Теория ООП програмиране

1. Документация на кода
   * Ефективни коментари - коментари не повтарят кода, обясняват го на по-високо ниво и разкриват неочевидни детайли
   * Най-добрата софтуерна документация е самият програмен код – трябва да е изчистен и четлив!
   * Самоописателният код се подразбира и не се нуждае от коментари (Прост дизайн, малки добре именувани методи, силна свързаност и слаба зависимост, проста логика, удачни имена на променливите, добро форматиране).
   * Коментира се целта на кода, а не детайли по изпълнението. Използват се коментари,за да се подготви читателя за това, което следва. Коментира се всичко, което е заобиколно решение на проблем или пък е недокументирана функция
   * Описва се идеята за дизайна на класа, ограничения, изисквания за работа, целта и съдържанието на всеки файл
   * Не се документират детайли по изпълнението в интерфейса на класа
2. Полезни практики при инициализиране на променливи
   * Инициализират се променливи използвани в цикъл непосредствено преди него
   * Не се задава на променлива стойност докато не трябва да я използвате
   * Започва се с най-ограничената видимост
   * Променливите трябва да имат една-единствена цел
   * В C# има два типа константи
     + Compile-time константи:
       - Заменят се със стойността си по време на компилация
       - Зад тях не стои никакво поле
     + Run-time константи:
       - Специални полета, инициализирани в static конструктора
       - Компилират се в асемблито като всеки друг член на клас
   * Константите трябва да се използват в следните случаи:
     + Когато трябва да използваме имена или други стойности и техните логически смисъл и стойност не са очевидни
     + Когато имаме **magic numberи** – при многократно използване на една и съща променлива, тя може да се направи като константа.
     + Константа се използва и за да не се хардкоудва дадена променлива, използва се направо константата.
   * Константи не се ползват, когато:
     + За интернационализация Ползват се ресурси, не константи
     + Ресурсите са специални файлове, вградени в асемблито достъпни са по време на изпълнение на програмата
3. Подредба и четимост на кода
   * Многократното влагане на условни команди и цикли прави кода неясен
     + Повече от 2-3 нива е твърде много
     + Многократно вложения код е сложен и труден за четене и разбиране
     + Обикновено може да преместите части от кода в отделни методи
     + Това опростява логиката на кода
     + Добрите имена на методи правят кода самоописателен
   * Когато редът на командите не е важен
     + Нека четенето на кода отгоре до долу да е като четенето на вестник
     + Групират се свързани команди заедно
     + Поставят се ясни граници при зависими команди
     + Ползват се празни редове за да отделите зависимостите
     + Ползват се в отделен метод
   * Избирането на правилния тип цикъл:
     + Ползват се цикъл **for** да повторите блок от код определен брой пъти
     + Ползват се цикъл **foreach** да извършите действие с всеки елемент от масив или колекция
     + Ползват се циклите **while** / **do-while** когато не знаете колко пъти трябва да повторите блока команди
     + Избягват се многократното влагане на цикли
     + Может да се изнесе тялото на цикъла в нов метод
   * Нека циклите са прости
     + Това подпомага тези, които ще четат вашия код
     + Отнасяйте се с вътрешността на цикъла като с процедура
     + Не карайте четящия да гледа в цикъла, за да разбере управлението му
     + Отнасяйте се към цикъла като към „черна кутия“
   * Опитвайте да направите циклите толкова кратки, че да се виждат наведнъж (на един екран)
     + Ползват се методи да съкратите тялото на цикъла
     + Правят дълги цикли особено ясни
     + Избягва се многократното влагане на цикли
4. Коректност на методите
   * 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)
   * 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
   * 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 PC хардуера всеки компонент решава една задача
     + Например харддискът извършва едно нещо – съхранение
   * Последователна специализация (алгоритъм)
     + Методът изпълнява точно определена поредица от операции, за да извърши една задача и да получи определен резултат
       - Капсулира алгоритъм
   * 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
   * Намаляване на зависимостта с OOP техники
     + Абстракция
     + Дефинират се public интерфейс и скрийте детайлите по реализацията
     + Капсулиране
     + Правят се методите и полетата private освен ако със сигурност не е необходимо да са други
     + Дефинират се нови членове като private
     + Увеличете видимостта им веднага щом ви потрябват
   * Недопустима зависимост
     + Метод в клас, зависим от static полета във външен клас
       - Ползват се 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
   * Брой параметри
     + 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
   * Колко дълъг трябва да е един метод?
     + Няма конкретно ограничение
     + Избягват се 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

5**. Какво е Обектно ориентирано програмиране (ООП)**

- ООП е наследник на процедурното програмиране и представлява пренасяне на обекти от реалния живот в света на програмирането. Поведението и характеристиките на Обектът от реалния свят се чрез класове, методи, пропъртита и свойства. ООП съдържа 4 много важни концепции, а именно:

**- Наследяване**

Предоставянето на свойствата и методите на един клас върху друг се нарича наследяване.

!!Един клас не може да наследява множество класове!!

* + - 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
    - Използват се наследяването за:
      * Многократна употреба на повтарящ се код: данни и логика
      * Опростяване поддръжката на кода

**- Полиморфизъм(„много форми“)**

Това е способността на обект да заема много форми

Пример: Човекът е бозайник, Делфинът е бозайник следователно бозайници имат 2 форми – Човек и Делфин (Mammal person = new Person; Mammal dolphin = new Dolphin)

-Полиморфизъм е напълно зависим от наследяването, тъй като той се създава само и единствено когато имаме наследяване.

- При полиморфизма променливите се запазват в референтен тип(лявата част)

- При полиморфизма дясната му част е обектен тип.

- При полиморфизма имаме достъп само на методите от референтния тип

Пример**: Mammal** person = new Person(); -Mammal е референтен тип и имаме достъп само до неговите методи.

-Ако искаме методите на обектния тип, то трябва да кастнем лявата страна към обектния тип.

Пример: **(Person)(Mammal)** person = new Person();

-Има два вида полиморфизъм – статичен(по време на компилиране) и динамичен(по време на изпълнение)

* + - 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

**- Абстракция**

-Абстракцията е една от най-важните концепции в ООП

-Абстракцията представлява способността на един клас да предостави всичко нужно на потребителя без да му представя вътрешната си логика.

Пример: Човешкото ни тяло – то ни е предоставено и го използваме всеки ден, но не знаем вътре как работи.

Пример 2: Използваме принтер всеки ден, но не знаем вътре как се принтират листовете.

Пример 3: Използваме клавиатура всеки ден, но не знаем вътрешната мембрана как е програмирана.

-Чрез абстракция ние разпределяме вътрешната логика в един клас по такъв начин, че тя да не е достъпна за потребителя. Най- често това може да се случи с private аccess модифайъри, пропъртита и вътрешни методи.

**-Енкапсулация**

**-**Основен принцип на ООП

- Използва се за да се скрие информацията на даден клас по такъв начин, че тя да важи само и единствено за него.

- Използва се за Single-responsibility дизайн патърн – един клас трябва да извършва една работа и има едно значение

- Енкапсулацията се създава чрез private access modifiers, като то важи както и за методи, така и за полета.

**7. SOLID принципи(дизайн принципи) в ООП**

S – Single-Responsibility Principle- Един клас трябва да носи само една отговорност

O – Open-Closed Principle – Един клас трябва да е отворен за разширение(да има възможност за промяна; да е еластичен), но затворен за модификация.

L - Liskov Substitution Principle – Обектите в дадена програма трябва да могат да бъдат заменими с инстанции от техните наследници, без да променят правилността на тази програма

I - Interface Segregation Principle – Гласи, че много на брой интерфейси е по-добре, отколкото един огромен такъв.

D - Dependency Inversion Principle – постига се чрез абстракция. Гласи че класове на по високо ниво в йерархията не трябва да са зависими от промените на класове по ниско в йерархията.

**8. Разлики между абстрактен клас и интерфейс.**

**-** Интерфейса е шаблон с методи на даден клас

- Интерфейса казва какво трябва да има даден клас, а не как го прави

-Един клас може да разширява и имплеменира няколко интерфейса, но не може да наследява няколко класа.

- Интерфейса винаги е с публични методи и трябва да е напълно имплементиран

- Интерфейса не може да съдържа констуктор в себе си.

- Абстрактния клас може да има както и методи така и вътрешна логика.

-Методите на абстрактния клас могат да бъдат abstract и virtual:

\*abstract методи – те нямат тяло и логиката им се развива в наследения клас като метода се overridва

\*virtual методи – те имат тяло и логика, но в наследения клас могат да бъдат overridвани и да им се приложи напълно различна логика от тази на абстрактния.

-Абстрактния клас може да има конструктор, както и полета, пропъртита, променливи и методи

**9.Качествени класове**

* + Специализация
    - 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
  + Зависимост
    - 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
  + Наследяване или включване
    - 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**
    - Не се скриват методи в подклас
      * Пример: ако класа **Timer** има public метод **Start()**, не дефинирайте private **Start()** в **AtomTimer**
    - Местят се 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 нужнмо още едно ниво на наследяване?
    - Бъдете подозрителни към класове, които презаписват процедура и не правят нищо в нея
      * Дали коректно е ползвана тази процедура?
    - Избягва се прекалено многократното наследяване
      * Не създавайте повече от 6 нива на наследяване
    - Избягва се ползването на protected полетата за данни в наследения клас
      * По-добре добавете наследен protected метод / свойства
  + Конструктори
    - Инициализират се 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
    - Инициализират се всички членове-данни в същия ред, в който са декларирани
    - Предпочита се deep copies пред shallow copies (**ICloneable** ще направи deep copy)
  + Причина за създаване на 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
    - 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

1. Лош код
   * Разводняване
     + Дълги методи
       - По-добре е методите да са по-къси (по-лесно именуване, по-разбираеми са, по-малко повторения на код)
     + Големи класове
       - Твърде много променливи в екземплярите
       - Нарушават принципа „Една-едничка цел" (Single Responsibility)
     + Мания за прости данни (за прекаленото им използване)
       - Прекаляване с прости типове, вместо добра абстракция
       - Може да бъдат обособени в свой клас с вградено валидиране
     + Дълъг списък с параметри (**in** / **out** / **ref**)
       - Може да подсказва процедурно вместо ОО програмиране
       - Може би методът върши твърде много неща
     + Групички от данни
       - Набор от данни, винаги използвани заедно без да са групирани
       - Например полетата на кредитна карта в **Order** класа
     + Избухване в комбинации
       - Напр. **ListCars()**, **ListByRegion()**, **ListByManufacturer()**, **ListByManufacturerAndRegion()** и т.н.
       - Решение може да е шаблона Interpreter (LINQ)
     + Чудати решения
       - Странни решения на често срещани проблеми
       - Непоследователност
       - Решение: Заменете алгоритъма или изПолзват се Adapter
     + Клас, който нищо не прави
       - Решение: Слейте го с друг клас или го премахнете
     + Задължителен начален / завършващ програмен код
       - Изисква винаги няколко реда код, преди да се ползва
       - Решение: параметър-обект, factory метод, **IDisposable**
   * Неясноти
     + Обхват
       - Целта на кода е неясна и се нуждае от коментар (лошо)
       - Програмният код е твърде дълъг, за да е обозрим (лошо)
       - Решение: частичен клас, нов клас, подреждане на кода
     + Коментари
       - Трябва да обясняват **ЗАЩО**, а не **КАКВО** или **КАК**
       - Добрите коментари: дават повече информация, препратки към ресурси, обясняват алгоритъм, причини или контекст
     + Къси / неподходящи имена
       - Трябва да са подходящи, говорвящи и еднотипно използвани
     + Вертикално отделяне
       - Трябва да декларирате променливите точно преди първата им употреба, за да се избегне скролирането
       - Или изПолзват се малки функции
     + Непоследователност
       - Следвайте POLA (Принципа на най-малкото чудене)
       - Непоследователността е объркваща и разсейваща
     + Неясни намерения
       - Кодът трябва да е толкова обяснителен, колкото е възможно
   * Блокаж на промени
     + Многократна промяна
       - Клас, който често се променя по различни начини / причини
       - Нарушава SRP принципа (single responsibility principle)
       - Решение: да се отдели класа
     + Принудителни промени
       - Една промяна налага промени в много класове
         * Трудно е да бъдат намерени, лесно е да се пропусне някой
       - Решение: местене на методи и полета, подреждане на кода
     + Сложност на условията
       - Увеличава общата сложност (Cyclomatic complexity, броят на различните пътища, по които може да бъде изпълнен кода)
       - Симптоми: много влагания (arrow code) и бъгливи **if**-ове
       - Решение: отделяне на метод, шаблон „Strategy“, „State“ или „Decorator“
     + Зле написани тестове
       - Лошо написаните тестове може да възпрепятстват промените
       - Много силни зависимости
   * Излишества
     + Повторения на код
       - Нарушава DRY принципа
       - Резултат е от copy-pasted код
       - Решение: отделяне на метод, клас, pull-up метод, шаблон  
         Template Method
     + Ненужен код (такъв, който никога не се ползва)
       - Обикновено се открива от инструментите за статистически анализ
     + Спекулативни обобщения
       - „Някой ден може да ни потрябва …"
     + Мързеливи класове
       - Класове, които не правят достатъчно, за да оправдаят своето съществуване трябва да бъдат премахнати
       - Всеки клас иска време и усилие да бъда разбран и поддържан
     + Класове с данни
       - Класове само с полета и свойства
       - Липсващо валидиране? Програмният код е в други класове?
       - Решение: да се премести свързаната с данните логика в класа
   * Сглобки
     + Завист за чужди екстри (Feature envy)
       - Метод, който изглежда по-заинтересуван от друг клас, различен от този, в който е метода
       - Дръжте заедно нещата, които взаимно се променят
     + Неуместно интимничене (Inappropriate intimacy)
       - Класове, които знаят твърде много един за друг
       - Проблеми: при наследяване, двупосочна връзка
       - те на методи / полета, извличане на клас, промяна на връзката в еднопосочна, делегиране вместо наследяване
     + Законът на Деметра (Law of Demeter, LoD)
       - Даден обект трябва да очаква възможно най-малко определена структура на свойства или каквото и да е
       - Лош пример: **customer.Wallet.RemoveMoney()**
     + Индиректно разкриване (Indecent exposure)
       - Някои класове или членове са public без да е нужно
       - Нарушава капсулирането
       - Може да доведе до неуместно интимничене
     + Верижни съобщения
       - **Something.Another.SomeOther.Other.YetAnother**
       - Тясна зависимост между клиент  
         и структурата за достъп
     + Посредник
       - Делегиране, отишло твърде далеч
       - Понякога може да го премахнем или вградим
     + Скитащи данни
       - Да се предават данни само защото са нужни на някой друг
       - Решение: Премахваме посредническите данни, извличаме клас
     + Изкуствени зависимост (Artificial coupling)
       - Неща, които не зависят едно от друго, няма нужда изкуствено да се обвързват в зависимост
     + Скрита временна зависимост (Hidden temporal coupling)
       - Не трябва да предполагаме реда на операциите
       - Например класът pizza не трябва да знае стъпките за правене на пица -> Template Method шаблон
     + Скрити зависимости (Hidden dependencies)
       - Класовете трябва да обявяват своите зависимости още в конструктора си
       - **new** е връзката / принципа на Dependency Inversion