираме булева променлива, като задаваме
стойност true
        // Започваме цикъл от 2 до корен квадрате от num
        for (int div = 2; div <= Math.Sqrt(num); div++)
        {
            // Проверяваме дали div се дели без остатък
            if (num % div == 0)
            {
                // Намираме разделителя -> числото не е просто
                prime = false;
                break; // Излизаме от цикъла
            }
        }
    }
}
```

Всички коментари обясняват очевидни детайли. Повтарят кода.

Самодокументиращ се код – Примери (1)

SoftUni Foundation

```
public static List<int> FindPrimes(int start, int end)
{
    List<int> primesList = new List<int>();
    for (int num = start; num <= end; num++)
    {
        bool isPrime = IsPrime(num);
        if (isPrime)
        {
            primesList.Add(num);
        }
    }

    return primesList;
}
```

Добрият код не се нуждае от обяснения. Той сам се обяснява.

Самодокументиращ се код – Примери (2)

```
private static bool IsPrime(int num)
{
    bool isPrime = true;
    int maxDivider = (int) Math.Sqrt(num);
    for (int div = 2; div <= maxDivider; div++)
    {
        if (num % div == 0)
        {
            // Намираме делител -> числото не е просто
            isPrime = false;
            break;
        }
     }

  return isPrime;
}
```

Добрите методи имат добри, лесни за четене и разбиране имена

Коментарът обяснява неочевидни детайли. Не повтаря кода

Правилно организиране на данните

Променливи, изрази, контролни оператори

Обхват и видимост на променливите (1)

SoftUni Foundation

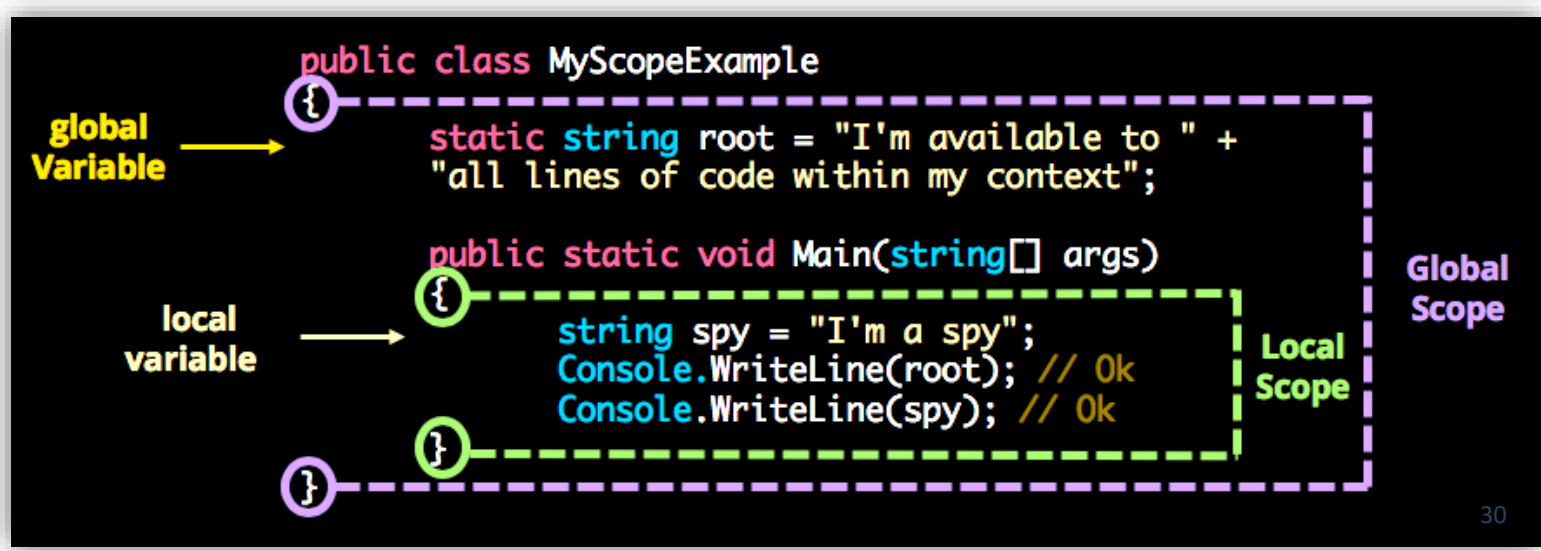
- **Обхват** – зона за **достъп** на **променливите**
  - **Глобална** (статична), **член-променлива**, **локална**
  - Обхватът често е **комбиниран** с **видимостта**
- **Видимостта** на **променливите е ограничение** на достъп до **променливите**
  - Винаги се опитвайте да **ограничите обхвата** и **видимостта** на **променливите**

internal

public

private

protected



Обхват и видимост на променливите (2)

SoftUni Foundation

Така променливата е **достъпна за всички**

```
public class Globals
{
    public static int state = 0;
}
public class ConsolePrinter
{
    public static void Print()
    {
        if (Globals.state == 0)
            Console.WriteLine("Hello.");
        else
            Console.WriteLine("…")
    }
}
```

По-добре сложете **променливата** като **аргумент**

```
public class ConsolePrinter
{
    public static void Print(int state)
    {
        if (state == 0)
            Console.WriteLine("Hello.");
        else
            Console.WriteLine("…")
    }
}
```

Продължителност и живот на променливата (1)

- **Продължителност на променливата** == среден **брой** на редовете на кода (LOC) между **употребата** на **променливите**

```
1   a = 1;
2   b = 1;
3   c = 1;
4   b = a + 1;
5   b = b / c;
```

продължителност = 1

продължителност = 0

Един ред между първото и второто посочване на b

Без редове между второто и третото посочване на b

Средната продължителност за b е (1 + 0) / 2 = 0.5

- **Живот на променливата** == броят на редове на код (LOC) между **първата** и **последната** употреба на **променливата** в **блока**

```
25 recordIndex = 0;
26 while (recordIndex < recordCount)
27 { ...
28 recordIndex = recordIndex + 1;
...
```

recordIndex (ред 28 - ред 25 + 1) = 4

Средният живот на recordIndex е 4

Продължителност и живот на променливата (2)

- **Продължителността** и **животът** на променливата трябва да се **намалят** до **минимум**
- Правило на минимализирането на **продължителността** и **живота**:
  - Инициализирайте променливите **преди първото** им **използване**

Инициализираме на променливата count преди нейното използване

```
int count = 0;

while (count != 10)
{
    count++;
}

Console.WriteLine(count);
```

Продължителност и живот на променливата (3)

SoftUni Foundation

```
int count;
int[] numbers = new int[100];
for (int i = 0; i < numbers.Length; i++)
{
  numbers[i] = i;
}
count = 0;
for (int i = 0; i < numbers.Length / 2; i++)
{
  numbers[i] = numbers[i] * numbers[i];
}
for (int i = 0; i < numbers.Length; i++)
{
  if (numbers[i] % 3 == 0)
  {
    count++;
  }
}
Console.WriteLine(count);
```

Живот = 19

Продължителност = (5+8+2) / 3 = 5

```
int[] numbers = new int[100];
for (int i = 0; i < numbers.Length; i++)
{
    numbers[i] = i;
}
for (int i = 0; i < numbers.Length / 2; i++)
{
    numbers[i] = numbers[i] * numbers[i];
}
int count = 0;
for (int i = 0; i < numbers.Length; i++)
{
    if (numbers[i] % 3 == 0)
    {
        count++;
    }
}
Console.WriteLine(count);
```

Живот = 9

Продължителност = (4+2) / 3 = 2

Не използвайте сложни изрази във вашия код!

SoftUni Foundation

```
for (int i = 0; i < xCoords.Length; i++)
  for (int j = 0; j < yCoords.Length; j++)
    matrix[i][j] =
      matrix[xCoords[FindMax(i) + 1]][yCoords[FindMin(j) - 1]] *
      matrix[yCoords[FindMax(j) + 1]][xCoords[FindMin(i) - 1]];
```

Какво трябва да направим при IndexOutOfRangeException?

Има 10 потенциални източника за IndexOutOfRangeException

Вместо това, опростете сложния израз

По-четим е от преди

По-лесен е за поддържане и дебъгване

```
int minXStartIndex = FindMin(i) - 1;
int maxXStartIndex = FindMax(i) + 1;
int minYStartIndex = FindMin(j) - 1;
int maxYStartIndex = FindMax(j) + 1;

int minXcoord = xCoords[minXStartIndex];
int maxXcoord = xCoords[maxXStartIndex];
int minYcoord = yCoords[minYStartIndex];
int maxYcoord = yCoords[maxYStartIndex];
int newValue = matrix[maxXcoord][minYcoord] * matrix[maxYcoord][minXcoord];
matrix[i][j] = newValue;
```

Използвайте константи

- **Константите** трябва да се използват в следните случаи:
  - При използването на **магически числа** / **стойности**

```
double area = 3.14159206 * radius * radius;
```

```
public const double PI = 3.14159206;
double area = PI * radius * radius;
```

  - Когато се нуждаем от **числа** или **стойности**, чиито **логическа значимост** и **стойност** не са очевидни
    - **Имена на файлове**

```
public static readonly string SettingsFileName =
    "ApplicationSettings.xml";
```

    - **Математически константи**

```
public const double E = 2.7182818284;
```

    - **Граници и ограничения**

```
public const int ReadBufferSize = 5 *
1024 * 1024;
```

Писане на код без вложени цикли и условия

SoftUni Foundation

```
ReportHeader CreateReportHeader(Report
report)
{
    // …
}

Report CreateReport()
{
    var report = new Report();
    report.Footer =
       CreateReportFooter(report);
    report.Content =
       CreateReportContent(report);

    report.Header =
       CreateReportHeader(report);
    return report;
}

ReportContent
CreateReportContent(Report report)
{
    // …
}
```

Разкрийте зависимостите

Наименувайте методите според тяхната зависимост

Използвайте параметри на метода

```
Report CreateReport()
{
  var report = new Report();

  report.Header =
       CreateReportHeader(report);
  report.Content =
       CreateReportContent(report);
  report.Footer =
       CreateReportFooter(report);

  return report;
}

ReportHeader CreateReportHeader(Report
report)
{
  // …
}

ReportContent CreateReportContent(Report
report)
{
  // …
}
```

Направете кода да се чете от отгоре надолу

Групирайте свързани твърдения заедно

Направете ясни граници за зависимостта

Използвайте отделни методи

Избягване на дълбоко вложени цикли

```
if (maxElem != Int32.MaxValue)
{
    if (arr[i] < arr[i + 1])
    {
        if (arr[i + 1] < arr[i + 2])
        {
            if (arr[i + 2] < arr[i + 3])
            {
                maxElem = arr[i + 3];
            }
            else
            {
                maxElem = arr[i + 2];
            }
        }
        else
        {
            if (arr[i + 1] < arr[i + 3])
            {
                maxElem = arr[i + 3];
            }
            else
            {
                maxElem = arr[i + 1];
            }
        }
    } …
```

Дълбокият вложен код е сложен и труден за четене и разбиране

Повече от 2-3 нива е твърде дълбоко

Използването на добре наименувани методи прави кода самодокументиращ се

```
private static int Max(int i, int j)
{
    if (i < j) return j;
    else {…}
}
private static int Max(int i, int j, int k)
{
    if (i < j)
    {
        int maxElem = Max(j, k);
        return maxElem;
    }
    else
    {
        int maxElem = Max(i, k);
        return maxElem;
    }
}
```

Може да извлечете части от кода в отделни методи

Това опростява логиката на кода

Поддържайте изразите и условията прости

- Не използвайте сложни **if-условия** → опростете ги с **булеви променливи** или **булеви методи**

```
if (x > 0 && y > 0 && x < Width-1 && y < Height-1 &&
    matrix[x, y] == 0 && matrix[x-1, y] == 0 &&
    matrix[x+1, y] == 0 && matrix[x, y-1] == 0 &&
    matrix[x, y+1] == 0 && !visited[x, y]) …
```

Сложните булеви изрази могат да бъдат вредни

Как ще намерите проблема при IndexOutOfRangeException?

```
bool inRange = x > 0 && y > 0 && x < Width-1 && y < Height-1;
if (inRange)
{
    bool emptyCellAndNeighbours =
        matrix[x, y] == 0 && matrix[x-1, y] == 0 &&
        matrix[x+1, y] == 0 && matrix[x, y-1] == 0 &&
        matrix[x, y+1] == 0;
    if (emptyCellAndNeighbours && !visited[x, y]) …
}
```

Лесно за четене – логиката на условията е вярна

Лесно за дебъгване – можете да слагате breakpoint на if

Подредба на Switch-случаи

```
void ProcessNextChar(char ch)
{
    switch (parseState)
    {
        case InTag:
            if (ch == ">")
            {
                Console.WriteLine($"Found tag:
{tag}");
                text = "";
                parseState =
ParseState.OutOfTag;
            }
            else
            {
                tag = tag + ch;
            }
            break;
        case OutOfTag: …
    }
}
```

Подредете случаите по азбучен числен ред

```
void ProcessNextChar(char ch)
{
  switch (parseState)
  {
    case InTag:
        ProcessCharacterInTag(ch);
        break;
    case OutOfTag:
        ProcessCharacterOutOfTag(ch);
        break;
    default:
       throw new InvalidOperationException
    ("Invalid parse state: " + parseState);
  }
}
```

Сложете нормалния случай на на първо място

Поставате най-необичайния случай последен

Използвайте default за прихващане на грешки

Качествени методи

Силна свързаност на отговорностите и слабо функционално обвързване

Защо се нуждаем от методи?

SoftUni Foundation

- Методите са **важни**, защото:
  - Намаляват **сложността**
    - Сложните задачи са разделени на **по-малки проблеми** – "разделяй и владей"
  - Подобрява **четимостта** на кода
    - **Малки** методи с **добри** имена прави кода **самодокументиращ** се
  - Избягвайте **повтарянето на код**
    - Повтарянето на кода е **трудно** за **поддръжка**

```
/**
* Code Readability
*/
if (readable()) {
    be_happy();
} else {
    refactor();
}
```

Използване на методи

- Методите трябва да **правят** това, което **гласи името им**

```
int Sum(int[] elements)
{
    int sum = 0;
    foreach (int element in elements)
    {
        sum = sum + element;
    }
    return sum;
}
```

Какво ще стане ако съберем 2,000,000,000 + 2,000,000,000

Резултат: -294967296

```
long Sum(int[] elements)
{
    long sum = 0;
    foreach (int element in elements)
    {
        sum = sum + element;
    }
    return sum;
}
```

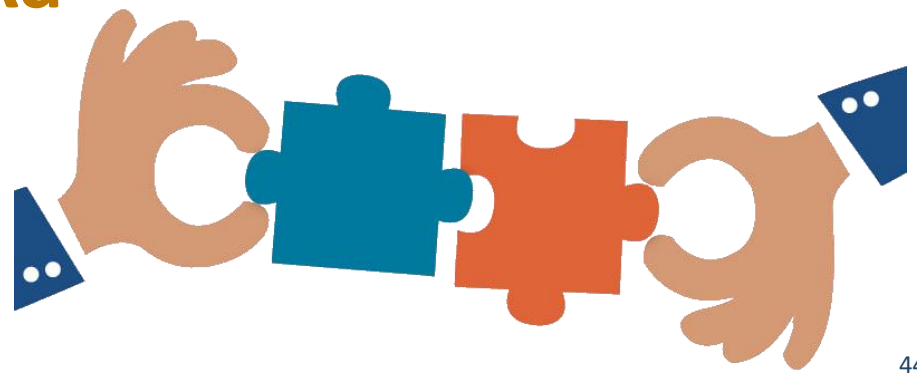
Няма странични ефекти

Работи правилно във всички случай

- В случай на **грешен вход** или **грешки от програмиста** трябва да се **върне грешка** (примерно throw exception)

Силна свързаност на отговорностите (Cohesion)

- **Свързаност на отговорностите** == степента, в която елементите в модула си **принадлежат**
  - Модул с **висока свързаност** на отговорностите съдържа елементи, които са **тясно свързани** помежду си
  - Модул с **ниска свързаност** на отговорностите съдържа елементи, които са **разделени** помежду си
- Методите трябва да имат **силна връзка**



Силен Cohesion в методите – Примери

- **Методите** трябва да решават **само една задача**
- Трудно е да се наименува метод, който извършва **няколко операции** на един път

Без **външни зависимости** или **странични ефекти**

```
Math.Sqrt(value) → square root

char.IsLetterOrDigit(ch)

string.Substring(str, startIndex,
length)
```

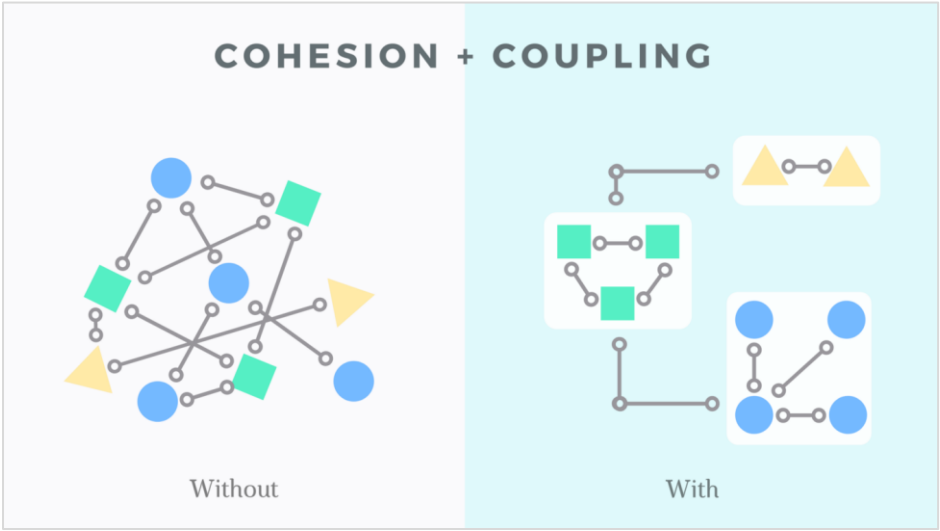
Методът изпълнява **операция** и **връща единичен резултат**

Извършва **различни операции** спрямо **параметъра**

```
object ReadAll(int operationCode)
{
    if (operationCode == 1) …
       // Прочети името на човек
    else if (operationCode == 2) …
       // Прочети адреси
    else if (operationCode == 3) …
       // Прочети дата
    …
}
```

Слабо функционално обвързване (Coupling)

- **Функционално обвързване** == степента на **взаимозависимост** между **модулите**
- **Слабото** функционално обвързване се отнася до **връзката между компонентите**
  - Осигурява **гъвкавост**
  - Повече **адаптивност**
  - Силно функционално обвързване → **спагети код**



- **Силната свързаност на отговорностите** е в корелация със **слабо функционално обвързване**

Слабо функционално обвързване – Примери


```
class Sumator
{
    public int a, b;
    int Sum()
    {
        return a + b;
    }
    static void Main()
    {
        Sumator sumator =
        new Sumator() { a = 3, b = 5
};

Console.WriteLine(sumator.Sum());
    }
}
```

Подаване на параметри чрез класови полета

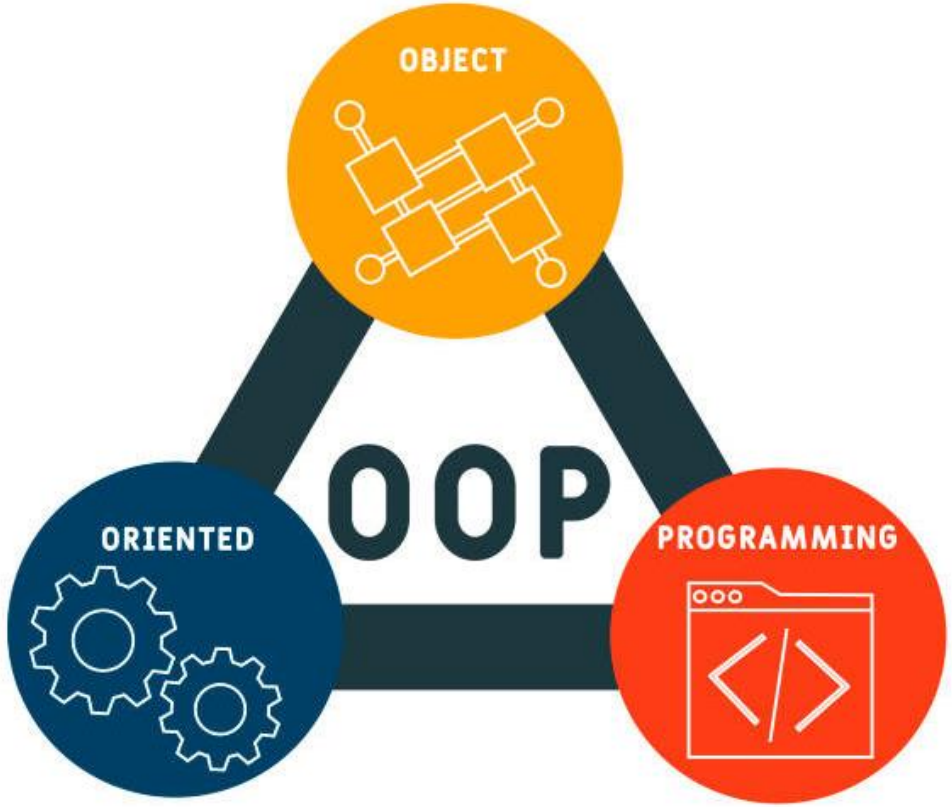
Не правете това, освен ако нямате добра причина!

```
byte[] EncryptAES
        (byte[] inputData, byte[] secretKey)
{
    MemoryStream inputStream =
       new MemoryStream(inputData);

    MemoryStream outputStream =
       new MemoryStream();

    EncryptionUtils.EncryptAES(
       inputStream, outputStream, secretKey);

    byte[] encryptedData =
       outputStream.ToArray();

    return encryptedData;
}
```

Скрийте сложната логика

Осигурете прост и ясен интерфейс

Създайте помощни класове

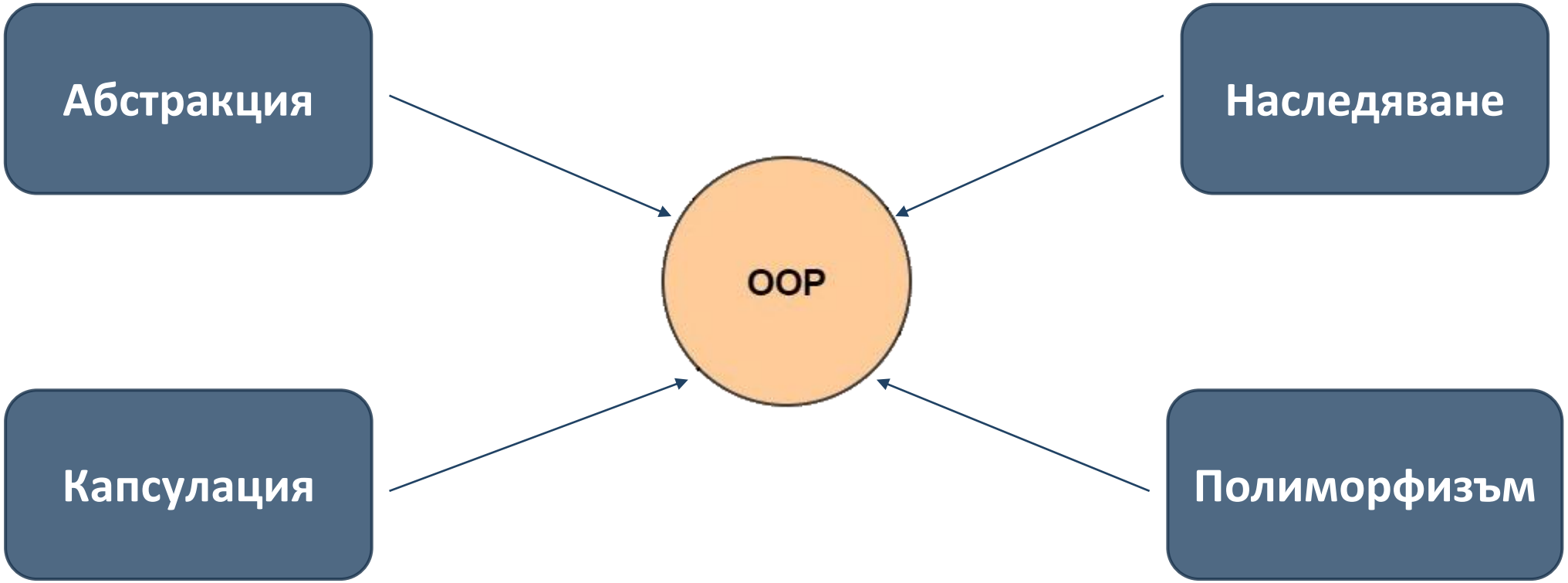


Качествени класове

Утвърдени практики в ООП

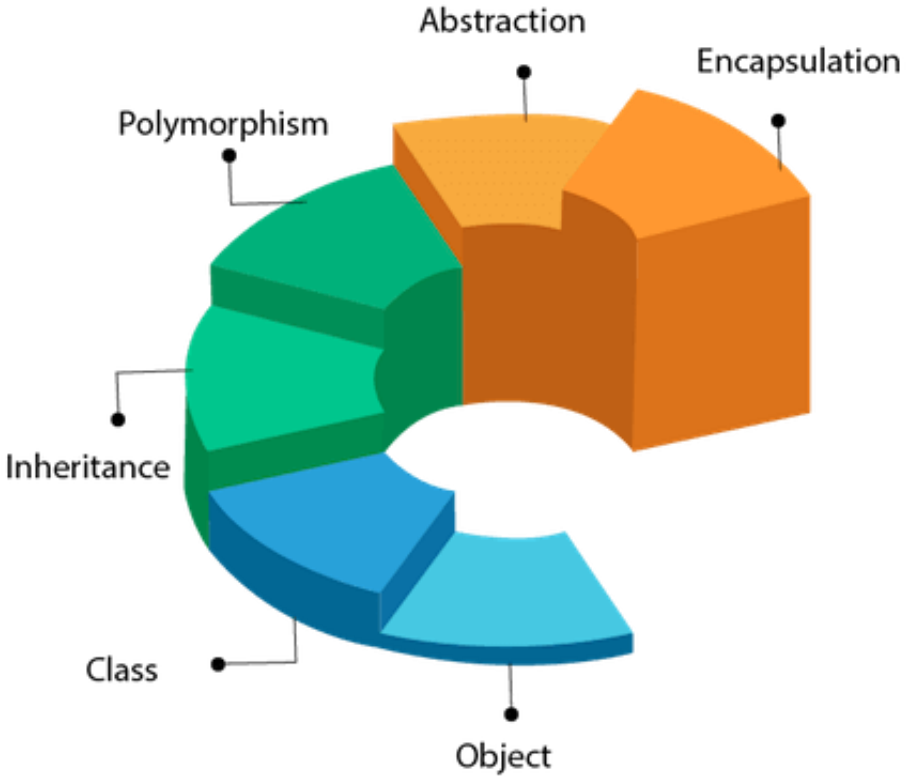
Чисто написани класове

- Когато **създавате качествен клас**, трябва да следвате основните правила, които произтичат от четирите основни **ООР принципа**



ООР принципи

- **Абстракция** == представяне на съществените **характеристики**, без да се представят **детайлите във фона**
- **Капсулация** == обединяване на кода и данните в **едно цяло**
- **Наследяване** == един **клас** съдържа **свойство** на **друг клас**
- **Полиморфизъм** == Способността на един **обект** да приема **различни форми**



Абстракция и капсулиране – Примери

SoftUni Foundation

```
abstract class MobilePhone
{
    public abstract void Calling();
    public abstract void SendSMS();
}
public class Nokia2700: MobilePhone
{
    public void FMRadio();
    public void MP3();
    public void Camera();
}
public class BlackBerry: MobilePhone
{
    public void FMRadio();
    public void MP3();
    public void Camera();
    public void Recording();
    public void ReadAndSendEmails();
}
```

2 обекта наследяват MobilePhone

```
public class Department
{
    private string departName;

    public Department(string departName)
    {
        this.DepartName = departName;
    }

    public string DepartName
    {
        get
        {
            return departName;
        }
        set
        {
            departName = value;
        }
    }
}
```

Капсулиране чрез използване на свойства

get accessor връща стойността на свойството

Свойството има два accessor-а: get и set

set accessor задава стойност на свойството

Полиморфизъм – Пример

Едно име, много форми: полиморфизъм

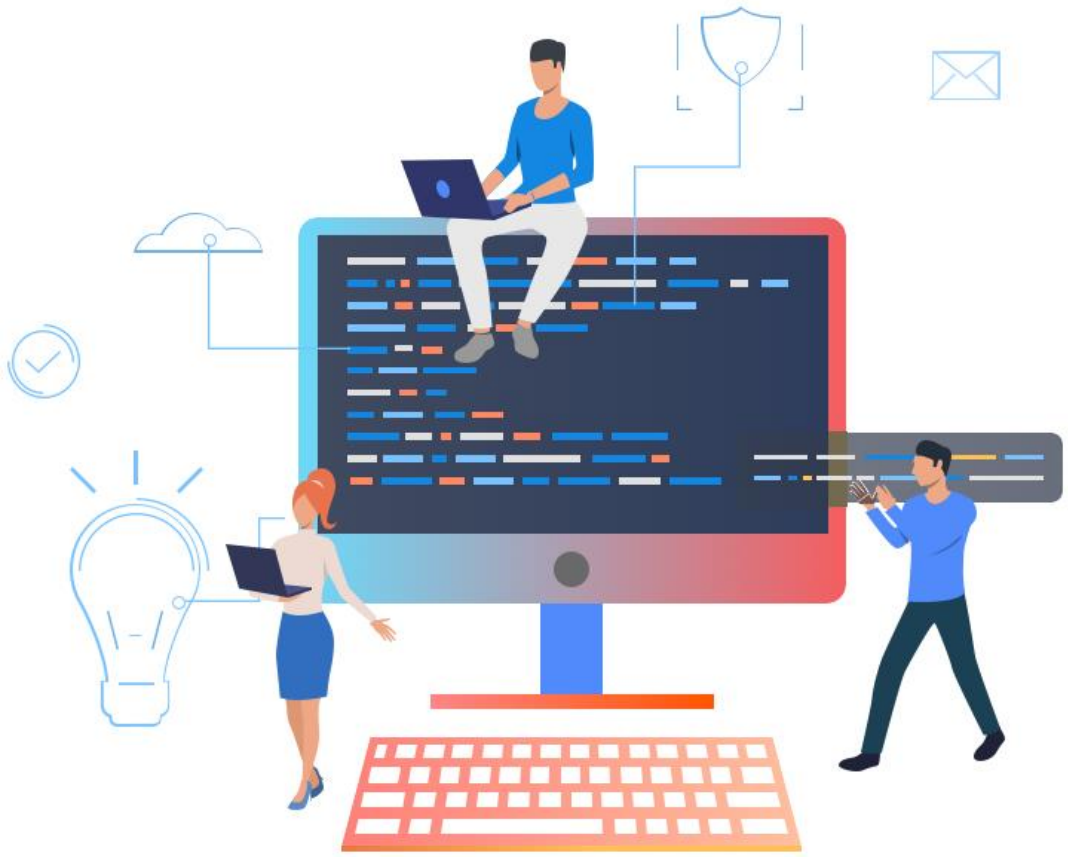
Класът Animal има виртуален метод

Класовете Pig и Dog наследяват Animal и могат да заменят неговия виртуален метод

```
class Animal
{
    public virtual void AnimalSound()
    {
        Console.WriteLine("The animal makes a sound");
    }
}

class Pig : Animal
{
    public override void AnimalSound()
    {
        Console.WriteLine("The pig says: wee wee");
    }
}

class Dog : Animal
{
    public override void AnimalSound()
    {
        Console.WriteLine("The dog says: bow wow");
    }
}
```

Класът Pig включва свойства и методи на класа Animal

Класът Dog включва свойства и методи на класа Animal

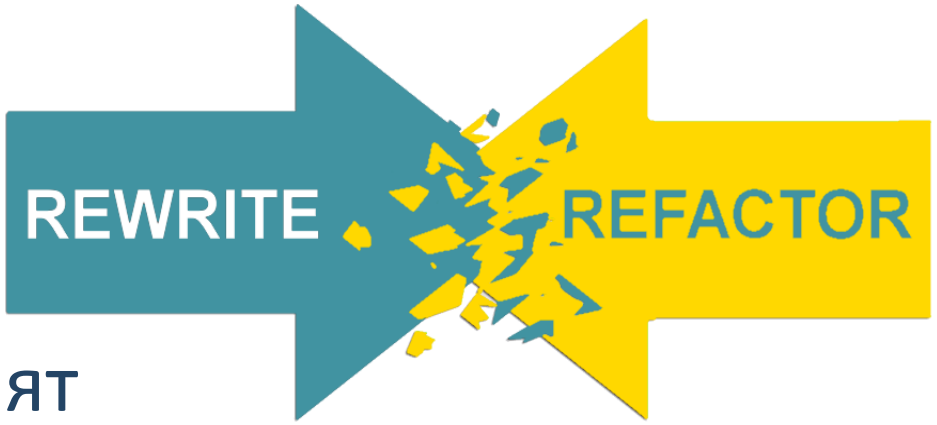


Кога и как преработваме кода?

Рефакториране на код

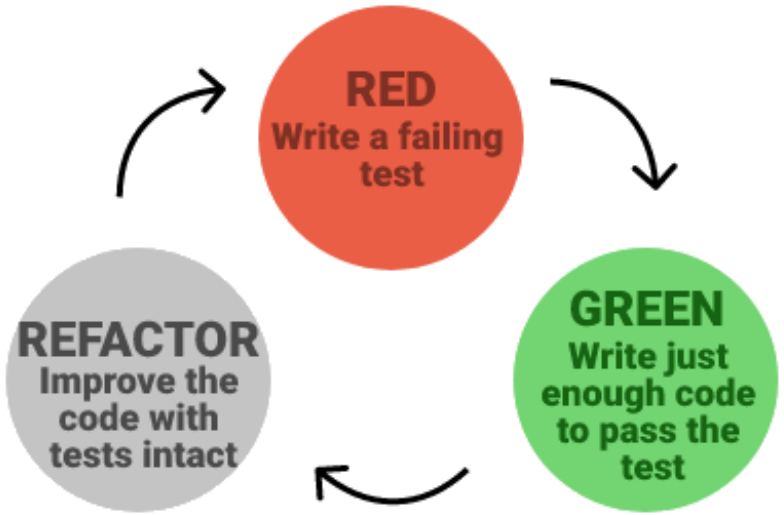
Какво е рефакториране (преработка) на код?

- Процес, който **стъпка по стъпка** променя **лошия код** в **добър код**
  - Без промяна на неговото външно поведение
  - Базирано на "**refactoring patterns**" → добре познат метод за подобряване на кода
- **Защо** се нуждаем от **преработване**?
  - Кодът **постоянно се променя** и качеството му се **влошава**
  - **Изискванията** постоянно се променят и **кодът** трябва да ги **следва**



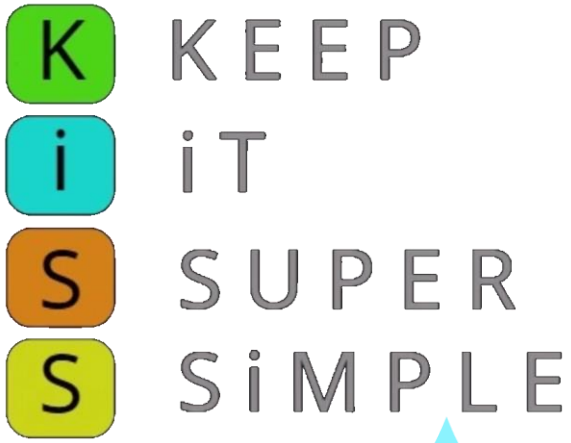
Кога преработваме код?

- Когато искаме да поправим **бъг**
- Когато разглеждаме **чужд код**
- Когато имаме **технически дълг**
- Когато правим **разработка**, **управлявана** от **тестове**
- **Компонентните тестове** гарантират, че преработването **не променя поведението**



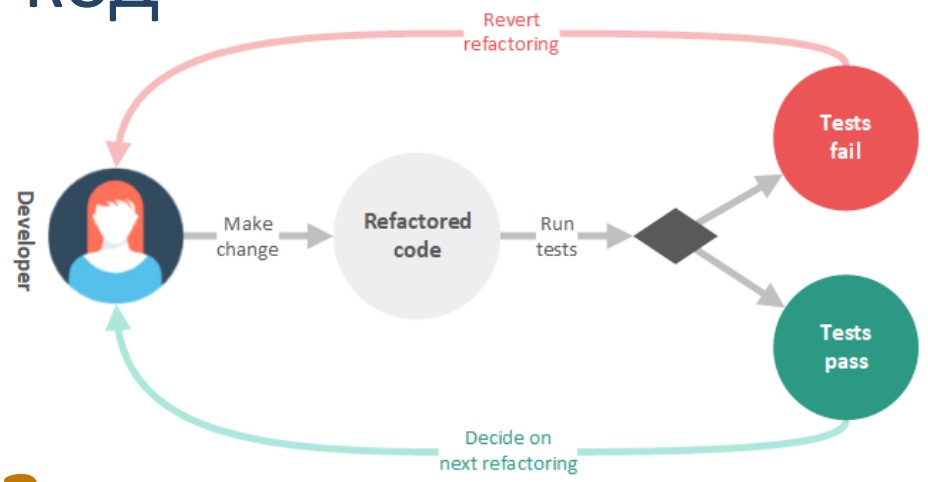
Преработване: основни принципи

- Поддържайте кода **прост** (KISS принцип)
- Избягвайте **повторението** (DRY principle)
- Направете го **експресивен**
  (самодокументиращ се, коментари и т.н.)
- **Разделяйте отговорностите** (decoupling)
- **Подходящо ниво** на **абстракция**
- Правило на бой скаутите
  - **Оставете** кода си **по-добре**, отколкото сте го **намерили**



Преработване: типични процеси

- Направете **резервно копие** на текущия код
- Добавете **тестове**, с които да **проверите работата** на преработения код
- При преработка правете **малки промени**
- Ако при **изпълнение** на **тестове** те са **неуспешни**, **премахнете преработката**



Шаблони за преработка

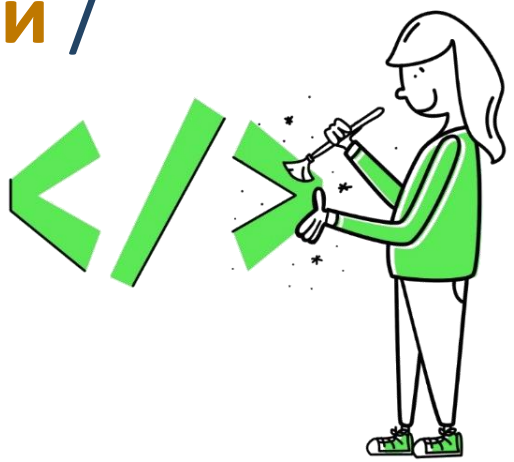
Утвърдени практики при рефакториране

Шаблони на преработване (1)

- При **повтарящ** се код → извадете **повтарящия** се **код** в отделен **метод**
- При **големи методи**→ **разделете** ги по **логика**
- При много **вложени структури** → извадете **част от кода** в **метод**
- **При слаба свързаност на отговорностите** → разделете на повече **класове** или **методи**
- При твърде **много параметри** в **метода** →създайте **клас**, който **групира** параметрите
- Ако методът вика повече **методи** от **друг клас**, отколкото от своя собствен → **преместете** го

Шаблон на преработване (2)

- При **силно функционално обвързване** → **слейте** двата класа
- При **магически числа** в кода →направете ги **константи**
- При **сложно булево условие** → разделете го на **няколко израза**
- При твърде **комплексна логика** на метода → **разделете** го на няколко **по-прости метода** или създайте **нов клас**
- При **неизползвани класове** / **методи** / **параметри** / **променливи** → **премахнете ги**



Нива на преработване

Нива на данни, твърдения, методи и класове

Преработване на ниво данни – Лош пример

Лошо инициализирани променливи

```
int[] nums = new int[10];
int sum = 0, avg = 0, low = 0, high = 0;
for (int i = 0; i < 10; i++)
{
    Console.Write($"Number {i + 1}: ");
    nums[i] =
int.Parse(Console.ReadLine());
    sum += nums[i];
}
avg = sum / 10;

for (int i = 0; i < 10; i++)
{
    if (nums[i] < avg)
    {
        low++;
    }
    else
    {
        high++;
    }
}
```

Конвертирайте примитивните данни в клас

Заменете магическото число с константа

Заменете израза с метод

Преработване на ниво данни – Добър пример

```
const int ArrayLength = 10;

int sum = 0;
int[] numbers = new int[ArrayLength];

for (int i = 0; i < numbers.Length; i++)
{
    Console.Write($"Number {i + 1}: ");
    numbers[i] = int.Parse(Console.ReadLine());
    sum += numbers[i];
}
int average = sum / numbers.Length;

int lowerThanAverage = 0;
int higherThanAverage = 0;

SeparateLowerAndHigherNumbersThanAverageValue
  (lowerThanAverage, higherThanAverage, numbers, average);
```

Няма магически числа

Добре наименувани променливи

Предишната част е заменена с метод

Преработване на ниво твърдения – Лош пример

```
if (date < SUMMER_START || date > SUMMER_END)
{
    charge = quantity * winterRate + winterServiceCharge;
}
else
{
    charge = quantity * summerRate;
}
```

Преместете сложния булев израз в добре наименувана булева функция

Използвайте break или return вместо контролна променлива за цикъл

```
string DailyGreetings(int dayTime)
{
    string greeting = "";

    if (dayTime <= 8)
    {
        greeting = "Good Morning";
    }
    if (dayTime >= 12)
    {
        greeting = "Good Afternoon";
    }
    …

    return greeting;
}
```

Заменете условията с полиморфизъм

Върнете се веднага, когато знаете отговора, вместо да задавате върната стойност

Преработване на ниво твърдения – Добър пример

Декомпозирани са сложните части на условния израз в отделни методи

```
if (isSummer(date))
{
    charge = SummerCharge(quantity);
}
else
{
    charge = WinterCharge(quantity);
}
```

```
string DailyGreetings(int dayTime)
{
    if (dayTime <= 8)
    {
        return "Good Morning";
    }
    if (dayTime >= 12)
    {
        return "Good Afternoon";
    }
    ...
    return "Good Nigth";
}
```

Връща се веднага, когато се разбере отговорът

Преработване на ниво методи – Лош пример

```
void PrtOwn()
{
    this.PrintBanner();

    Console.WriteLine("name: " + this.Name);
    Console.WriteLine("amount: " + this.GetAmount());
}
```

Преименувайте метода

Извлечете метод / вмъкнете метод

```
string FoundPerson(string[] people)
{
    for (int i = 0; i < people.Length; i++)
    {
        if (people[i].Equals("Don"))
        {
            return "Don";
        }
        if (people[i].Equals("John"))
        {
            return "John";
        }
    }
    return "Not Found";
}
```

Добавете / премахнете параметър

Комбинирайте сходни методи като ги параметризирате

Измислете по-прост алгоритъм

Разделете методите, чието поведение е зависимо от подадените параметри

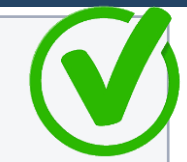
Подайте цял обект, а не конкретни полета

Преработване на ниво методи – Добър пример

По-добро име

Отделен метод

```
void PrintOwing()
{
    this.PrintBanner();
    this.PrintDetails();
}

void PrintDetails()
{
    Console.WriteLine("name: " + this.Name);
    Console.WriteLine("amount: " + this.GetAmount());
}
```



```
string FoundPerson(string[] people, string[] candidates)
{
    for (int i = 0; i < people.Length; i++)
    {
        if (candidates.Contains(people[i]))
        {
            return people[i];
        }
    }

    return "Not Found";
}
```

Добавя нов параметър

Опростен алгоритъм

Преработване на ниво класове – Лош пример

```
public class LocalCourse : Course, ILocalCourse
{
    public string Lab { get; set; }
    public override string ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.Append(this.GetType().Name);
        sb.AppendFormat("(Name={0}", this.Name);
        if (!(this.Teacher == null))
            sb.AppendFormat("; Teacher={0}", this.Teacher.Name);
        sb.AppendFormat("; Lab={0})", this.Lab);
        return sb.ToString();
    }
}
public class OffsiteCourse : Course, ILocalCourse
{
    public string Town { get; set; }

    public override string ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.Append(this.GetType().Name);
        sb.AppendFormat("(Name={0}", this.Name);
        if (!(this.Teacher == null))
            sb.AppendFormat("; Teacher={0}", this.Teacher.Name);
        sb.AppendFormat("; Town={0})", this.Town);
        return sb.ToString();
    }
}
```

Повтарящ се код

Повтарящ се код

Променете структурата на класа

Издигане на членовете в йерархията

Комбинирайте подобния код в супер клас

Използвайте наследяване

Преработване на ниво класове – Добър пример


```
public class LocalCourse : Course, ILocalCourse
{
    public string Lab { get; set; }

    public override string ToString()
    {
        return base.ToString() + "; Lab=" + this.Lab + ")";
    }
}

public class OffsiteCourse : Course, ILocalCourse
{
    public string Town { get; set; }

    public override string ToString()
    {
        return base.ToString() + "; Town=" + this.Town + ")";
    }
}
```

Когато **презаписвате** методи, извикайте **базовия метод**

Какво научихме днес?

SoftUni Foundation

- **Качествен код** – **коректен** и **четим**
- Използвайте смислени **имена** на променливи, параметри, методи и класове
- **Форматиране** – логическо разделяне на **свързани блокове от код** + **самодокументиращ** се код
- Пишете **кратки изрази** и използвайте **константи**, за да избегнете **магически стойности**
- **Чисто написани класове** – използвайте **OOP принципите**, **силен cohesion**, **слаб coupling**
- **Преработването** прави лошия код добър, като **не променя** неговото поведение