

# Качествени класове и йерархии от класове

## Утвърдени практики за обектно-ориентиран дизайн



Учителски екип

Обучение за ИТ кариера

<https://it-kariera.mon.bg/e-learning/>



```
public class Student
{
    private string name;
    private int age;
    ...
}
```

# Съдържание

## 1. Основни принципи

- Специализация, зависимост
- Абстракция, капсулиране, наследяване, полиморфизъм

## 2. Висококачествени класове

- Коректна употреба на ООП
- Клас методи, конструктори, данни
- Добра причина за създаването на класа

## 3. Типични грешки, които да избягвате в ООП



# Специализация (Cohesion)

- Специализацията показва доколко близки са всички процедури в клас или модул
  - Специализацията трябва да е висока
  - Класовете трябва да съдържат силно взаимосвързана функционалност и да се стремят да имат една-едничка цел (single purpose)
- Силната специализация е полезен инструмент за справяне със сложността
  - Добре дефинираните абстракции водят до силна специализация
  - Лошите абстракции са с малка специализация



# Силна специализация

- Пример за силна специализация

- Класът **System.Math**

- **Sin(), Cos(), Asin()**

- **Sqrt(), Pow(), Exp()**

- **Math.PI, Math.E**

```
double sideA = 40, sideB = 69;  
double angleAB = Math.PI / 3;  
double sideC = Math.Pow(sideA, 2) + Math.Pow(sideB, 2) -  
    2 * sideA * sideB * Math.Cos(angleAB);  
double sidesSqrtSum =  
    Math.Sqrt(sideA) + Math.Sqrt(sideB) + Math.Sqrt(sideC);
```



# Зависимост (Coupling)

- **Зависимостта** описва доколко здраво клас или процедура е свързана с други класове или процедури
- Зависимостта трябва да бъде държана **слаба**
  - Модулите би трябвало да зависят малко един от друг
  - Всички класове и процедури трябва да имат
    - Малки, преки, явни и гъвкави връзки с други класове / процедури
  - Един модул трябва лесно да може да бъде ползван в други модули, без сложни зависимости

# Слаба зависимость – пример

```
class Report
{
    public bool LoadFromFile(string fileName) {...}
    public bool SaveToFile(string fileName) {...}
}

class Printer
{
    public static int Print(Report report) {...}
}

class Program
{
    static void Main()
    {
        Report myReport = new Report();
        myReport.LoadFromFile("C:\\\\DailyReport.rep");
        Printer.Print(myReport);
    }
}
```



# Силна зависимость – пример

```
class MathParams
{
    public static double operand;
    public static double result;
}

class MathUtil
{
    public static void Sqrt()
    {
        MathParams.result = CalcSqrt(MathParams.operand);
    }
}

...

MathParams.operand = 64;
MathUtil.Sqrt();
Console.WriteLine(MathParams.result);
```



# Наследяване (Inheritance)

- **Наследяването** е способност на класа неявно да получи всички членове на друг-клас
  - Наследяването е основна концепция в ООП
  - Класът, чийто методи се наследяват, се нарича **базов** (родителски) клас
  - Класът, който получава нова функционалност, се нарича **производен** (дъщерен) клас
- Използвайте наследяването за:
  - **Многократна употреба на повтарящ се код**: данни и логика
  - Опростяване поддръжката на кода



# Полиморфизъм (Polymorphism)

- Полиморфизмът е основна концепция в ООП
- Способността да работим с обекти от даден клас както с екземпляри от неговия базов клас
  - За извикване на функционалност, скрита зад абстракция
- Полиморфизмът позволява да създадем йерархии с по-стойностна **логическа структура**
- Полиморфизмът е подход, позволяващ **многократната употреба на кода**
  - Общата логика се изнася в базовия клас
  - Специфичната логика се реализира в производния клас в презаписан метод

# Полиморфизъм

- В C# полиморфизмът се реализира чрез:
  - **Виртуални** методи
  - **Абстрактни** методи
  - **Интерфейси**
- **override** презаписва виртуален метод

# Полиморфизъм – пример

```
class Person
{
    public virtual void PrintName()
    {
        Console.Write("I am a person.");
    }
}

class Trainer : Person
{
    public override void PrintName()
    {
        Console.Write(
            "I am a trainer. " + base.PrintName());
    }
}

class Student : Person
{
    public override void PrintName()
    {
        Console.WriteLine("I am a student.");
    }
}
```




# Висококачествени класове: Абстракция (Abstraction)

- Present a consistent level of **abstraction** in the class contract (publicly visible members)
  - What abstraction the class is implementing?
  - Does it represent only **one thing**?
  - Does the class name well describe its **purpose**?
  - Does the class define clear and easy to understand public interface?
  - Does the class hide all its implementation details?



# Добра абстракция – пример

```
public class Font
{
    public string Name { get; set; }
    public float SizeInPoints { get; set; }
    public FontStyle Style { get; set; }
    public Font(string name, float sizeInPoints, FontStyle style)
    {
        this.Name = name;
        this.SizeInPoints = sizeInPoints;
        this.Style = style;
    }
    public void DrawString(DrawingSurface surface,
        string str, int x, int y) { ... }
    public Size MeasureString(string str) { ... }
}
```



# Лоша абстракция – пример

```
public class Program
{
    public string title;
    public int size;
    public Color color;
    public void InitializeCommandStack();
    public void PushCommand(Command command);
    public Command PopCommand();
    public void ShutdownCommandStack();
    public void InitializeReportFormatting();
    public void FormatReport(Report report);
    public void PrintReport(Report report);
    public void InitializeGlobalData();
    public void ShutdownGlobalData();
}
```

Този клас наистина ли представя „програма"? Това име добро ли е?



Този клас дали има една-едничка цел?

# Постигане на добра абстракция

- Define operations along with their opposites, e.g.
  - **Open()** and **Close()**
- Move unrelated methods in another class, e.g.
  - In class **Employee** if you need to calculate **Age** by given **DateOfBirth**
    - Create a static method **CalcAgeByBirthDate(...)** in a separate class **DateUtils**
- Group related methods into a single class
- Does the class name correspond to the class content?

## Постигане на добра абстракция (2)

- Beware of breaking the interface abstraction due to evolution
  - Don't add public members inconsistent with abstraction
  - Example: in class called **Employee** at some time we add method for accessing the DB with SQL

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName; { get; set; }
    ...
    public SqlCommand FindByPrimaryKeySqlCommand(int id);
}
```







# Капсулиране (Encapsulation)

- Minimize visibility of classes and members
  - In C# start from **private** and move to **internal**, **protected** and **public** if required
- Classes should **hide** their **implementation details**
  - A principle called **encapsulation** in OOP
  - Anything which is not part of the class interface should be declared **private**
  - Classes with good encapsulated classes are: less complex, easier to maintain, more loosely coupled
- Classes should keep their state clean → throw an exception if invalid data is being assigned

## Капсулиране (2)

- Never declare fields public (except constants)
  - Use properties / methods to access the fields
- Don't put private implementation details in the public interface
  - All public members should be consistent with the abstraction represented by the class
- Don't make a method public just because it calls only public methods
- Don't make assumptions about how the class will be used or will not be used

## Капсулиране (3)

- Don't violate encapsulation semantically!
  - Don't rely on non-documented internal behavior or side effects
  - Wrong example:
    - Skip calling **ConnectToDB()** because you just called **FindEmployeeById()** which should open connection 
  - Another wrong example:
    - Use **String.Empty** instead of **Titles.NoTitle** because you know both values are the same 

# Наследяване или включване (Containment)?

- Включването е връзка тип „той има“
  - Например: **Клавиатура** има множество **Клавиши**
- Наследяването е връзка тип „той е“
  - Проектиран за наследяване: направете класа **abstract**
  - Забрана за наследяване: направете го **sealed / final**
  - Подкласовете трябва да са ползваеми и през базовия клас
    - Без да се налага на потребителя да научава какви са разликите
  - Декларирайте инструменталните класове **static**



# Наследяване

- Не скривайте методи в подклас
  - Пример: ако класа **Timer** има public метод **Start()**, не дефинирайте private **Start()** в **AtomTimer**
- Преместете общите интерфейси, данни и поведение **толкова нагоре, колкото е възможно** в дървото на наследяването
  - Това максимизира многократното използване на кода
- Бъдете скептични към базови класове, които имат само един клас-наследник
  - Наистина ли е нужно още едно ниво на наследяване?

## Наследяване (2)

- Бъдете подозрителни към класове, които презаписват процедура и **не правят нищо** в нея
  - Дали коректно е ползвана тази процедура?
- Избягвайте прекалено **многократното наследяване**
  - Не създавайте повече от 6 нива на наследяване
- Избягвайте ползването на protected полетата за данни в наследения клас
  - По-добре добавете наследен protected метод / свойства

## Наследяване (3)

- Предпочитайте пред многократна проверка на типа:

```
switch (shape.Type)
{
    case Shape.Circle:
        ((Circle) shape).DrawCircle();
        break;
    case Shape.Square:
        ((Square) shape).DrawSquare();
        break;
    ...
}
```



- Помислете за наследяване на **Circle** и **Square** от **Shape** и презаписване на метода **Draw()**

# Клас-методи и данни

- Дръжте броят на методите в клас **възможно най-малък** → намалява се сложността
- Намалете директното извикване на методи на други класове
  - Намалете индиректното извикване на методи на други класове
  - По-малко викания на външни методи == **по-малка зависимост**
  - Известно също като "**fan-out**"
- Минимизирайте степента на взаимодействие на класа с други класове
  - Намалява се зависимостта между класовете



# Конструктори на класа

- Инициализирайте всички членове данни във всички конструктори, ако е възможно
  - Неинициализираните данни са предпоставка за грешки
  - Частично инициализираните са дори още по-лоши
  - Некоректен пример: присвоява **FirstName** в класа **Person** но оставя **LastName** празно
- Инициализирайте всички членове-данни в същия ред, в който са декларирани
- Предпочитайте deep copies пред shallow copies (**ICloneable** ще направи deep copy)

# Използвайте шаблони в дизайна

- Използвайте `private` конструктори, за да забраните директното създаване на инстанции на класа
- Използвайте шаблони в дизайна за класическите случаи
  - **Шаблони при създаването** като Singleton, Factory Method, Abstract Factory
  - **Шаблони в структурата** като Adapter, Bridge, Composite, Decorator, Façade
  - **Шаблони в поведението** като Command, Iterator, Observer, Strategy, Template Method

# Singleton шаблон

- **Singleton** клас е такъв клас, който трябва да има само един-единствен екземпляр
- Понякога Singleton погрешно е смятан за глобална променлива – не е!
- Възможни употреби:
  - Късно зареждане
  - Thread-safe
- <http://www.dofactory.com/net/singleton-design-pattern>

## Singleton

Type: Creational

### What it is:

Ensure a class only has one instance and provide a global point of access to it.

### Singleton

-static uniqueInstance  
-singletonData

+static instance()  
+SingletonOperation()

# Основни причини да създадете клас

- Моделиране на обекти от **реалния свят** чрез ООП класове
- Моделиране на **абстрактни** обекти, процеси и т.н.
- **Намаляване на сложността**
  - Работа на по-високо ниво
- **Изолиране на сложността**
  - Скрива я в клас
- Скрива детайлите по реализацията → **капсулиране**
- Намалява ефекта на промените
  - Промените засягат само съответния клас

# Основни причини да създадете клас (2)

- **Скрива** глобалните данни
  - Работи чрез методи
- **Групира** променливи, които се ползват заедно
- Създава централизирани точки за контрол
  - Една задача трябва да се изпълнява от едно място
  - Избягване на дублирането на код
- Улеснява **многократната употреба на кода**
  - С ползването на йерархии от класове и виртуални методи
- Пакетира свързаните операции на едно място



# Пространства от имена

- Групирайте свързаните класове в **пространства от имена**
- Следвайте една и съща конвенция в именуването

```
namespace Utils
{
    class MathUtils { ... }
    class StringUtils { ... }
}

namespace DataAccessLayer
{
    class GenericDAO<Key, Entity> { ... }
    class EmployeeDAO<int, Employee> { ... }
    class AddressDAO<int, Address> { ... }
}
```



# Множествено число в името на класа

- Никога не ползвайте множествено число в името на класа
  - Освен ако не са някакъв вид колекция!

- Лош пример:

```
public class Teachers : ITeacher
{
    public string Name { get; set; }
    public List<ICourse> Courses { get; set; }
}
```

Единствено число: Teacher  
(един учител, не няколко)



- Добър пример:

```
public class GameFieldConstants
{
    public const int MinX = 100;
    public const int MaxX = 700;
}
```



# Хвърляне на изключения без параметри

- Не хвърляйте изключения без параметри:

```
public ICourse CreateCourse(string name, string town)
{
    if (name == null)
    {
        throw new ArgumentNullException();
    }
    if (town == null)
    {
        throw new ArgumentNullException();
    }
    return new Course(name, town);
}
```



Кой параметър е  
**null** тук?

# Параметри, проверявани в Getter-а

- Проверка за невалидни данни да е в **setter**-и и **конструктори**
  - Не в **getter**!

```
public string Town
{
    get
    {
        if (string.IsNullOrEmpty(this.town))
            throw new ArgumentNullException();
        return this.town;
    }
    set
    {
        this.town = value;
    }
}
```

Преместете проверката в setter!



# Липсващ This за локалните членове

- Винаги ползвайте **this.XXX** вместо **XXX** за достъп до членовете на клас:

```
public class Course
{
    public string Name { get; set; }

    public Course(string name)
    {
        Name = name;
    }
}
```



Ползвайте **this.Name**

- StyleCop проверява за this и извежда предупреждение



# Празен низ за липсваща стойност

- Използвайте **null** когато липсва стойност, не **0** или **""**
  - Направете поле / свойство nullable, за да можете да ползвате **null** стойности или забранете липсата на стойности

- Лош пример:

Празното име е лоша идея! Ползвайте **null**

```
Teacher teacher = new Teacher("");
```



- Коректни алтернативи:

```
Teacher teacher = new Teacher();
```



```
Teacher teacher = new Teacher(null);
```



# Мистериозни числа в класовете

- Не използвайте „мистериозни“ числа
  - Особено ако класът има членове, свързани с тези числа:

```
public class Wolf : Animal
{
    ...
    bool TryEatAnimal(Animal animal)
    {
        if (animal.Size <= 4)
        {
            return true;
        }
    }
}
```



Това if условие е грешно. 4 е размера на **Wolf**, който има свойство **Size**, наследено от **Animal**. Защо не ползваме **this.Size** вместо 4?

# Не се вика базовия конструктор

- Извикайте базовия конструктор за да **се възползвате** от инициализацията на състоянието на обекта:

```
public class Course
{
    public string Name { get; set; }
    public Course(string name) { this.Name = name; }
}

public class LocalCourse : Course
{
    public string Lab { get; set; }
    public Course(string name, string lab) {
        this.Name = name;
        this.Lab = lab;
    }
}
```



: base(name)

Извикайте вместо това  
базовия конструктор!

# Повтаряне на код в базовия и дъщерните класове

- Никога не копирайте код от базовия в наследения клас

```
public class Course
{
    public string Name { get; set; }
    public ITeacher Teacher { get; set; }
}

public class LocalCourse : Course
{
    public string Name { get; set; }
    public ITeacher Teacher { get; set; }
    public string Lab { get; set; }
}
```



Защо тези полета са дублирани,  
вместо да са наследени?

# Лошо капсулиране чрез конструктор без параметри

- Погрижете се полетата да са добре капсулирани

```
public class Course
{
    public string Name { get; private set; }
    public ITeacher Teacher { get; private set; }
    public Course(string name, ITeacher teacher)
    {
        if (name == null)
            throw ArgumentNullException("name");
        if (teacher == null)
            throw ArgumentNullException("teacher");
        this.Name = name;
        this.Teacher = teacher;
    }
    public Course() { }
}
```

Валидация в setter-а

Нарушава капсулирането:  
**Name & Teacher** ще са **null**.





# Зависимост на базовия клас от наследниците му

- Базовият клас не трябва **никога** да знае за наследниците си!

```
public class Course
{
    public override string ToString()
    {
        StringBuilder result = new StringBuilder();
        ...
        if (this is ILocalCourse)
        {
            result.Append("Lab = " + ((ILocalCourse)this).Lab);
        }
        if (this is IOffsiteCourse)
        {
            result.Append("Town = " + ((IOffsiteCourse)this).Town);
        }
        return result.ToString();
    }
}
```



# Скрито третиране на базов клас като наследник

- Не дефинирайте **IEnumerable<T>** полета, които после ще ползвате като **List<T>** (нарушена абстракция)

```
public class Container<T>
{
    public IEnumerable<T> Items { get; private set; }
    public Container()
    {
        this.Items = new List<T>();
    }
    public void AddItem (T item)
    {
        (this.Items as List<T>).Add(item);
    }
}
```



Лоша практика: скрит  
**List<T>**

## Скрито третиране на базов клас като наследник (2)

- Използвайте **List<T>** за полето и върнете него там, където се изисква **IEnumerable<T>**:

```
public class Container<T>
{
    private List<T> items = new List<T>();
    public IEnumerable<T> Items
    {
        get { return this.items; }
    }
    public void AddItem (T item)
    {
        this.items.Add(item);
    }
}
```

Това частично нарушава капсулацията. Помислете за **клонирание**, за да избегнете опасност от промяна на елементите.

# Повтарящ се код не е преместен нагоре в йерархията

```
public abstract class Course : ICourse
{
    public string Name { get; set; }
    public ITeacher Teacher { get; set; }
}
```

```
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();
```

```
}
```

```
}
```



Повтаряне на код

// Продължава на другия слайд

## Повтарящ се код не е преместен нагоре в йерархията(2)

- При презаписване (overriding) на методи, извикайте базовия метод ако ви трябва функционалността му, не я копирайте!

```
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();
    }
}
```




Повтаряне на код



# Преместване на повтарящ се код нагоре в йерархията

```
public abstract class Course : ICourse
{
    public string Name { get; set; }
    public ITeacher Teacher { 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);
        return sb.ToString();
    }
}
```




*Продължава на другия слайд*

## Преместване на повтарящ се код нагоре в йерархията (2)

```
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 + ")";
    }
}
```



# Обобщение

## 1. Проектиране на класове

- Използвайте коректно принципите на ООП
  - **Абстракция** – използвайте сходно ниво на абстракция в целия проект
  - **Наследяване** – не повтаряйте код
  - **Капсулиране** – подсигурете винаги валидно състояние на обектите
  - **Полиморфизъм** – показвайте ясно логическата структура на кода
- **Осигурете силна специализация и слаба зависимост**
- Използвайте шаблони в дизайна ако е нужно



Качествени класове



Въпроси?





# Министерство на образованието и науката (МОН)

- Настоящият курс (презентации, примери, задачи, упражнения и др.) е разработен за нуждите на Национална програма "**Обучение за ИТ кариера**" на МОН за подготовка по професия "Приложен програмист"



Министерство  
на образованието  
и науката



Национална  
програма  
„Обучение за  
ИТ кариера“

- Курсът е базиран на учебно съдържание и методика, предоставени от **фондация "Софтуерен университет"** и се разпространява под свободен лиценз **CC-BY-NC-SA**



SoftUni  
Foundation

