

12. Тест-резюме

Кохезията (Cohesion) е мярка за това колко добре са свързани и работят заедно вътрешните части на един модул в софтуерната архитектура. Тя показва доколко елементите в един модул споделят обща цел или функционалност.

Висока кохезия = Всички части на модула са силно свързани и работят за постигане на една основна цел.

Ниска кохезия = Частите на модула вършат различни, несвързани задачи. ✗ Пример: Модул, който съдържа ф-и за обработка на данни, изпращане на имейл и валидиране ...

▶ От най-лоша към най-добра кохезия:

Случайна кохезия (най-нежелана) - Частите нямат нищо общо. Просто са сложени заедно без връзка. ✗ Пример: Функция, която едновременно изтрива файл и изпраща имейл.

Логическа кохезия - Частите вършат различни задачи в зависимост от условие (например параметър). ✗ Пример: Функция, която прави различни неща в зависимост дали входът е число или текст.

Темпорална кохезия - Частите се изпълняват по едно и също време, но нямат логическа връзка. ✗ Пример: Всички задачи, които се изпълняват при стартиране на програмата.

Процедурна кохезия - Частите се изпълняват в определен ред, но не винаги са силно свързани. ! Пример: Функция, която първо проверява данни, после ги записва.

Комуникационна кохезия- Всички части работят с едни и същи данни. Пример: Модул, който обработва информация за клиент – чете, проверява и записва.

Последователна кохезия- Резултатът от една част се използва в следващата. Пример: Вземане на данни → обработване → записване.

Функционална кохезия (най-желана) - Всички части работят за една конкретна цел. Пример: Функция, която изчислява ДДС. Това е най-чистият и лесен за поддръжка код.

- **Случайна кохезия** = хаос
- **Процедурна/Темпорална кохезия** = частична връзка
- **Функционална кохезия** = идеален, чист код

Icons for Each Type of Cohesion:

1. Случайна кохезия (Coincidental Cohesion)

Icon: (зар) или Meaning: Случайни елементи без връзка, както случайно хвърлени зарове.

2. Логическа кохезия (Logical Cohesion)

Icon: (стрелки в различни посоки) Meaning: Различни логически операции, насочени според условие.

3. Темпорална кохезия (Temporal Cohesion)

Icon: (часовник) Meaning: Задачи, които се изпълняват по едно и също време.

4. Процедурна кохезия (Procedural Cohesion)

Icon: → (стрелка напред) **Meaning:** Последователни стъпки в процедура.

5. Комуникационна кохезия (Communicational)

Icon:  (диаграма) или  (комуникация) **Meaning:** Всички операции работят с едни и същи данни.

6. Последователна кохезия (Sequential Cohesion)

Icon:  (верига) **Meaning:** Резултатът от една стъпка се използва в следващата.

7. Функционална кохезия (Functional Cohesion)

Icon:  (мишена) **Meaning:** Всички части са насочени към една конкретна цел.

 Функционална  Последователна  Комуникационна →
Процедурна  Темпорална  Логическа  Случайна

 В UML свойството **Direction** се използва, за да определи **посоката на данните** за параметър в операция на клас. Това показва как данните се предават – дали влизат в операцията, излизат от нея или и двете: **in** , **out** , **inout**, **return**

Когато описваме **сценарий на случай на употреба (use case scenario)** в UML или софтуерно моделиране, е **задължително да зададем входни и изходни условия**.

Входни условия (Entry Conditions)  - Това са условията, които трябва да са изпълнени, за да може сценарият да започне.

Изходни условия (Exit Conditions) ✅ - Това са условията, които показват **кога и как завършва успешно** случаите на употреба.

UML изглед на потребителските случаи (Use Case View) показва **как потребителите взаимодействат със системата**.

Този изглед е полезен за анализ на функционалните изисквания и за планиране на архитектурата на софтуера.

Какво съдържа Use Case View?

-**Диаграми на потребителските случаи (Use Case Diagrams)**

👉 Показват какви функции (сценарии) предлага системата и кои актьори (потребители или други системи) ги използват.

Пример: Диаграма, която показва как потребител може да „Влезе в акаунт“, „Поръча продукт“ и др.

-**Диаграми на сътрудничеството (Collaboration Diagrams)** 🤝

Показват **как обектите в системата си сътрудничат**, за да изпълнят даден сценарий. Фокусирани са върху взаимодействията между обекти и как те си изпращат съобщ.

-**Диаграми на класовете (Class Diagrams)** 💡 - Представят **структурата на системата** чрез класове, техните атрибути, методи и връзки помежду им. Те показват **какви данни** обработва системата и как е организирана.

▶ **Какво НЕ включва?** Диаграми на последователност (Sequence Diagrams) и Диаграми на състоянието (State Diagrams) се използват в други изгледи (например за динамично поведение на системата), но **не са част от основния Use Case View**.

Use Case View включва:

- ✓ **Диаграми на потребителските случаи** – какво прави системата;
- ✓ **Диаграми на сътрудничеството** – как обектите работят заедно;
- ✓ **Диаграми на класовете** – как е структурирана с-мата.

Да се види ясно какво трябва да прави с-мата и как работи „отвътре“.



Обяснение на **типовете свързаност**:

Data Coupling (Свързаност чрез данни) – Най-желаната.
Модулите си обменят само нужните данни (параметри). Това прави кода лесен за промяна и тестване.

```
function calculateArea(width, height){  
    return width * height;  
}  
  
console.log(calculateArea(5, 10));
```

Internal Data Coupling (Свързаност чрез вътрешни данни) – Все още добра, но **модулите имат достъп до вътрешните структури на данните на друг модул**, което намалява независимостта им.

```
const rectangle = { width: 5, height: 10 };  
  
function calculateArea(rect) {  
    return rect.width * rect.height; }  
console.log(calculateArea(rectangle));
```

Вместо да подадем конкр. стойности, **подаваме обект**. Ако структурата на обекта се промени, ще трябва да променим и функцията. Това прави модулите малко по-зависими.

Control Coupling (Свързаност чрез контрол) – По-нежелана.

Един модул контролира поведението на друг, изпращайки му контролни флагове или логика.

```
function processData(data, shouldLog) {  
    if (shouldLog) {  
        console.log("Processing data:", data);  
    }  
    return data * 2;  
}  
  
processData(5, true);
```

💡 Използваме флаг `shouldLog`, който контролира как функцията работи. Това означава, че и външният код определя логиката вътре във функцията.

Global Data Coupling (чрез глобални данни) – Най-нежеланата.

Модулите споделят глобални променливи, което прави с-мата трудна за поддръжка и увеличава риска от грешки.

```
let globalValue = 10;  
  
function doubleValue() {  
    return globalValue * 2;  
}  
  
console.log(doubleValue());
```

SOLID принципите са 5 основни правила за създаване на добър и лесно поддържащ се обектно-ориентиран софтуер. Те помагат кодът да бъде по-гъвкав, четим и лесен за разширение.

 **S – Single Responsibility Principle (Принцип на единствената отговорност)** - Всеки клас трябва да има **само една отговорност** и да прави **само едно нещо добре**.

 **Лошо:**

```
class User {  
  
    saveToDatabase() /* записва потребител в база данни */  
  
    sendEmail() /* изпраща имейл на потребителя */  
  
}
```

 **Добро:**

```
class User {  
  
    // Представлява потребител  
  
}  
  
class UserRepository {  
  
    saveToDatabase(user) /* само за запис в база данни */  
  
}  
  
class EmailService {  
  
    sendEmail(user) /* само за изпращане на имейли */  
  
}
```

🎯 Лесно можеш да променяш и тестваш всеки клас, без да засягаш другите.

🟠 **O – Open/Closed Principle (Принцип на отвореност/затвореност)**-Кодът трябва да е **отворен за разширение, но затворен за промени**. Това значи, че можеш да добавяш нова функционалност, без да променяш съществуващия код.

✗ **Лошо:**

```
function calculateDiscount(type, price) {  
    if (type === "student") return price * 0.8;  
    if (type === "vip") return price * 0.9;  
    return price;  
}
```

✓ **Добро:**

```
class Discount {  
    calculate(price) {  
        return price;  
    }  
}  
  
class StudentDiscount extends Discount {  
    calculate(price) {  
        return price * 0.8;  
    }  
}
```

```
    }  
}  
  
class VIPDiscount extends Discount {  
  
    calculate(price) {  
  
        return price * 0.9;  
  
    }  
  
}
```

🎯 Когато трябва да добавиш нов тип отстъпка, просто създаваш нов клас, без да променяш стария код.

✳️ **L – Liskov Substitution Principle (Принцип на Лисков за заместване)**- Подкласовете трябва да могат да се използват **на мястото на родителските класове**, без това да води до грешки.

Пример:

✗ **Лошо:**

```
class Bird {  
  
    fly() {  
  
        console.log("Flying");  
  
    }  
  
}  
  
class Penguin extends Bird {  
  
    fly() {
```

```
    throw new Error("Penguins can't fly!");
}

}
```

 Добро:

```
class Bird {

    move() {

        console.log("Moving");

    }

}

class Sparrow extends Bird {

    move() {

        console.log("Flying");

    }

}

class Penguin extends Bird {

    move() {

        console.log("Swimming");

    }

}
```

 Класовете се държат логично, без да нарушават поведението на програмата.

 **I – Interface Segregation Principle**(разделяне на интерфейса)- Не принуждавай класовете да имплементират методи, които не използват. По-добре е да има **малки, специфични интерфейси**, вместо един голям.

 **Лошо:**

```
class Worker {  
    work() {}  
    eat() {}  
}
```

```
class Robot extends Worker {  
    eat() {  
        /* Роботът не яде, но трява да имплементира този метод */  
    }  
}
```

 **Добро:**

```
class Workable {  
    work() {}  
}  
  
class Eatable {  
    eat() {}  
}
```

```
class Human extends Workable, Eatable {  
    work() {  
        console.log("Working");  
    }  
    eat() {  
        console.log("Eating");  
    }  
}  
  
class Robot extends Workable {  
    work() { console.log("Working like a robot"); }  
}
```

⌚ Всеки клас използва само методите, от които има нужда.

⚠ D – Dependency Inversion Principle (Принцип на обръщане на зависимостите)-Класовете трябва да зависят от **абстракции** (интерфейси), а не от конкретни реализации. Така променяш лесно логиката без да засягаш другите части на системата.

✗ Лошо:

```
class MySQLDatabase {  
    connect() {  
        console.log("Connected to MySQL");  
    }  
}
```

```
}
```

```
class UserService {
```

```
    constructor() {
```

```
        this.db = new MySQLDatabase();
```

```
    }
```

```
}
```

 Добро:

```
class Database {
```

```
    connect() {}
```

```
}
```

```
class MySQLDatabase extends Database {
```

```
    connect() {
```

```
        console.log("Connected to MySQL");
```

```
    }
```

```
}
```

```
class UserService {
```

```
    constructor(database) {
```

```
        this.db = database;
```

```
    }
```

```
}
```

🎯 Можеш лесно да смениш базата данни, без да променяш UserService.

- **S:** Един клас = една отговорност.
- **O:** Лесно разширение без промяна на съществуващия код.
- **L:** Подкласовете работят без проблеми вместо родителските класове.
- **I:** Малки интерфейси, специфични за нуждите на класа.
- **D:** Зависимост от абстракции, не от конкретни реализации

Data Coupling → Internal Data Coupling → Control Coupling → Global Data Coupling 🚀

Когато има само „**слаба връзка**“ (**loose coupling**) между класовете, това означава, че те са слабо зависими един от друг. **ClassA** има метод, който използва **ClassB** като **параметър**, но **ClassA** не държи постоянна референция (инстанция) към **ClassB**.

📊 Как се показва това в диаграма на класовете (UML)?

👉 **Зависимост (Dependency)** слаба връзка

- **Пунктирна стрелка** (---►) от **ClassA** към **ClassB**
- Показва, че **ClassA** зависи от **ClassB** само временно (например като параметър в метод). Няма силна връзка или асоциация между тях.

```
class ClassB {
```

```
    void doSomething() {}
```

```
}
```

```
class ClassA {  
    void performAction(ClassB b) { // ClassB е само параметър  
        b.doSomething();  
    }  
}
```

Тук **ClassA** използва **ClassB**, но не съхранява референция към него. Това е **слаба връзка**, която се показва с **пунктирна стрелка (зависимост)** в UML диаграмата.

"Един клас може ли да се използва като върнат тип или като тип на параметър в операция на съставния клас?" - **Може да се ползва като върнат тип или като тип на параметър на операция на съставния клас.**" В обектно-ориентираното програмиране, когато имаш клас (например **ClassA**), той може:

- 1. Да се върне като резултат** от метод (върнат тип).
- 2. Да се използва като параметър** в метод.

 **Ключова идея:** Няма ограничения в ООП, които да забраняват използването на клас едновременно като **върнат тип** и като **тип на параметър**

 Обяснение за състоянието "H" в диаграма на състояния (State Diagram): В UML диаграмите на състояния често се използват специални символи, наречени **псевдо-състояния**. Те

помагат да се моделира поведението на системата, като показват как тя преминава от едно състояние в друго.

Буквата "H" обикновено означава "History" (история) и се използва за проследяване на предишното състояние на системата. Има два основни вида "H" състояния:

Псевдо-състояние на плитка история (Shallow History) — обозначава се с „H“ - **Запомня само последното активно под-състояние в контейнер** (но не и състоянията в по-дълбоки нива).

Пример: Ако миялната машина е била в режим "Изплакване" и се изключи, при включване ще започне от "Изплакване".

Псевдо-състояние на дълбока история (Deep History) — обозначава се с „H“* - **Запомня не само последното активно състояние, но и всички вложени под-състояния**. Пример: Ако машината е била в "Интензивно пране" → "Изплакване" → "Сушене", ще се врне точно в "Сушене", дори ако това е на по-дълбоко ниво.

"H" = **Псевдо-състояние на плитка история (Shallow History Pseudostate)** ✓ "H"*= **Псевдо-състояние на дълбока история (Deep History Pseudostate)**

В OCL (Object Constraint Language) няма дефинирана пряка характеристика, която да задава максимално време за изпълнение на даден израз. OCL се използва за формализиране на условия и ограничения в UML модели и не е инструмент, който да задава или измерва време за изпълнение.

OCL обикновено се фокусира върху логическо проверяване на условия или изрази за модели, без да се обръща внимание на

производителността или времевите ограничения на тяхното изпълнение.

Заместването на **наследяване на имплементация** (където един клас наследява методите на друг клас) с **делегиране** (където един клас поиска от друг клас да извърши дадена задача).

Нарастването на дълбочината на дървото на наследяване:

Когато класовете наследяват от много нива в йерархията, тя може да стане твърде дълбока и сложна, което е трудно за управление. Делегирането може да помогне за намаляване на тази сложност. **Множественото наследяване:** Това се случва, когато клас наследява от повече от един клас. Делегирането не решава директно този проблем, но избягва проблемите, които могат да възникнат от множественото наследяване.

Използването на делегиране може да помогне да се избегне създаването на дълбока и сложна йерархия на наследяване, която става трудна за поддръжка

```
// Интерфейс за говорене
```

```
interface Speaker {  
    void speak();  
}
```

```
// Клас Котка, реализиращ интерфейса Speaker
```

```
class Cat implements Speaker {  
    @Override
```

```
public void speak() {  
    System.out.println("Meow");  
}  
  
}  
  
  
// Клас Куче, реализиращ интерфейса Speaker  
class Dog implements Speaker {  
  
    @Override  
    public void speak() {  
        System.out.println("Woof");  
    }  
  
}  
  
  
// Клас Животно, което делегира на Speaker  
class Animal {  
  
    private Speaker speaker;  
  
  
  
    public Animal(Speaker speaker) {  
        this.speaker = speaker;  
    }  
}
```

```
public void speak() {  
    speaker.speak();  
}  
  
}  
  
// Основен клас  
  
public class Main {  
    public static void main(String[] args) {  
        Speaker catSpeaker = new Cat();  
        Speaker dogSpeaker = new Dog();  
  
        Animal animal1 = new Animal(catSpeaker);  
        Animal animal2 = new Animal(dogSpeaker);  
  
        animal1.speak(); // Изход: Meow  
        animal2.speak(); // Изход: Woof  
    }  
}
```

В този пример класът Animal не наследява методите speak() директно, а ги делегира на обекти от класовете Cat и Dog, които имплементират интерфейса Speaker. Това е пример за

делегиране, където класът Animal разчита на външни обекти да извършват работата вместо него.

Три основни стереотипа на класове:

1. «**boundary**» / «граница» - представляват интерфейса между системата и актьорите/външния свят
2. «**control**» / «управление» - представляват бизнес логиката и координацията между другите класове
3. «**entity**» / «същност» - представляват информацията и данните в системата

Този отговор е базиран на стандартната UML нотация, където тези три стереотипа са основните типове класове при анализ на системата. Те помагат за разделянето на отговорностите в системата според шаблона Model-View-Controller (MVC).

Видимости в UML: Публична видимост (+): Това означава, че елементът е достъпен за всички други класове, без значение дали са в същия пакет или не. Обикновено се използва за публични методи и атрибути, които трябва да бъдат достъпни от всяко място в програмата.

Заштита видимост (#): Защитената видимост означава, че елементът е достъпен само за класове, които наследяват текущия клас (т.е. за подкласовете) и за класовете в същия пакет. Често се използва за атрибути и методи, които трябва да бъдат достъпни само от подкласовете или в рамките на пакета.

```
class MyClass {
```

```
#protectedMethod(); // Защитен метод, достъпен за  
наследници  
}
```

Пакетна видимост (~): Пакетната видимост означава, че елементът е достъпен само за класове, които принадлежат към същия пакет. Тази видимост е полезна, когато искаме да ограничим достъпа до класове, но все пак позволяваме взаимодействие между класовете в същия пакет.

```
class MyClass {  
  
    ~packageVariable; // Пакетна видимост, достъпно само в  
    същия пакет  
  
}
```

Частна видимост (-): Частната видимост означава, че елементът е достъпен само в рамките на самия клас и не може да бъде достъпен извън него. Това е най-строгата видимост и се използва за защита на данни или методи, които трябва да бъдат скрити от външни класове.

```
class MyClass {  
  
    -privateVariable; // Частна променлива, достъпна само в  
    MyClass  
  
}
```

Публична + Достъпно за всички класове и обекти

Защитена # Достъпно за наследници и класове в същия пакет

Пакетна ~ Достъпно само за класове в същия пакет

Частна - Достъпно само в рамките на самия клас

Тези видимости ви позволяват да контролирате достъпа до вътрешните компоненти на вашите класове и да поддържате добра инкапсуляция в системата.

Диаграмите на профили в UML включват: "**Стереотипи (stereotypes), дифиниции на маркирани стойности (tagged value definitions) и ограничения (constraints)**" —

Стереотипи — те са разширения на основните елементи в UML и могат да се използват за добавяне на нови типове елементи, които са специфични за даден домейн.

Маркирани стойности (Tagged values) — те добавят допълнителна информация към елементите, като например метаданни, които описват конкретни атрибути.

Ограничения (Constraints)- условия / правила, които трябва да са изпълнени, за да се гарантира, че моделът е валиден.

Разликата между фрагментите от тип **seq** и **strict** е в диаграмите на последователността **strict** е по-строг и ограничава изпълнението на събитията в диаграмата да бъде **напълно последователно**, докато **seq** позволява по-гъвкаво изпълнение. **strict** може да бъде редуциран до **seq**, но не и обратно.

"frozen" — обектът не може да бъде модифициран от операцията на неговия клас. Това гарантира, че състоянието на обекта не се променя, когато се извършва дадена операция върху него.

"restricted" — може да се използва за ограничаване на достъпа или действия върху обектите, но не се отнася директно до забрана за модифициране от операция.

"query" — обозначава операция, която не променя състоянието на обекта, а просто извлича информация от него. Това е операция за четене, а не за модифициране.

"safe" — се отнася до операции, които не водят до неочекани резултати или нарушения на условията на системата, но не забранява промените на обекта.

Случай на употреба:

System boundary box (Граници на системата) — Това е част от описанието на случай на употреба. То определя какво влиза в обхвата на системата и какво остава извън нея. Границата показва взаимодействието между системата и външния свят.

Входни и изходни условия — Това също е част от описанието на случай на употреба. Входните условия определят какво трябва да бъде предоставено на системата, за да започне случая на употреба, докато изходните условия описват резултатите от успешното изпълнение на случая.

Име на случая — Това е основна част от описанието на случай на употреба. Името трябва ясно да описва какво прави системата или какво изпълнява дадената дейност.

Поток от събития — Това е основна част от описанието на случай на употреба и обяснява как се движат събитията през системата, включително стъпките на изпълнение и взаимодействията между акторите и системата.

Участващи актьори — Това също е част от описанието на случай на употреба. Акторите могат да бъдат потребители, системи или други външни обекти, които взаимодействват с разглежданата система.

Качествени изисквания: Тези изисквания се отнасят до характеристиките на системата (като производителност, сигурност, надеждност и др.), но не са част от самото описание на случай на употреба.