

Качествени методи

Дизайн и реализация на качествени методи. Специализация и зависимост



Учителски екип

Обучение за ИТ кариера

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



Съдържание

- Защо въобще се нуждаем от методи?
- Специализация и зависимост на ниво метод
 - Силна специализация
 - Слаба зависимост
- Параметри на метода
- Псевдокод



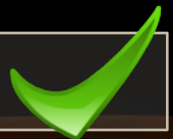
Защо се нуждаем от методи?

- Методите (функции, процедури) са важна част от разработката на софтуер
 - Намаляване на сложността
 - „Разделяй и владей“
 - Сложните задачи се разделят на сбор от по-прости задачи
 - Подобряване на четливостта на кода
 - Малки методи с подходящи имена правят кода самоописателен
 - Избягване на повторенията в кода
 - Програмен код с повторение е труден за поддръжка

Защо се нуждаем от методи?(2)

- Методите **улесняват** разработката на софтуера
 - Скриване на детайлите по реализацията
 - Сложната логика е капсулирана и скрита зад прост интерфейс
 - Алгоритми и структури от данни са скрити и може после лесно да бъдат сменени с други
 - Увеличава се нивото на абстракция
 - Методите адресират конкретния бизнес проблем, а не техническата реализация:

```
Bank.Accounts[customer].Deposit(500);
```



Ползване на методи: Основни положения

- Основният принцип за коректно използване на методи :

Методът трябва да върши онова, което твърди името му или да сигнализира, че има грешка (като хвърли изключение).
Всяко друго поведение е некоректно!

- Методът трябва да върши точно онова, **което казва името му**
 - Нищо по-малко (т.е. да работи коректно при всички случаи)
 - Нищо повече (т.е. без странични ефекти)
- В случай на некоректен вход или некоректни предпоставки
 - Трябва да върне грешка (например като хвърли изключение)

Лоши методи – примери

```
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



Какво ще стане, ако $a = b = c = -1$?

```
double CalcTriangleArea(double a, double b, double c)
{
    double s = (a + b + c) / 2;
    double area = Math.Sqrt(s * (s - a) * (s - b) * (s - c));
    return area;
}
```

Същото, както ако $a = b = c = 1 \rightarrow$ и двата
триъгълника ще са с еднакво лице.



Добри методи – примери

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



```
double CalcTriangleArea(double a, double b, double c)
{
    if (a <= 0 || b <= 0 || c <= 0)
    {
        throw new ArgumentException("Sides should be positive.");
    }
    double s = (a + b + c) / 2;
    double area = Math.Sqrt(s * (s - a) * (s - b) * (s - c));
    return area;
}
```



Съобщаване за грешки

- Някои методи не дават коректна индикация за грешки

```
internal object GetValue(string propertyName)
{
   PropertyDescriptor descriptor =
        this.propertyDescriptors[propertyName];

    return descriptor.GetDataBoundValue();
}
```




- Ако името на свойството не съществува
 - Ще бъде хвърлено null reference изключение (индиректно)
→ не е много говорящо

Съобщаване за грешки (2)

- По-добре е коректно да се обработят изключенията:

```
internal object GetValue(string propertyName)
{
   PropertyDescriptor descriptor =
        this.propertyDescriptors[propertyName];
    if (descriptor == null)
    {
        throw new ArgumentException("Property name: "
            + propertyName + " does not exists!");
    }

    return descriptor.GetDataBoundValue();
}
```



Симптоми на сгрешени методи

- Метод, правещ нещо различно от това, което твърди името му, е грешен поради поне една от тези три причини:
 - Методът понякога връща некоректен резултат → **грешка**
 - Методът връща неверен резултат при некоректен вход → **некачествен**
 - Може да е донякъде приемливо само за private методи
 - Методът прави твърде много неща → **лоша специализация**
 - Методът има странични ефекти → **спагети код**
 - Методът връща странна стойност при грешка → **може да е индикация за грешки в кода**

Сгрешени методи – пример

```
long Sum(int[] elements)
{
    long sum = 0;
    for (int i = 0; i < elements.Length; i++)
    {
        sum = sum + elements[i];
        elements[i] = 0; // Hidden side effect
    }
    return sum;
}
```



```
double CalcTriangleArea(double a, double b, double c)
{
    if (a <= 0 || b <= 0 || c <= 0)
    {
        return 0; // Incorrect result
    }
    double s = (a + b + c) / 2;
    double area = Math.Sqrt(s * (s - a) * (s - b) * (s - c));
    return area;
}
```



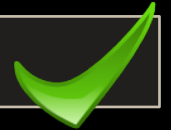
Силна специализация (Strong Cohesion)

- Методите трябва да имат силна специализация
 - Да вършат една работа и да я вършат добре
 - Трябва да имат ясно намерение
- Методите, които се опитват да правят едновременно няколко неща е трудно да бъдат наименувани
- Силната специализация се използва в инженерството
 - В РС хардуера всеки компонент решава една задача
 - Например харддискът извършва едно нещо – съхранение

Приемливи типове специализация

- **Функционална специализация** (независима функция)
 - Методът извършва някакво добре дефинирано изчисление и връща един резултат
 - Целият вход се подава чрез параметри и целият изход се връща като резултат
 - Няма външни зависимости или странични ефекти

`Math.Sqrt(value) → square root`



`char.IsLetterOrDigit(ch)`



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



Приемливи типове специализация (2)

■ Последователна специализация (алгоритъм)

- Методът изпълнява точно определена поредица от операции, за да извърши една задача и да получи определен резултат
 - Капсулира алгоритъм

■ Пример:

```
SendEmail(recipient, subject, body)
```



1. Свързване със сървъра на пощата
2. Изпращане на хедъра на съобщението
3. Изпращане на тялото на съобщението
4. Прекъсване на връзката със сървъра на пощата

Приемливи типове специализация (3)

■ Комуникационна специализация (обща данни)

- Набор от операции, използвани за преработката на определени данни и произвеждане на някакъв резултат

■ Пример:

```
DisplayAnnualExpensesReport(int employeeId)
```



1. Извличане на входни данни от базата данни
2. Извършване на вътрешни пресмятания с извлечените данни
3. Изграждане на доклад
4. Форматиране на доклада като работен лист в Excel
5. Показване на работния лист на екрана

Приемливи типове специализация (4)

■ Времева специализация (действия, свързани във времето)

- Операции, които не са свързани, но трябва да се случат в някакъв определен момент

■ Примери:

`InitializeApplication()`



1. Зареждане на потребителските настройки
2. Проверка за подобрения
3. Зареждане на всички фактури от базата данни

`ButtonConfirmClick()`



- Поредица от действия, изпълняващи се в дадения случай

Неприемлива специализация

- Логическа специализация

- Извършва различни операции в зависимост от входния параметър
- Лош пример:

```
object ReadAll(int operationCode)
{
    if (operationCode == 1) ... // Read person name
    else if (operationCode == 2) ... // Read address
    else if (operationCode == 3) ... // Read date
    ...
}
```



- Допустима при манипулатори на събития
 - Напр. събитието **KeyDown** в Windows Forms

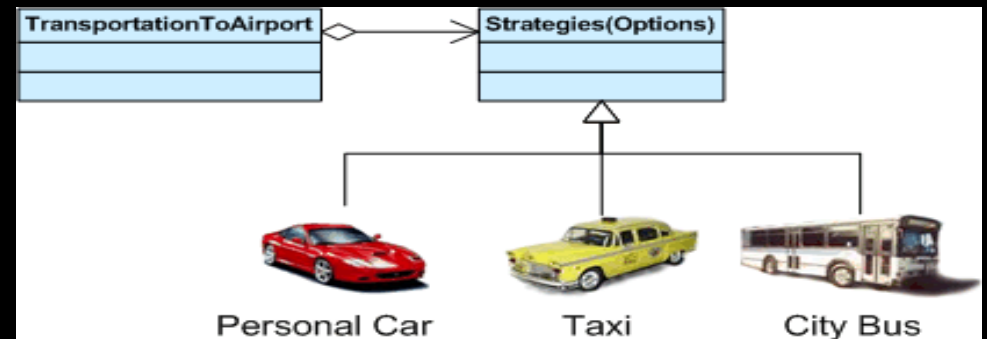
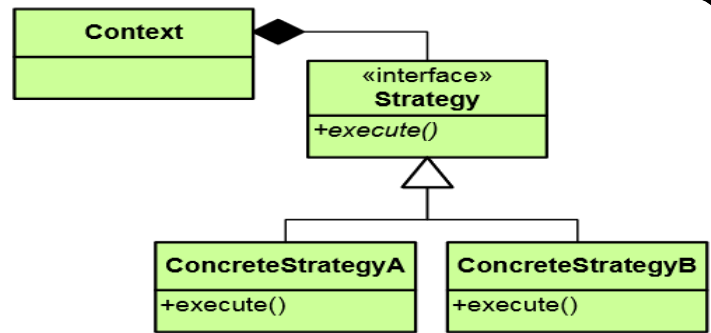
Шаблон Strategy

- Капсулира алгоритъм в някакъв клас
 - Така всеки алгоритъм е заменен от други
 - Всички алгоритми могат да работят прозрачно с еднакви данни
 - Клиентът може да работи прозрачно с всеки алгоритъм

Strategy

Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



- <http://www.dofactory.com/net/strategy-design-pattern>

Шаблон Strategy – пример

```
abstract class SortStrategy {  
    public abstract void Sort(IList<object> list);  
}
```

```
class QuickSort : SortStrategy {  
    public override void Sort(IList<object> list) { ... }  
}
```

```
class MergeSort : SortStrategy {  
    public override void Sort(IList<object> list) { ... }  
}
```

```
class SortedList {  
    private IList<object> list = new List<object>();  
    public void Sort(SortStrategy strategy) {  
        // sortStrategy can be passed in constructor  
        sortStrategy.Sort(list);  
    }  
}
```

Неприемлива специализация

■ Случайна специализация (spaghetti)

- Несвързани (случайни) операции, групирани в метод по неясна причина
- Лош пример:

```
HandleStuff(customerId, int[], ref sqrtValue, mp3FileName, emailAddress)
```



1. Подготвяне на доклад за годишните приходи на даден купувач
2. Подреждане на масив цели числа във възходящ ред
3. Пресмятане на квадратния корен на дадено число
4. Конвертиране на даден MP3 файл в WMA формат
5. Изпращане на имейл на даден купувач

Слаба зависимост (Loose Coupling)

- Какво е слаба зависимост?
 - Минимална зависимост на метода от други части на програмния код
 - Минимална зависимост на членовете на класа или на външни класове и техните членове
 - Без странични ефекти
 - Ако зависимостта е слаба, можем лесно да използваме метод или група от методи отново за нов проект
- Силна зависимост → спагети код

Слаба зависимост (2)

- Идеалната зависимост

- Методът зависи само от параметрите си
- Няма друг вход или изход
- Пример: **Math.Sqrt()**

- В реалния свят

- Сложният софтуер не може да избегне зависимостта, но може да я направи възможно най-слаба
- Пример: сложен криптиращ алгоритъм, извършващ инициализация, криптиране, финализиране

Зависимост – пример

- Нарочно увеличена зависимост за по-голяма гъвкавост (.NET cryptography API):

```
byte[] EncryptAES(byte[] inputData, byte[] secretKey)
{
    Rijndael cryptoAlg = new RijndaelManaged();
    cryptoAlg.Key = secretKey;
    cryptoAlg.GenerateIV();
    MemoryStream destStream = new MemoryStream();
    CryptoStream csEncryptor = new CryptoStream(
        destStream, cryptoAlg.CreateEncryptor(),
        CryptoStreamMode.Write);
    csEncryptor.Write(inputData, 0, inputData.Length);
    csEncryptor.FlushFinalBlock();
    byte[] encryptedData = destStream.ToArray();
    return encryptedData;
}
```



Слаба зависимост – пример

- За да намалим зависимостта може да правим **ПОМОЩНИ КЛАСОВЕ**
 - Скрийте сложната логика и представете прост, пряк интерфейс (т.е. **фасада**):

```
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;
}
```



Силна зависимост – пример

- Предаване на параметри през полета на класове
 - Типичен пример за силна зависимост
 - Не правете това без добра причина!

```
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());
    }
}
```



Силна зависимост в реално ползван код

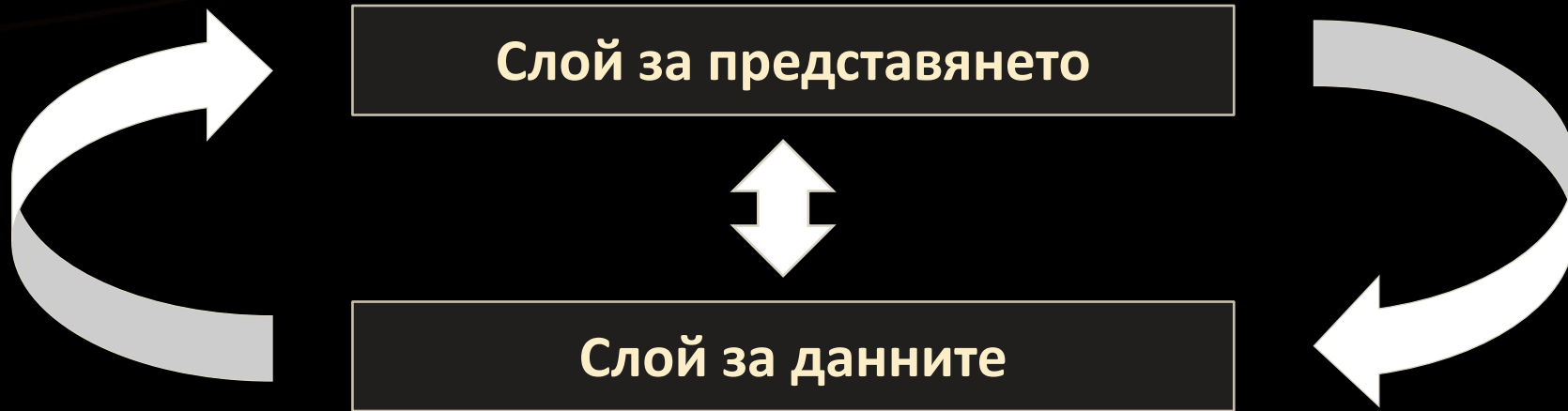
- Да кажем, че имаме голяма част софтуер
 - Трябва да обновим подсистемите, а те не са точно независими
 - Т.е. промяна във филтрирането ще засегне сортирането и т.н.:

```
class GlobalManager
{
    public void UpdateSorting() {...}
    public void UpdateFiltering() {...}
    public void UpdateData() {...}
    public void UpdateAll () {...}
}
```



Проблеми със зависимостта в реално ползван код

- Нека имаме приложение, състоящо се от два слоя:



- Не обновявайте отгоре-надолу и отдолу-нагоре от един метод!
 - Напр. методът **RemoveCustomer()** в **DataLayer** ще промени и слоя за представянето
 - По-добре използвайте известие (шаблон observer / събитие)

Слаба зависимост и OOP

- Намаляване на зависимостта с OOP техники
 - Абстракция
 - Дефинирайте public интерфейс и скрийте детайлите по реализацията
 - Капсулиране
 - Направете методите и полетата private освен ако със сигурност не е необходимо да са други
 - Дефинирайте нови членове като private
 - Увеличете видимостта им веднага щом ви потребват

Допустима зависимост

- Метод, зависим от параметрите си

- Това е най-добрия тип зависимост

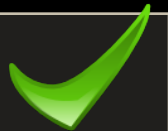
```
static int Sum(int[] elements) { ... }
```



- Метод в клас, свързан с няколко полета на класа

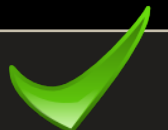
- Тази зависимост е обичайна, не се тревожете много

```
static int CalcArea()  
{ return this.Width * this.Height; }
```



- Метод в клас, зависим от static методи, свойства или константи
във външен клас

```
static double CalcCircleArea(double radius)  
{ return Math.PI * radius * radius; }
```



Недопустима зависимост

- Метод в клас, зависим от **static полета** във външен клас
 - Ползвайте `private` полета и `public` свойства
- Методи, взимащи като входни данни някакви полета, които биха могли да се подадат като параметри
 - Проверете целта на метода
 - Той ли е направен да преработва данни от вътрешен клас или помощният метод?
- Метод, дефиниран като `public` без да е част от интерфейса на `public` класа → възможна зависимост

Параметри на методи

- Слагайте по-важните параметри по-напред
 - Сложете основните входни параметри първи
 - Сложете неважните евентуални параметри последни
- Пример:

```
void RegisterUser(string username, string password,  
                  Date accountExpirationDate, Role[] roles)
```



- Лош пример:

```
void RegisterUser(Role[] roles, string password, string username,  
                  Date accountExpirationDate)
```



```
void RegisterUser(string password, Date accountExpirationDate,  
                  Role[] roles, string username)
```



Параметри на методи (2)

- Не променяйте входните параметри
 - Вместо това използвайте нови променливи
 - Лош пример:

```
bool CheckLogin(string username, string password)
{
    username = username.ToLower();
    // Check the username / password here ...
}
```



- Добър пример:

```
bool CheckLogin(string username, string password)
{
    string usernameLowercase = username.ToLower();
    // Check the username / password here ...
}
```



Параметри на методи (3)

- Бъдете **последователни** в употребата на параметри
 - Ползвайте същите имена и ред във всички методи
 - Лош пример:

```
void EncryptFile(Stream input, Stream output, string key);  
void DecryptFile(string key, Stream output, Stream input);
```



- Изходните параметри слагайте последни

```
FindCustomersAndIncomes(Region region,  
    out Customer[] customers, out decimal[] incomes)
```



Подаване на цял обект като параметър или само на полетата му?

- Кога трябва да подадем като параметър обект, съдържащ няколко стойности, и кога стойностите поотделно?
 - Понякога подаваме обект и използваме само едно негово поле
 - Това добра практика ли е?
 - Примери:

```
CalculateSalary(Employee employee, int months);
```

```
CalculateSalary(double rate, int months);
```

- Вижте нивото на абстракция на метода
 - Със служители ли е направен да оперира или с проценти и месеци?
→ първото е грешно

Колко параметъра трябва да има един метод?

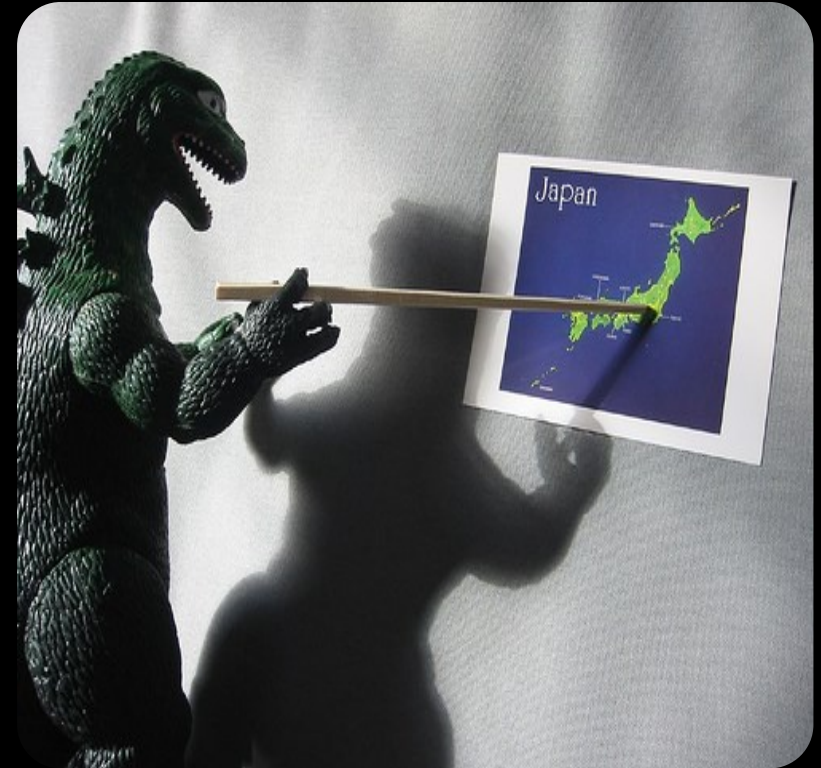
- Ограничете броя параметри до 7 (+/-2)
 - 7 е „магическо“ число в психологията
 - Човешкият мозък не може да обработи повече от 7 (+/-2) неща едновременно
- Ако параметрите трябва да са твърде много, преосмислете целта на метода
 - Тя ясна ли е?
 - Обмислете извличането на няколко параметъра в нов клас

Дължина на метода

- Колко дълъг трябва да е един метод?
 - Няма конкретно ограничение
 - Избягвайте методи, по-дълги от **един екран (30 реда)**
 - Дългите методи невинаги са лоши
 - Уверете се че имате добра причина за дължината им
 - **Специализацията и зависимостта** са по-важни от дължината на метода!
 - Дългите методи често съдържат части, които могат да се извлекат като отделни методи с добри имена и ясна цел

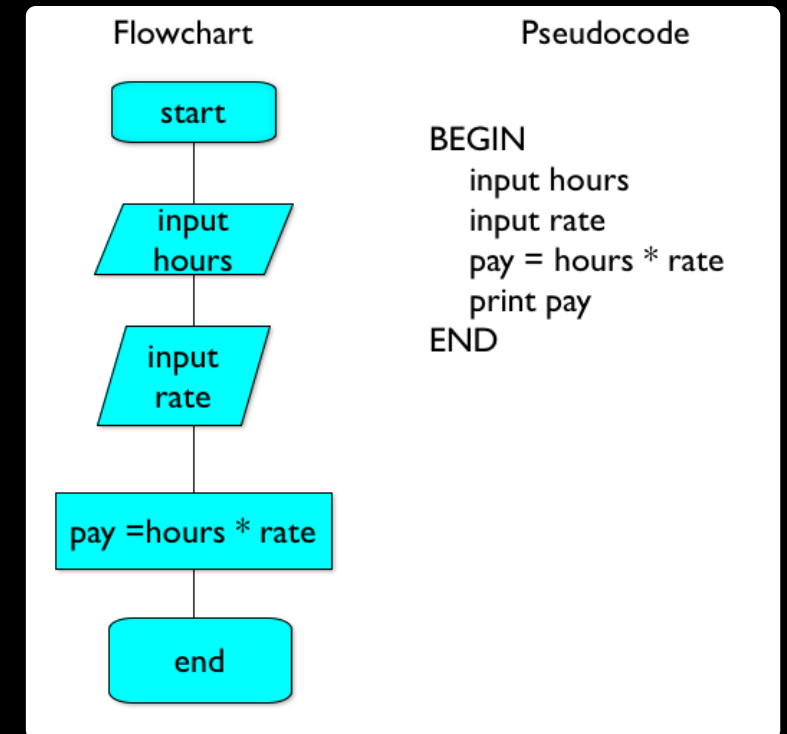
Псевдокод

- Псевдокодът може да помогне при:
 - Проектиране на подпрограми
 - Писане на подпрограми
 - Проверка на кода
 - Почистване на недостижими клонове от подпрограмата



Дизайн чрез псевдокод

- Каква ще е абстракцията в подпрограмата, т.е. каква информация ще **скрие**?
- Входни параметри на подпрограмата
- Изход на подпрограмата
- Предусловия
 - Условия, които трябва да са верни преди подпрограмата да се извика
- Постусловия
 - Условия, които трябва да са верни след изпълнение на подпрограмата



Дизайн преди писане на кода

- Защо е по-добре да отделите време на дизайна преди да започнете да пишете кода?
 - Функционалността може вече да е достъпна в библиотека и изобщо да няма нужда да пишете код!
 - Помислете за най-добрия начин да реализирате задачата с оглед на изискванията на проекта си
 - Ако не успеете да напишете кода вярно от първия път, знайте, че програмистите стават емоционални към кода си

Псевдокод – пример

Routine that evaluates an aggregate expression for a database column (e.g. Sum, Avg, Min)

Parameters: Column Name, Expression

Preconditions:

- (1) Check whether the column exists and throw an argument exception if not
- (2) If the expression parser cannot parse the expression throw an ExpressionParsingException

Routine code: Call the evaluate method on the DataView class and return the resulting value as string

Public подпрограми в библиотеки

- Public подпрограмите в библиотеките и в системния софтуер са трудни за промяна
 - Защото купувачите **не искат блокиращи промени**
- Две причини да трябва да промените public подпрограма:
 - Трябва да се добави нова функционалност, противоречаща на старите характеристики
 - Името е объркващо и прави използването на библиотеките неинтуитивно или неудобно
- Предварително проектирайте по-добре или променяйте внимателно

Неизползваеми методи

- **Неизползваем (deprecated) метод**
 - Ще е премахнат в бъдещи версии
- Когато отбелязвате стар метод като неизползваем
 - Включете това в документацията
 - **Посочете новия метод, който ще се използва**
- Ползвайте атрибута **[Obsolete]** в .NET

```
[Obsolete("CreateXml() method is deprecated.  
Use CreateXmlReader instead.")]  
public void CreateXml (...) { ... }
```

Вградени подпрограми

- Вградени подпрограми (в C / C++) дават два плюса:
 - Подобрява изпълнението, защото не се създава нова подпрограма в стека
 - Абстракция – използва добре именувана подпрограма вместо вграден код
- Някои приложения (напр. **игри**) се нуждаят от такава оптимизация
 - Ползва се за най-често използваните подпрограми
 - Пример: кратка подпрограма, извикана 100,000 пъти
- Не всички езици поддържат вградени подпрограми
 - Когато е нужно, C# компилаторът вгражда подпрограми по време на компилирането

Рекурсия

- Полезна е, когато искате да обходите дървовидни или графовидни структури
- Внимавайте с безкрайната и индиректната рекурсия
- Пример за рекурсия:

```
void PrintWindowsRecursive(Window w)
{
    w.Print()
    foreach(childWindow in w.ChildWindows)
    {
        PrintWindowsRecursive(childWindow);
    }
}
```



Съвети за рекурсията

- Уверете се, че рекурсията има край (дъно)
- Потвърдете, че рекурсията не е много „скъпа“
 - Проверете заетите системни ресурси
 - Винаги може да използвате стекове и итерации
- Не ползвайте рекурсия, когато има по-добро **линейно** (базирано на итерации) решение, напр.
 - Факториели
 - Числа на Фибоначи
- Някои програмни езици оптимизират извикването на рекурсии

Обобщение

1. Дизайн на методи

- Няма едно-единствено решение
- Има много компромиси

2. Избиране на най-добрия подход

- Преценете изискванията
- Изберете най-подходящото решение
- Опитайте да сте гъвкави ако изискванията се променят
- Подсигурете силна специализация и слаба зависимост



Качествени методи



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

