1. **Szoftverfejlesztés**

**Fogalmak:**

* **Alkalmazásprogramozási interfész (API)**egy külső rendszer vagy szolgáltatás funkcionalitáshoz/adataihoz illeszkedő szoftver létrehozására szolgáló függvények, protokollok, paraméterek és objektumok.  
  Egy API specifikáció definiálja, leírja és/vagy implementálja az API-t. Különböző szoftverek implementálhatják ugyanazt az API-t.

Pl.: DOM, JDBC, ODBC, OpenGL, POSIX, WebGL, WebSocket

* **Programkönyvtárak**

szoftverek együttese, amely arra lett tervezve, hogy a szoftverfejlesztést, használatot és karbantartást segítse.  
Egy könyvtár erőforrásokat (adat, kód) tartalmaz, aminek van egy jól meghatározott interfésze (API)  
Pl.: Gson, Apache Log4j 2, Apache Commons libraries, jQuerry

* **Keretrendszerek**

Egy félkész alkalmazás, mely arra lett tervezve, hogy újrafelhasználható szoftvertermékeket (osztály, objektum, komponens) biztosítson hasonló alkalmazásokhoz. Egy keretrendszer megvalósításához öröklődő osztályokat kell létrehozni, amelyek megvalósítsák a keretrendszer absztrakt osztályait.

Fajtái:

* + Alkalmazásfejlesztési keretrendszerek (.NET, Micronaut, Mono, Spring Framework)
  + Webalkalmazás/webfejlesztési keretrendszerek (Angular, Vue.js, Django, Express, Meteor, Rails)
  + Keresztplatformos alkalmazásfejlesztési keretrendszerek (JavaFX, Qt)
  + Mobil alkalmazásfejlesztési keretrendszerek (Flutter, React Native)
  + Perzisztencia keretrendszerek (EclipseLink, Entity Framework, Hibernate ORM)
  + Teszt automatizálási keretrendszerek (JUnit, unittest, Selenium, Jest, Google Test)
  + Gépi tanulási keretrendszerek (PyTorch, Keras, Deeplearning4j)
* **Platformok**

Olyan számítógép és hozzá tartozó oprendszer/virtuális környezet, ahol szoftverek telepíthetőek és futtathatóak

Pl.: .NET, Android, Arduino, AWS, CUDA, Google Cloud, Mono, Node.js, Oracle Database, TensorFlow, Web platform

Néha ugyanazt a szót használják egy platformra és egy keretrendszerre is. (.NET)

* **Szoftverfejlesztő készlet (Software Development Kit – SDK)**

Eszközök olyan gyűjteménye, mely alkalmazások egy platformra történő fejlesztéséhez szükséges. Tartalmaz könyvtárakat, keretrendszereket, doksit, mintakódokat.

Pl.: AWS SDK, Cloud SDK, Dart SDK, Facebook SDK, JDK

* **Integrált fejlesztői környezet (Integrated Development Environment – IDE)**

Egységes keretrendszerben és felhasználói felületen keresztül ad szoftvereszközöket a szoftverfejlesztéshez. Egy bizonyos programozási nyelven történő fejlesztés támogatnak.

Kódszerkesztő: Notepad++, Sublime Text, VS Code  
IDE: Apache NetBeans, Eclipse, IntelliJ IDEA, VS

1. **Szoftver sebezhetőségek**

**Sebezhetőség**: tökéletlenségből származó hiba egy szoftverben/firmware-ben/ hardverben/komponensben, mely kiaknázható a titkosságra, integritásra, rendelkezésre állásra negatív módon.

**Nulladik napi sebezhetőség:** olyan sebezhetőség, melyről nem tudnak a gyártók. Ennek kihasználása a **nulladik napi támadás.** Pl. Log4Shell

**Sebezhetőségi adatbázis:** talált sebezhetőségekre vonatkozó információkról szóló adatbázis.   
**Szabad**: Github Advisory Database, National Vulnerability Database, OSV, …  
**Nem** **szabad**: Debricked VD, Mend.io VD, Snyk VD, VulDB, …

**Common Vulnerabilities and Exposures (CVE):**

* Program a nyilvánosságra hozott kiberbiztonsági sebezhetőségek azonosítására, meghatározására, kategorizálására.
* Egy sebezhetőséghez egy CVE rekord tartozik
* Szabad nyilvános használatra
* Nem egy sebezhetőségi adatbázis, inkább szótár
* Fogalmak:
  + **CVE azonosító (ID)**: alfanumerikus azonosító, egy sebezhetőségre hivatkozik
  + **CVE rekord**: sebezhetőséget leíró adatok ember és gép által olvasható formátumban (HTML, JSON)
  + **CVE lista**: bejelentett CVE rekordok katalógusa

**National Vulnerability Database (NVD):**

* CVE listára épülő és azzal szinkronizált sebezhetőségi adatbázis.
* Kiegészíti a CVE rekordokat, pl. hibajavítási információ, súlyossági pontszám (0-10)
* Fejlett kereséssel rendelkezik, pl. oprendszer szerint, sebezhetőség típus alapján
* Az adatok egy web API-n elérhetők

1. **Szoftver verziózás**

**Fogalmak:**

* **Verziózás:** egyedi verziónevek/verziószámok hozzárendelése szoftver konfiguráció elemek állapotaihoz egy bizonyos célra (szoftver kiadása csoport számára, termékvonal azonosítása).
* **Verzió:** egy szoftver konfigurációs elem egyedi és megkülönböztethető állapota
* **Kiadás:** egy szoftver olyan verziója, melyet elérhetővé tesznek az ügyfeleknek. Fajtái:
  + **Fő kiadások:** jelentős új információkat hoznak.
  + **Minor kiadások:** hibajavításokat, kisebb továbbfejlesztéseket hoznak.
* **Changelog:** állomány, mely egy időrendbe rendezett listát tartalmaz a projekt egyes verzióiban említésre méltó változásairól. Pl.: git-cliff

**Szemantikus verziószámozás:** egyszerű szabályok és követelmények a verziószámok kiosztásához és növeléséhez. Olyan szoftverekhez használható, mely nyilvános API-val rendelkezik.

* **normál verziószámok:** X.Y.Z formájú, ahol X – főverzió, Y – alverzió, Z – patch verzió
* **verziószámok növelése:**
  + **főverziót** kell növelni, ha a korábbi verzióval inkompatibilis módon változik az API.
  + **alverziót** kell növelni, ha a korábbi verzióval kompatibilis módon új funkcionalitást vezetünk be.
  + **patch verziót** kell növelni, amikor a korábbi verzióval kompatibilis hibajavítások történnek.
* **breaking change:** nem visszafelé kompatibilis változás egy nyilvános API-ban (fordításidejű, szerkesztésidejű, futásidejű hiba)
* **precedencia meghatározása:**
  + balról jobbra kell haladni, numerikusan összehasonlítva a verziószámokat

Pl.: 1.0.0 < 1.0.1 < 1.2.0 < 1.10.0 < 2.0.0

* + ha egyeznek a verziók, akkor a kiadás előtti verzió kisebb precedenciával bír

Pl.: 1.0.0-alpha < 1.0.0

* + ha betűket vagy kötőjeleket tartalmazó azonosítók abc-sorrend szerint kell összehasonlítani.
  + a numerikus azonosító precedenciája **mindig alacsonyabb** a nem numerikus azonosítóhoz képest
  + Pl.: 1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-alpha.beta < 1.0.0-beta < 1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0-rc.1 < 1.0.0

1. **Annotációk a Java programozási nyelvben**

**Annotáció:** programkonstrukcióra vonatkozó metaadat, nincs közvetlen hatása a programvégrehajtásra. Gépi feldolgozásban alkalmas, fordítási vagy futási időben elérhető. Dokumentációs célokat is szolgálnak.

**Lehetséges felhasználások:**

* **Információk szolgáltatása a fordítónak**: pl. tekintsen el a figyelmeztetésektől, jelezzen bizonyos hibákat. (@Deprecated, @Override)
* **Kódgenerálás**: Dagger, Jakarta XML Binding (JAXB), Project Lombok
* **Futásidejű feldolgozás:** bizonyos annotációkhoz hozzá lehet férni végrehajtási időben. Pl. egységtesztelés (JUnit), perzisztencia (Jackson), HTTP kliensek (Feign), Függőség befecskendezés (Spring Framework), Alkalmazás keretrendszerek (Micronaut Framework)

**Szintaxis:**

* Annotáció interfész név + opcionális lista, mely vesszővel elválasztott elem-érték párok alkotnak
* Az annotáció interfész határozza meg a használható elem-érték párokat, sorrendjük nem fontos
* Ha egy elem típusa tömb, akkor kapcsoszárójel közé kell írni (kivéve, ha egyelemű a tömb)

@Target({ElementType.METHOD}) = @Target(ElementType.METHOD)

**Fajtái:**

* **Közönséges annotáció**

@XmlElement(name = „creator”, namespace = „http://purl.org/dc/terms”, required = true)

* **Egyelemű annotáció**

@SuppressWarnings(value = „unchecked”)

@SuppressWarnings(„unchecked”)

@Target(value = {ElementType.FIELD, ElementType.METHOD})

@Target({ElementType.FIELD, ElementType.METHOD})

* **Jelölő annotáció:** ha nincs megadva elem-érték pár, akkor elhagyható a zárójel

@NotNull

@NotNull()

**Alkalmazása:**

* **Deklarációknál** (konstruktor, osztályváltozó, enum konstans, lokális változó, metódus, modul, csomag, formális paraméter, osztály, interfész, enum, típusparaméter, rekord, rekord komponens): Deklaráció annotáció
* **Deklarációkban és kifejezésekben használt típusokra**: Típus annotáció

**Előre definiált annotáció interfészek:**

* **@Deprecated:** annak a jelzésére szolgál, hogy az ellátott elem kerülendő (pl. veszélyes, elérhető jobb alternatíva). Dokumentálni @Deprecated Javadoc címkével.   
  Opcionális elemek:
  + **since** – melyik verzióban lett elavult az elem („”)
  + **forRemoval** – eltávolításra kerül-e a jövőben az elem (false)

Pl. @Deprecated(since=”9”, forRemoval=true)

jdeprscan – ezek észlelésére szolgáló parancssori kódelemző eszköz

* **@SuppressWarnings:** annak jelzésére szolgál, hogy el kell tekinteni az annotált elemen (és benne tartalmazó programelemeknél) az adott figyelmeztetésektől

Pl. @SuppressWarnings(„unchecked”), @SuppressWarnings(„deprecation”)

* **@Override:** annak jelzésére szolgál, hogy a megjelölt metódus felülír egy olyan metódust, melyet egy ősosztályban már deklaráltak. Nem kötelező megadni.
* **@FunctionalInterface:** annak jelzésére szolgál, hogy egy interfész funkcionális (pontosan egy absztrakt metódusa van)

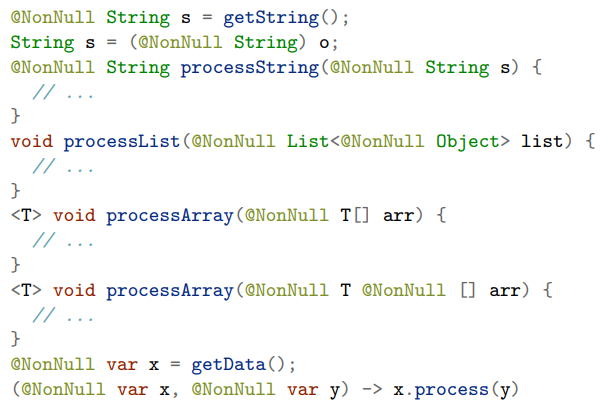
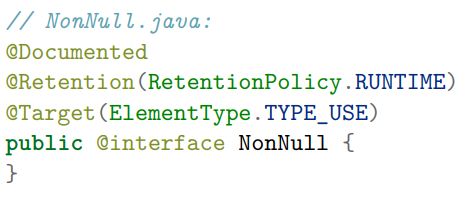
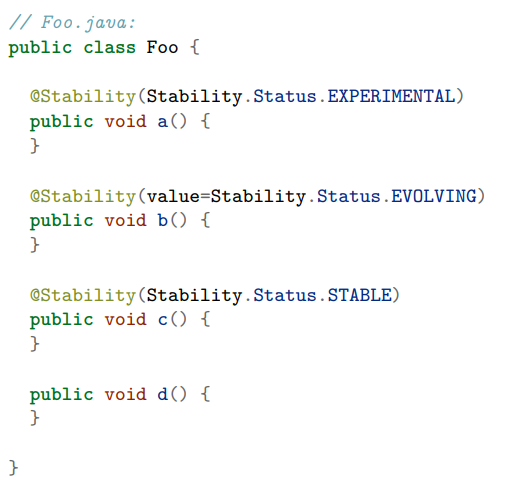
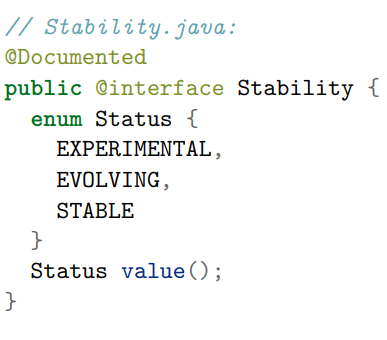
**Meta-annotációk:**

* **@Documented:** azt jelzi, hogy az annotáció interfész annotációi dokumentálva kell, hogy legyenek.
* **@Inherited:** azt jelzi, hogy egy annotáció interfész automatikusan öröklődik
* **@Repeatable:** azt jelzi, hogy egy annotáció interfész annotációi többször is alkalmazhatóak ugyanarra a deklarációra/típusra.
* **@Retention:** az interfész annotációihoz meghatározza a tárolás módját:
  + **RetentionPolicy.SOURCE** –a fordító figyelmen kívül hagyja az annotációkat
  + **RetentionPolicy.CLASS** – eltárolja bájtkódban, futásidőben nem elérhető
  + **RetentionPolicy.RUNTIME** – eltárolja bájtkódban, futásidőben elérhető
* **@Target:** meghatározza, hogy az annotáció interfész hol használható
  + **Annotáció interfész deklarációja:** ElementType.ANNOTATION\_TYPE
  + **Konstruktor deklaráció:** ElementType.CONSTRUCTOR
  + **Osztályváltozó, enum konstans deklarációja:** ElementType.FIELD
  + **Lokális változó deklarációja:** ElementType.LOCAL\_VARIABLE
  + **Metódus deklaráció:** ElementType.METHOD
  + **Modul deklaráció:** ElementType.MODULE
  + **Csomagdeklaráció:** ElementType.PACKAGE
  + **Formális paraméter deklarációja:** ElementType.PARAMETER
  + **Rekord komponens deklaráció:** ElementType.RECORD\_COMPONENT
  + **Osztály, interfész, enum, rekord deklarációja:** ElementType.TYPE
  + **Típusparaméter deklarációja:** ElementType.TYPE\_PARAMETER
  + **Típus használata:** ElementType.TYPE\_USE

**Annotáció interfész deklarálása:**

* módosítók @interface name { deklarációk }
* Az annotáció típusban a metódus deklarációk mindegyike egy elemet deklarál
* Megengedettek:
  + osztálydeklarációk
  + interfész deklarációk
  + konstans deklarációk
  + speciális metódus deklarációk
* Nem megengedettek:
  + formális paraméterek
  + típusparaméterek
  + throws kulcsszó
  + private, default, static módosítók
* Visszatérési típus lehet:
  + primitív típus
  + String
  + Class / Class<T1, …, Tn>
  + enum típus
  + Annotáció interfész típus
  + Tömb, mely eleminek típusa az előzők valamelyike
* Alapértelmezett érték adható default kulcsszóval
* **A képen szöveg, képernyőkép, Betűtípus látható

  Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**Egyelemű annotációknál a value-t szokás az elemnek adni



**Típus annotációk:** típusra vonatkozó annotáció, @Target(Element.TYPE\_USE) meta-annotációval megjelölt kell, hogy legyen.

Az annotáció interfész határozza meg, hogy egy annotáció egy deklaráció annotáció vagy típus annotáció, vagy mindkettő. Például:

@Foo private String s;

* **deklaráció annotáció** s-en, ha Foo meta-annotálva van a @Target(ElementType.FIELD) annotációval
* **típus annotáció** String-en, ha Foo meta-annotálva van a @Target(ElementType.TYPE\_USE) annotációval

1. **Szoftvertesztelés**

**Szoftvertesztelés:** célja a szoftverek minőségének megállapítása és a működés közbeni meghibásodási kockázat csökkentése. Annak a dinamikus verifikálása, hogy a program várt módon viselkedik egy tesztesetek véges halmazán.

**Hibát leíró szakkifejezések:**

* **Tévedés/tévesztés (error/mistake):** rossz eredményt adó emberi tevékenység. Egy személy egy tévedést/tévesztést követ el, mely egy hibát vezethet be a szoftver kódjába vagy munkatermékbe.
* **Hiba (defect/fault/bug):** tökéletlenség vagy hiányosság egy munkatermékben, melynél nem teljesülnek a követelmények/előírások. Ha végrehajtásra kerül a hiba, akkor meghibásodást okozhat, de nem minden esetben.
* **Meghibásodás (failure):** esemény, melynél egy komponens/rendszer nem lát egy megkövetelt funkciót a megszabott határok között.

**Tesztelés hét alapelve:**

* A tesztelés a hibák jelenlétét mutatja meg, nem a hiányukat
* Lehetetlen a kimerítő tesztelés
* A kora tesztelés időt és pénzt takarít meg
* A hibák csoportosulnak
* A tesztek elkopnak
* A tesztelés környezetfüggő
* A hibamentesség egy tévhit

**Teszteset:** tesztfeltételek alapján meghatározott előfeltételek, bemenetek, tevékenységek, elvárt eredmények, utófeltételek halmaza. **Tesztfeltétel**: egy komponens/rendszer tesztelhető vonatkozás, melyet a tesztelés alapjául választunk.

* **Magas szintű teszteset:** absztrakt előfeltételekkel, bementi adatokkal, elvárt eredményekkel, utófeltételekkel és lépésekkel rendelkezik
* **Alacsony szintű teszteset:** konkrét előfeltételekkel, bemeneti adatokkal, elvárt eredményekkel, utófeltételekkel és lépések részlete leírásával rendelkezik.

**Tesztadat:** tesztvégrehajtáshoz szükséges adatok. A konkrét értékek alacsony szintű tesztesetekké teszik a magas szintű teszteseteket. Ugyanaz a magas szintű teszteset különböző tesztadatokat használhat különböző végrehajtásoknál.

**Tesztelési szintek:**

* **Egység/komponens tesztelés** (Component / Unit testing)
  + Függetlenül tesztelhető komponensekre összpontosít
  + Általában az a fejlesztő végzi, aki a kódot írja.
  + Gyakran a kód megírása után írnak és hajtanak végre egységteszteket.
* **Komponensintegrációs tesztelés** (Component / Unit integration testing)
  + A komponensek közötti kommunikációra és interfészekre összpontosít.
  + Gyakran a fejlesztő felelősége, egységtesztelés után végzik, általában automatizált.
  + Magára az integrációra kell koncentrálnia, nem a komponensek működésére
* **Rendszertesztelés** (System testing)
  + A rendszer egészének viselkedésére összpontosít.
  + Jellemzően független tesztelők végzik specifikációkra támaszkodva.
* **Rendszerintegrációs tesztelés** (System integration testing)
  + A rendszerek közötti kommunikációra és interfészekre összpontosít.
  + Kiterjedhet külső szervezetekkel való interakciókra.
  + Rendszertesztelés után, vagy folyamatban lévő rendszertesztelési tevékenységgel párhuzamosan történhet, általában a tesztelők felelőssége.
  + Olyan tesztkörnyezetre van szükség, amely hasonló a működési környezethez.
* **Elfogadási tesztelés** (Acceptance testing)
  + Azt határozza meg, hogy a rendszer kész-e a telepítésre és a használatra
  + Gyakran az üzemeltetők felelőssége
  + Kiadása előtt néha odaadják potenciális felhasználóknak egy kis kiválasztott csoportjának (**alfa tesztelés**) és/vagy reprezentatív felhasználók egy nagyobb halmazának (**béta tesztelés**)

**Változással kapcsolatos tesztelés:** olyan tesztek, amikor módosítások történnek egy rendszerben egy hiba kijavításához vagy új funkció hozzáadásához / módosításához. Szükséges minden tesztelési szinten. Fajtái:

* **Megerősítő tesztelés:** annak megerősítése, hogy a hiba sikeresen kijavításra került.
* **Regressziós tesztelés:** a változások által okozott akaratlan mellékhatások érzékelése.

**A jó egységtesztek ismertetőjegyei: FIRST**

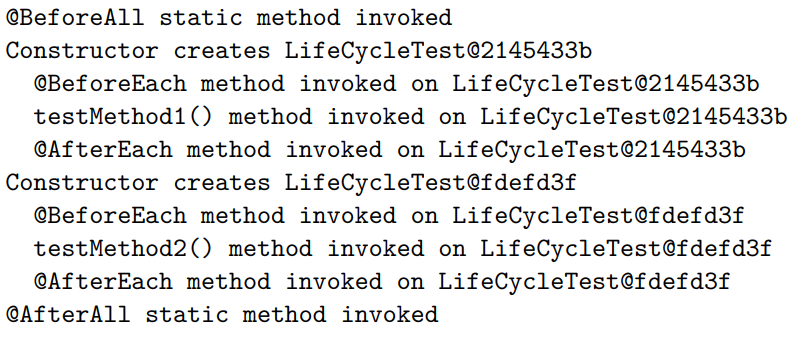
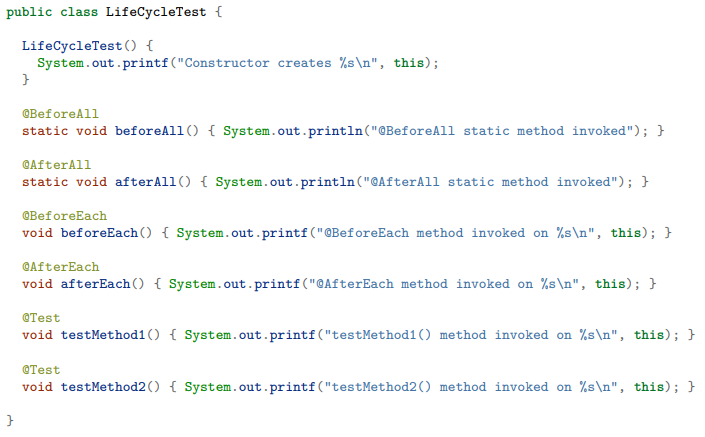
* **Gyors (Fast):** a tesztek gyorsak kell, hogy legyenek.
* **Független** **(Independent):** a tesztek nem függhetnek egymástól.
* **Megismételhető (Repeatable):** a tesztek bármely környezetben megismételhetők kell, hogy legyenek.
* **Önérvényesítő (Self-Validating):** a teszteknek logikai kimenete kell, hogy legyen. Vagy átmennek, vagy megbuknak.
* **Jól időzített (Timely):** a teszteket időben kell megírni, közvetlenül a tesztelendő kód előtt.

**Egységtesztek szervezése: az AAA minta**

* **Elrendez (Arrange):** felelős a tesztelt rendszer és függőségei egy kívánt állapotba állításáért.
* **Cselekszik (Act):** ez szolgál a tesztelt rendszer metódusainak meghívására, függőségek átadására, kimeneti érték elkapására.
* **Kijelent (Assert):** ez szolgál a kimenet ellenőrzésére (visszatérési érték, végső állapot)

**JUnit:**

* **Tesztosztály:** olyan osztály, amely legalább egy tesztmetódust tartalmaz. Nem lehet absztrakt és egyetlen konstruktora kell, hogy legyen.
* **Tesztmetódus:** @Test, @RepeatedTest, @ParameterizedTest, @TestFactory, @TestTemplate annotációval megjelölt metódus.
* **Teszt végrehajtási életciklus:** a JUnit egy új példányt hoz létre minden tesztosztályból a tesztmetódus végrehajtás előtt. Megváltoztatható a **@TestInstance(Lifecycle.PER\_CLASS)** annotációval.  
  **@BeforeAll** – a teszteket megelőzően fut le  
  **@BeforeEach** – minden egyes tesztet megelőzően fut le  
  **@AfterEach** – minden egyes teszt után fut le  
  **@AfterAll** – az összes teszt után fut le



* **Teszteredmények:**
  + **Siker (success):** minden eredmény megegyezik a várt végeredményekkel.
  + **Bukás (failure):** az eredmény nem egyezik meg a várt végeredménnyel.
  + **Hiba (error):** egy hiba következik be, mely megakadályozza a befejeződést. (kivétel)

**Kódlefedettségi metrikák:**

* **Utasítás lefedettség** = Végrehajtott utasítások / Összes utasítás száma
* **Sor lefedettség** =Végrehajtott kódsorok / Összes sor száma
* Mindent egyszer számolunk, sor lefedettség függ a forráskód formázástól.
* **Ág lefedettség (branch coverage):** a végrehajtott ágak arányát méri egy tesztkészlet futtatásakor az összes ág számához viszonyítva (if/switch).  
  **Ág lefedettség** = Végrehajtott ágak / Összes ág száma
* Veszélyes egy bizonyos érték elérésének megcélzása egy lefedettségi metrikánál, inkább a **megfelelő egységtesztelésre** kell koncentrálni. Jó, ha egy rendszer fő részeinél nagy a lefedettség, de nem jó ezt magas szintű követelménnyé tenni.

1. **Unified Modeling Language (UML)**

**Szakterület-specifikus nyelv (DSL):** egy bizonyos fajta problémára koncentráló számítógépes nyelv. Pl. LaTeX, CSS, DOT, SQL, Gradle DSL, Make, PlantUML

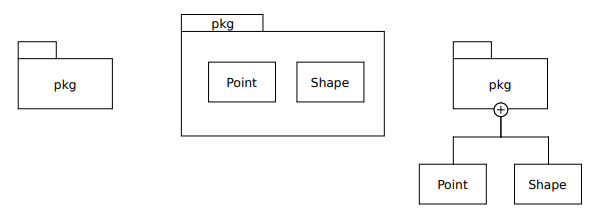
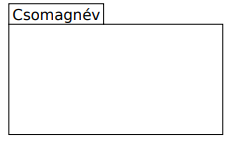
**UML modellelemek:**

* **Osztályozók:** közös jellemzőkkel (tulajdonságokkal, műveletekkel) rendelkező példányok halmazát ábrázolja, hierarchiába szervezhetőek. Specializációi:
  + **adattípus** (DataType)
  + **asszociáció** (Association)
  + **interfész** (Interface)
  + **osztály** (Class)
  + **…**

**Jelölése:** mint az osztályoké, félkövéren a név.

* **Csomagok:** egy modellelemek csoportosítására szolgáló konstrukció, mely egy névteret határoz meg a tagjai számára. Hivatkozás: csomagnév::elemnév

**Jelölése:**



* A képen szöveg, Betűtípus, képernyőkép, sor látható

  Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**Függőségek:** modellelemek közötti szolgáltató-kliens kapcsolatot jelent. **Jelölése:** szaggatott nyíl, a függő (kliens) a szolgáltató felé mutat. Megadható kulcsszó vagy sztereotípia.
* **Kulcsszavak:** szöveges annotációként jelenik meg egy UML grafikus elemhez kapcsolva, vagy UML diagram szövegsorának részeként. Lehetővé teszi azonos grafikus jelölésű UML fogalmak megkülönböztetését. Egy modellelemre több kulcsszó is vonatkozhat.  
  **Jelölése:** francia idézőjelek (« ») között.
* A képen szöveg, Betűtípus, képernyőkép, sor látható

  Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**Megjegyzések:** nincs jelentése, a modell olvasója számára hordozhat hasznos információt.  
  **Jelölése:** téglalap jobb felül lehajtva

**Osztálydiagram:** objektumok típusait írja le egy rendszerben és a köztük fennálló kapcsolatokat. Mutatja az osztályok tulajdonságait, műveleteit, valamit azokat a megszorításokat, melyek az objektumok összekapcsolására vonatkoznak. **Fajtái:**

* **Elemzési:** az osztályok az alkalmazási szakterület fogalmai, a szakterület felépítését modellezi.
* **Tervezési:** megjelennek a megvalósítás módjának technikai aspektusai.
* **Megvalósítási:** egy implementációs nyelv konstrukcióval ekvivalensek.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Osztály** | |  | | --- | | Név | | Attribútumok | | Műveletek | |
| **Láthatóság** | + (nyilvános) - (privát)  # (védett) ~ (csomagszintű) |
| **Számosság** | Megszorítás egy kollekció elemszámára: [alsó\_korlát ..] felső\_korlát 5 = 1..5, 10 = 1..10 5..7, 8..12, 3..\* 1 = 1..1, 5 = 5..5 0..\* = \* |
| **Tulajdonságok** | Egy attribútumot vagy egy asszociációvéget ábrázol:  [^] [láthatóság] [/] név [: típus] [[számosság]] [= alapérték] [{módosító [, módosító]\*}]  ^ - a tulajdonság örökölt-e / - a tulajdonság származtatott-e Számosság elhagyásakor alapértelemzés 1  Módosító: readOnly, ordered, unordered, unique, … Például: +/price: int [1] {readOnly} |
| **Műveletek** | [^] [láthatóság] név ([paraméterlista]) [: típus] [[számosság]] [, tulajdonság\*]\*}]  Tulajdonság: nonunique, ordered, query (nem változtatja meg a rendszer állapotát), redefines [név], seq/sequence, unique Például:  +getName(): String {query} +setTitle(title:String) |
| **Statikus attribútumok és műveletek** | A statikus attribútumokat és műveleteket aláhúzás jelöli |
| **Absztrakt osztályok** | Nevét dőlt betűvel szedjük és/vagy a név után vagy alatt megadjuk az {abstract} szöveges annotációt |
| **Asszociációk** | Szemantikus viszony, mely osztályozók példányai között áll fenn. Olyan példányok között áll fenn, melyek megfelelnek az asszociált típusoknak vagy implementálják azokat. Legalább két végük van (2 = bináris asszociáció). Egy **kapcsolat** egy asszociáció egy példánya. Jelölése: összekötő folytonos vonal, több esetén rombusz.  **Asszociáció vég**: a vonal és egy osztályozót ábrázoló ikon kapcsolata (elhelyezhető név, számosság, módosító, láthatóság) Elhelyezhető a vonal végén egy **pont**, ami azt jelenti, hogy a modell tartalmaz egy olyan tulajdonságot, melynek típusát a pont által érintett osztályozó ábrázolja. A pont hiánya azt jelzi, hogy a vég magához az asszociációhoz tartozik.  **Navigálhatóság**: a kapcsolatban résztvevő példányok hatékonyan elérhetőek az asszociáció többi végén lévő példányokból. |

|  |  |
| --- | --- |
| **Egész-rész kapcsolat** | Bináris asszociációk egész-részt kifejező fajtái:   * **Aggregáció (aggregation):** a kompozit objektum törlésekor a másik függetlenül létezhet. Pl. Csapat - Játékos * **Kompozíció (composition):** erősebb aggregáció, a teljes objektum hozzá tartozik. Kompozit objektum törlésekor az összes részobjektum vele együtt törlődik. Pl. Ház - Szoba |
| **Általánosítás** | Egy **általánosítás/specializáció** kapcsolatot határoz meg az osztályok között. Közvetlen általánosításokat az osztályozó **szülőjének** nevezzük, osztályok esetén **ősosztálynak**. Egy osztályozó példánya minden általánosítás példánya, örökli az általános osztályozó bizonyos tagjait. |
| **Interfészek** | Nyilvános jellemzőket és kötelezettségeket deklarál. Nem példányosíthatók, osztályozók implementálják vagy realizálják az interfészt. |

1. **Objektumorientált tervezési alapelvek**

**Statikus kódelemzés:** a programkód elemzésének folyamata, mely a kód végrehajtása nélkül történik.

**Statikus kódelemző:** statikus kódelemzést végző automatikus eszköz. Pl.:

* **C#**: InferSharp, Roslyn Analyzers, Roslynator
* **C++:** Cppcheck
* **ECMAScript/JavaScript**: ESLint, JSHint, JSLint, RSLint
* **Java**: Checkstyle, Error Prone, NullAway, SpotBugs
* **Python**: Prospector, Pylint
* **Több nyelvet támogató**: Coala, Infer, MegaLinter, PMD, Semgrep

**DRY (Don’t Repeat Yourself) elv:**

* A tudás minden darabkájának egyetlen, egyértelmű, hiteles reprezentációja kell, hogy legyen egy rendszerben
* Ellenkezője a WET (We enjoy typing, write everything twice, waste everyone’s time)
* Ismétlések fajtái:
  + **Kényszerített ismétlés**: fejlesztők szerint nincs választásuk, a környezet láthatólag megköveteli az ismétlést
  + **Nem szándékos ismétlés**: a fejlesztők nem veszik észre, hogy információkat duplikálnak
  + **Türelmetlen ismétlés**: a fejlesztő lustaságából fakad, ismétlés könnyebb útnak látszik
  + **Fejlesztők közötti ismétlés**: egy csapatban / különböző csapatokban többen duplikálnak egy információt.
* **Kódismétlés** – azonos forráskódrész, mely egynél többször fordul elő egy programban. Nem minden kódismétlés információ ismétlés!
* A megsértés nem mindig kódismétlés formájában jelenik meg. Pl:

**A képen szöveg, Betűtípus, sor, képernyőkép látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

Kiküszöbölhető a length adattag egy metódusra való kicserélésével.

A képen szöveg, képernyőkép, Betűtípus, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

* A jobb teljesítmény érdekében választható a DRY elv megsértése.
* **Reprezentációs ismétlés**: a kód gyakran függ a külvilágtól, mely mindig a DRY elv valamiféle megsértését vonja maga után.

**KISS (Keep it simple, stupid) elv:** az egyszerűségre való törekvés

**YAGNI (You aren’t gonna need it)** **elv:**

* Extrém programozás (XP) alapelve
* „Mindig akkor implementálj valamit, amikor tényleg szükséged van rá, soha ne akkor, amikor csak sejted, hogy kell”
* Csak azon képességekre vonatkozik, melyek egy feltételezett lehetőség támogatásához kerülnek beépítésre a szoftverbe, nem vonatkozik a szoftver módosítását könnyítő törekvésekre.
* Csak akkor járható stratégia, ha a kód könnyen változtatható.

**Csatoltság:**

* Egy szoftvermodul függésének mértéke egy másik szoftvermodultól
* **Szoros csatoltság**: bonyolultságot növeli, megnehezíti a kód, módosítását, karbantarthatóságot és újrafelhasználhatóságot csökkenti.
* **Laza csatoltság:** a kódot kiterjeszthetővé teszi, a kiterjeszthetőség pedig karbantarthatóvá, lehetővé teszi a párhuzamos fejlesztést.

**GoF alapelvek:**

* Interfészre programozzunk, ne implementációra.
* Részesítsük előnyben az objektum-összetételt az öröklődéssel szemben.

|  |  |
| --- | --- |
| Öröklődés | Objektum-összetétel |
| Pro:   * Használata egyszerű, programozási nyelv közvetlenül támogatja. * Könnyebbé teszi az újrafelhasznált megvalósítás módosítását is.   Con:   * A szűlőosztályoktól örökölt megvalósításokat futásidőben nem változtathatjuk meg, mert az öröklődés már fordításkor eldől. * A szülőosztályok alosztályaik fizikai ábrázolását is meghatározzák, legalább részben. * Az implementációs függőségek gondot okozhatnak az alosztályok újrafelhasználásánál. | Pro:   * Az objektumokat csak az interfészükön keresztül érhetjük el, nem szegjük meg az egységbezárás elvét. * Bármely objektumot lecserélhetünk egy másikra futásidőben, amíg típusaik egyeznek. * Az osztályok és osztályhierarchiák kicsik maradnak.   Con:   * A tervezés során több objektumunk lesz. |

**Példa öröklődésre:**class Animal {  
 public void speak() {  
 System.out.println("Some generic animal sound");  
 }  
}  
  
class Dog extends Animal {  
 @Override  
 public void speak() {  
 System.out.println("Woof!");  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Dog dog = new Dog();  
 dog.speak(); // Kiírja: Woof!  
 }  
}

**Példa objektum összetételre:**

interface SoundBehavior {  
 void speak();  
}  
  
class DogSound implements SoundBehavior {  
 public void speak() {  
 System.out.println("Woof!");  
 }  
}  
  
class AnimalSound implements SoundBehavior {  
 public void speak() {  
 System.out.println("Some generic animal sound");  
 }  
}  
  
class Dog {  
 private SoundBehavior sound;  
  
 public Dog() {  
 this.sound = new DogSound();  
 }  
  
 public void setSound(SoundBehavior sound) {  
 this.sound = sound;  
 }  
  
 public void speak() {  
 sound.speak();  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Dog dog = new Dog();  
 dog.speak(); // Kiírja: Woof!  
  
 // Dinamikusan változtatjuk a viselkedést  
 dog.setSound(new AnimalSound());  
 dog.speak(); // Kiírja: Some generic animal sound  
 }  
}

**SOLID alapelvek:**

* **Single Responsibility Principle – Egyszeres felelősség elve**
  + *Egy osztálynak egy oka legyen a változásra*
  + Egynél több felelősség esetén a felelősségek csatolttá válnak. Egy felelősségben történő változások gyengíthetik vagy gátolhatják az osztály azon képességét, hogy eleget tegyen a többi felelősségének.
  + Példa megsértésre:

**A képen szöveg, képernyőkép, Betűtípus, diagram látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

Problémák:

* + - a számítógépes geometriai alkalmazásnak tartalmaznia kell a grafikus felhasználói felületet
    - ha a grafikus alkalmazás miatt változik a Rectangle osztály, az szükségessé teheti a számítógépes geometriai alkalmazás összeállításának, tesztelésének, telepítésének megismétlését.

Megfelelő változat:

A képen szöveg, diagram, képernyőkép, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

* + Egy modul egy, és csak **egyetlen** **aktornak** (emberek egy olyan csoportja, akik azt akarják, hogy a szoftver ugyanúgy változzon) van alárendelve.
  + Megsértése:

A képen szöveg, képernyőkép, Betűtípus látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

* **Open/Closed Principle – Nyitva zárt elv**
  + *A szoftver entitások legyenek nyitottak a bővítésre, de zártak a módosításra.*
  + Nyitott a bővítésre: a modul viselkedése kiterjeszthető
  + Zárt a módosításra: modul viselkedésének kiterjesztése nem eredményezi a modul forráskódjának változását.
  + Megsértése**:**

**A képen szöveg, képernyőkép, Betűtípus, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

Megfelelő változat:

A képen szöveg, képernyőkép, diagram, Betűtípus látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

* **Liskov Substitution Principle – Liskov-féle helyettesítési elv**
  + *Ha az S típus a T típus altípusa, nem változhat meg egy program működése, ha benne a T típusú objektumokat S típusú objektumokkal helyettesítjük*.
  + Megsértése:

class Bird {

void fly() {

System.out.println("Flying");

}

}

class Ostrich extends Bird {

@Override

void fly() {

throw new UnsupportedOperationException("Ostriches can't fly");

}

}

Az Ostrich példány viselkedése megsérti a Bird osztály elvárását.

* **Interface Segregation Principle – Interfész szétválasztási elv**
  + *Nem szabad arra kényszeríteni az osztályokat, hogy olyan metódusoktól függjenek, melyeket nem használnak.*
  + **Vastag interfész:** ésszerűen szükségesnél több tagfüggvénnyel és baráttal rendelkező interfész. Problémái: sok felelősséget tartalmaznak, különböző metódusokat külön kliensek használnak, nem szándékos csatoltságot okoznak a kliensek között.
  + **Interfész szennyezés:** szükségtelen metódusokkal egy interfész szennyezése.
  + Megsértése:

A képen szöveg, képernyőkép, diagram, Betűtípus látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

Megfelelő változat:

**A képen szöveg, diagram, képernyőkép, Tervrajz látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

* **Dependency Inversion Principle – Függőség megfordítási elv**
  + *Magas szintű modulok (üzleti logika) ne függjenek alacsony szint moduloktól (pl. fájlkezelés, adatbázis, I/O). Mindkettő absztrakciótól (interfésztől) függjön.*
  + **Probléma**: magas szintű modulok függenek az alacsony szintű moduloktól. Ha alacsony szinten változás van, az hatással lesz a magas szintre is. Ez csökkenti az újrafelhasználhatóságot.
  + **Előnye**: a magas szintű modul független az implementáció részleteitől, könnyebb cserélni, tesztelni, újrafelhasználni a komponenseket. Az implementációk csak a saját interfészüket ismerik, nem a hívóikat.
  + **Megsértése**:

**A képen szöveg, Betűtípus, képernyőkép, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

* + **Megfelelő változat**

**A képen szöveg, diagram, Tervrajz, Párhuzamos látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

**Függőség befecskendezés:**

* Olyan szoftvertervezési elvek és minták összessége, melyek lazán csatolt kód fejlesztését teszik lehetővé.
* Egy objektumra egy olyan szolgáltatásként tekintünk, melyet más objektumok kliensként használnak.
* Az objektumok közötti kliens-szolgáltató kapcsolatot függésnek nevezzük. Ez a kapcsolat tranzitív.
* **Függőség:** kliens által igényelt szolgálatás, mely a feladatának ellátásához szükséges.
* **Függő:** olyan kliens objektum, melynek függőségekre van szüksége a feladatának ellátásához.
* **Objektum gráf:** függő objektumok és függőségeinek összessége
* **Befecskendezés:** egy kliens függőségeinek megadása
* **DI konténer:** függőség befecskendezési funkcionalitást nyújtó programkönyvtár.
* **Tiszta DI:** függőség befecskendezés alkalmazásának gyakorlata DI konténer nélkül.
* **Példák**
  + nincs függőség befecskendezés: **A képen szöveg, képernyőkép, képernyő, Betűtípus látható

    Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**
  + függőség befecskendezés konstruktorral**A képen szöveg, képernyőkép, Betűtípus, sor látható

    Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**
  + függőség befecskendezés interfésszel**A képen szöveg, képernyőkép, képernyő, Betűtípus látható

    Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**
* **Előnyei:**
  + kiterjeszthetőség
  + karbantarthatóság
  + tesztelhetőség

1. **Minták a szoftverfejlesztésben**

**Minta:** egy gyakori probléma vagy kérdés általános megoldásának leírása, melyből meghatározható egy konkrét probléma részletes megoldása.

**Architekturális minták:**

* Szoftverrendszerek alapvető szerkezeti felépítésére adnak sémákat.
* Előre definiált alrendszereket biztosítanak, meghatározzák ezek felelősségi köreit, valamint szabályokat és irányelveket tartalmaznak a közöttük lévő kapcsolatok szervezésére vonatkozólag.

**MVC (Model-View-Controller) – Modell-nézet vezérlő:**

* **Környezet:** rugalmas ember-gép (grafikus) felülettel rendelkező interaktív alkalmazások.
* **Probléma:** gyakori az igény a felhasználói felületek változására
* **Megoldás:** három részre osztás:
  + A modell komponens az adatokat és funkcionalitást csomagolja be, független a kimenet ábrázolásmódjától vagy az input viselkedésétől.
  + Nézet komponensek jelenítik meg az információkat a felhasználónak.
  + A vezérlő fogadja a bemenetet, mely szolgáltatáskérésekké alakít a modell vagy nézet felé.
* A modell elválasztása több nézetet is lehetővé tesz ugyanahhoz a modellhez.
* A nézet elválasztása a vezérlő komponenstől kevésbé fontos.
* A képen szöveg, diagram, sor, képernyőkép látható

  Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.A modell a szakterület valamilyen információját ábrázoló objektum, mely adatokat csomagol be.
* Példa:
* **Változatok:**
  + Hierarchikus modell-nézet vezérlő (HMVC)
  + Model-template-view (MTV)
  + Model-view-presenter (MVP)
  + Model-view-viewmodel (MVVM)
* **Keretrendszerek**: Spring Framework, Laravel, Django, JavaFX (valójában nem az, de megvalósítható az MVC minta)

**Tervezési minták**:

* Középszintű minták, kisebb léptékűek az architekturális mintáknál.
* Meghatározzák egy alrendszer felépítését.
* Függetlenek programozási nyelvektől/paradigmáktól.
* **Osztályzás** céljuk szerint:
  + **Létrehozási minták**: az objektumok létrehozásával foglalkoznak

|  |  |
| --- | --- |
| **Elvont gyár** | **Cél:** kapcsolódó vagy egymástól függő objektumok családjának létrehozására szolgáló felületet biztosít a konkrét osztályok megadása nélkül.  **Ismert felhasználása:** java.sql.Connection Pl. Interfészek és konkrét implementációk:  Maga a gyártó interfész és osztály: |
| **Egyke (Singleton)** | **Cél:** Egy osztályból csak egy példányt engedélyez, és ehhez globális hozzáférési pontot ad meg.  **Ismert felhasználása**: java.io.Console, java.lang.Runtime  Pl.: |
| **Építő (Builder)** | **Cél:** Az összetett objektumok felépítését függetleníti az ábrázolásuktól, így ugyanazzal az építési folyamattal különböző ábrázolásokat hozhatunk létre.  **Ismert felhasználása**: java.lang.Appendable (StringBuilder), java.lang.ProcessBuilder, java.util.StringJoiner, lombok  **Pl.** Osztály, amiből gyártani szeretnénk:  Az építő osztály:    Használata:  var s = new StringJoiner(",", "[", "]")  .add("John")  .add("Paul")  .add("George")  .add("Ringo")  .toString(); // "[John,Paul,George,Ringo]" |
| **Gyártófüggvények** | **Cél:**   * Felületet határoz meg egy objektum létrehozásához, az alosztályokra bízva, melyik osztályt példányosítják. * A gyártófüggvények megengedik az osztályoknak, hogy a példányosítást az alosztályokra ruházzák át.   **Ismert felhasználása**: javax.xml.parsers.\*  **Pl.** Interfész és implementációi**:**    Használata: |
| **Objektumkészlet** | **Cél:** Inicializált objektumok egy halmazát tartja nyilván az igények kiszolgálásához, ahelyett, hogy létrehozná és megsemmisítené az objektumokat.  **Ismert felhasználása**: Apache Commons Pool, Adatbázis kapcsolatok gyorsítótárazása  **Pl.:** Autó osztály, amit újrahasznosítani akarunk    **Objektumkészlet:**  **Használata:** |

* + **Szerkezeti minták**: azzal foglalkoznak, hogy hogyan alkotnak osztályok és objektumok nagyobb szerkezeteket.

|  |  |
| --- | --- |
| **Díszítő** | **Cél:**   * Az objektumokhoz dinamikusan további felelősségi köröket rendel. * A kiegészítő szolgáltatások biztosítása terén e módszer rugalmas alternatívája az alosztályok létrehozásának.   **Pl.:** Troll interéfsz és osztály:    Díszítő osztály:    Használata: |
| **Illesztő** | **Cél:**   * Az adott osztály interfészét az ügyfelek által igényelt interfésszé alakítja. * E módszerrel az egyébként összeférhetetlen interfészű osztályok együttműködését biztosíthatjuk.   **Példa:** RowingBoat és FishingBoat osztályok    Ahhoz, hogy a kapitány tudjon mozogni, kell neki egy RowingBoat    Viszont, ha csak FishingBoat-ja van, ahhoz kell csinálni egy adaptert    Így tud mozogni a FishingBoat-tal is |

* + **Viselkedési minták**: az osztályok vagy objektumok egymásra hatását, valamint a felelősségek elosztását írják le.

|  |  |
| --- | --- |
| **Bejáró** | **Cél:** Az összetett objektumok elemeinek soros elérését a háttérben megbúvó ábrázolás felfedése nélkül biztosító módszer kialakítása.  **Pl.** Szelet osztály    Pizza osztály  Használata |
| **Sablonfüggvény** | **Cél:**   * Egy adott művelet algoritmusának vázát elkészíteni, amelynek egyes lépéseit alosztályokra ruházzuk át. * Így az alosztályok az algoritmus egyes lépéseit felülbírálhatják, anélkül, hogy az algoritmus szerkezete módosulna.   **Pl.:** Italkészítő absztrakt osztály    TeaKeszito osztály    KaveKeszito osztály    Használat |
| **Megfigyelő** | **Cél:** Objektumok között egy sok-sok függőségi kapcsolatot létrehozni, így amikor az egyik objektum állapota megváltozik, minden tőle függő objektum értesül erről és automatikusan frissül.  **Pl.** Megfigyelő interfész (Ember)  Az objektum, ami változik (Hűtő)    Konkrét megfigyelők  Használata: |

**Programozási idiómák/implementációs minták**

* Egy idióma egy programozási nyelvre jellemző alacsony szintű minta.
* Leírja hogyan valósítsuk meg komponensek és kapcsolataik bizonyos vonatkozásait az adott nyelv eszközrendszerével.
* equals(), hashCode() felülírása

**A képen szöveg, képernyőkép, képernyő, szoftver látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.** **A képen szöveg, Betűtípus, sor, képernyőkép látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.** A képen szöveg, képernyőkép, Betűtípus, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

Egy rekord osztálynak van equals(), hashCode() metódusa!

* clone() felülírása

A képen szöveg, képernyőkép, Betűtípus, szám látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

Ha az osztály adattagjai olyan objektumra hivatkoznak, melyek megváltozhatnak, akkor mélymásolást kell használni.

A képen szöveg, képernyőkép, Betűtípus, képernyő látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

**Antiminták:**

* Egy problémára adott általánosan előforduló megoldások, melyek kifejezetten negatív következményen járnak, bármely szinten megjelennek.
* Kategóriák:
  + Szoftverfejlesztési antiminták
  + Szoftver architekturális antiminták
  + Szoftverprojekt vezetési antiminták
* **Tervezési szag**: olyan struktúrák a tervezésben, melyek alapvető tervezési elvek megsértését jelzik és negatív hatással vannak a tervezés minőségére

**Massza:**

* *Olyan tervezésnél fordul elő, ahol a feldolgozást egyetlen osztály sajátítja ki magának, a többi pedig elsősorban adatokat zár egységbe.*
* Egyetlen bonyolult osztályt körbevesznek egyszerű adatosztályok.
* A massza általában procedurális tervezésű
* **Következmények**:
  + Egy ilyen osztály túl bonyolult az újrafelhasználáshoz vagy teszteléshez
  + Költséges egy ilyen osztály memóriába való betöltése, egyszerű műveletekhez sok erőforrást használ.
* **Okok**:
  + Objektumorientált architektúra hiánya
  + Bármilyen architektúra hiánya
  + Az architektúra kikényszerítésének hiánya
  + Túl korlátozott beavatkozás
  + Kódolt katasztrófa
* **Kivétel**: elfogadható kompatibilitási okokból megtartott korábbi rendszer becsomagolásakor
* **Megoldás**: kódújraszervezés

**Spagetti kód:**

* Klasszikus, leghíresebb antiminta, amely egyidős a programozási nyelvekkel.
* Strukturálatlan, nehezen átlátható programkódként jelenik meg.
* Objektumorientált nyelvek esetén kevés osztály jellemzi, ahol a metódus megvalósítása nagyon hosszú.
* **Tünetek és következmények:**
  + A metódusok nagyon folyamat-orientáltak, objektum = folyamat
  + Végrehajtást az objektum implementáció határozza meg, nem az objektum kliensei.
  + Kevés kapcsolat van az objektumok között.
  + Sok a paraméter nélküli metódus, melyek osztályszintű és globális változókat használnak.
  + Nehéz a kód újrafelhasználása.
  + Elvesznek az objektumorientáltság előnyei, nincs öröklés / polimorfizmus.
  + A karbantartás súlyosbítják a problémát.
  + Költségesebb a karbantartás, mint új megoldás kifejlesztése.
* **Okok:**
  + Tapasztalatlanság az objektumorientált tervezés terén
  + Nincs mentorálás, nem megfelelő kódátvizsgálás
  + Nincs implementálást megelőző tervezés
  + A fejlesztők elszigetelten dolgoznak
* **Kivétel:** elfogadható, ha az interfészek következetesek és csak az implementáció spagetti.
* **Megoldás:** kódújraszervezés
* **Kapcsolódó megoldások:** analízis-paralízis, lávafolyás

1. **Tiszta kód**

**Tiszta kód:** egy olyan fejlesztő számára is olvasható, továbbfejleszthető, aki nem az eredeti szerző.

**Értelmes nevek:**

* Olyan neveket használjunk a kódban, melyekből kiderül a szándék.
* Kerüljük a félrevezető neveket (pl. accountList, ha nem listáról van szó)
* Ne használjunk olyan túl általános zajszavakat, mint például Data, Info, Object
* Használjunk kiejthető neveket
* Használjunk kereshető neveket (egybetűs neveket csak lokális változókhoz rövid metódusokban)
* Egy név hossza meg kell, hogy feleljen a hatásköre méretének
* Kerüljük a kódolásokat a nevekben, melyek a típusról vagy a hatáskörről szolgáltatnak információkat.
* Osztályok neveként használjunk főneveket vagy főnévi kifejezéseket.
* Metódusok neveként használjunk igéket vagy igei kifejezéseket.
* Elnevezéseknél kerüljük a jópofáskodást.
* Következetesen alkossunk neveket.
* Használjunk olyan neveket, melyeket a többi programozó is megért.
* Használjuk az adott problématerület neveit.
* Nevek kontextusba helyezése (pl. street, city, zipCode, state változók egy címet alkotnak, de ha csak a state változót látjuk egy metódusban nem biztos, hogy ez egy cím része)

**Függvények:**

* A függvények nagyon rövidek kell, hogy legyenek (inkább 2-4 sorosak)
* Egy dolgot csináljanak, de azt jól.
* Függvényenként egy absztrakciós szint
  + Az utasítások azonos absztrakciós szintűek kell, hogy legyenek
  + Stepdown szabály: felülről lefelé haladva csökkenjen a függvények absztrakciós szintje (felülről lefelé haladva olvasható)
* **Argumentumok**
  + Niladikus**:** argumentum nélküli, ideális
  + Monadikus**:** egyargumentumú
  + Diadikus**:** kétargumentumú
  + Triadikus**:** háromargumentumú, lehetőleg kerülni kell
  + Poliadikus**:** háromnál több argumentumú, soha ne használjuk
  + Kettőnél/háromnál több argumentum esetén érdemes lehet becsomagolni egy osztályba.
  + Változó argumentumszámú függvényekre lista argumentumú függvényként tekinthetünk
* **Mellékhatásmentesség**:
  + Csak azt csinálják, amit ígérnek
  + Kerülni kell output argumentumok használatát
* **Parancs-lekérdezés szétválasztás**: egy függvény vagy csináljon valamit, vagy válaszoljon valamit, de egyszerre mindkettőt ne.
* Hibakód visszaadása helyett részesítsük előnybe a kivételeket.
* try/catch blokkokat emeljük ki önálló függvényekbe. Például:  
  **A képen szöveg, képernyőkép, képernyő, Betűtípus látható

  Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**
* Használható egy függvényben akár több return utasítás, ciklusokban break és continue utasítás.

**Megjegyzések:**

* Azért nemkívánatosak a megjegyzések, mert nem mindig, és nem szándékosan, de túl gyakran közölnek pontatlan vagy valótlan információt. Ezek rosszabbak, mint a megjegyzések teljes hiánya.
* Ha a kód változik, fejlődik, akkor nem minden esetben követnek megjegyzések.

**Jó megjegyzések:**

* Jogi megjegyzések
* Informatív megjegyzések

**A képen szöveg, képernyőkép, Betűtípus, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

* Szándékot magyarázó megjegyzés

**A képen szöveg, képernyőkép, Betűtípus, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

* Tisztázó megjegyzés

**A képen szöveg, képernyőkép, Betűtípus, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

* Következményekre figyelmeztető megjegyzés

**A képen szöveg, képernyőkép, Betűtípus, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

* TODO megjegyzés

**A képen szöveg, képernyőkép, Betűtípus, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

* Megerősítő megjegyzés
* Javadoc megjegyzés nyilvános API-ban

**Rossz megjegyzések:**

* **Motyogás**

**A képen szöveg, Betűtípus, sor, fehér látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

* **Fölösleges megjegyzés**
* **Félrevezető megjegyzés**

**A képen szöveg, képernyőkép, Betűtípus, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

* **Redundáns megjegyzés:** nem szolgál semmiféle plusz információval a kódról
* **Kötelező megjegyzés:** nyilvánvalót magyarázó megjegyzés, ami csak azért van ott, mert megkövetelik, hogy tartozzon dokumentációs megjegyzés
* **Napló megjegyzés:** modul elején a benne végzett minden egyes módosítást naplószerűen dokumentáló megjegyzés
* **Zaj-megjegyzés**

**A képen szöveg, képernyőkép, Betűtípus, szám látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

* Pozíciójelző/szalagcím megjegyzés

**A képen szöveg, képernyőkép, sor, Betűtípus látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

**A képen szöveg, Betűtípus, képernyőkép, tervezés látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

* Záró kapcsos zárójel megjegyzés

**A képen szöveg, Betűtípus, képernyőkép, szám látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

* **Szerző neve megjegyzésben**: verziókezelő rendszerek feleslegessé tették
* **Megjegyzésbe tett kód:** rossz megjegyzések legutálatosabbja, kerülendő. Mások azt gondolják, hogy okkal van ott, fontos, ezért nem lesz bátorságuk törölni.
* **HTML megjegyzés:** rontja a forráskód olvashatóságát
* **Nem lokális megjegyzés:** olyan megjegyzés, mely nem közvetlen környezetében lévő kódra vonatkozik
* **Túl sok információt tartalmazó megjegyzés**
* **A kódhoz nem nyilvánvalóan kapcsolódó megjegyzés**
* **Javadoc megjegyzés nem nyilvános kódban**

**Formázás:** úgy kell formázni, hogy az olvasható legyen

**Újság metafora:** egy jól megírt újságcikket felülről lefelé haladva olvasunk. Egy forrásállomány is legyen olyan, mint egy újságcikk.

**Függőleges formázás:**

* Egy-egy üres sor jelöljön minden új fogalmat.
* Szorosan kapcsolódó programsorok sűrűn kell, hogy megjelenjenek
* A szorosan kapcsolódó fogalmakat tartsuk függőlegesen egymáshoz közel
* Változókat deklaráljuk a használatuk helyéhez a lehető legközelebb
* A példányváltozókat az osztályok elején kell deklarálni
* Ha egy függvény meghív egy másikat, akkor függőlegesen közel kell, hogy legyenek egymáshoz. Lehetőleg a hívó előzze meg a hívottat.

**Vízszintes formázás:**

* A sorok ne legyenek túl hosszúak. (<120 karakter)
* Használjunk szóközöket a gyengén összetartozó elemek között.
* Nem érdemes a deklarációban és értékadásokban a neveket és a kifejezéseket igazítani.
* Nagyon fontos a megfelelő behúzás
* Ha for vagy while ciklus törzseként üres utasítást kell használni, ezt célszerű egy új sorba helyezni.

**Hibakezelés:**

* Hibakód helyett kivétel
* Használjunk nem ellenőrzött kivételeket
* Ne adjunk át/vissza null-t

**Ellenőrzött kivételek:**

* Olyan kivétel, amit a fordító kényszerít kezelni.
* Arra kényszerítik a programozót, hogy foglalkozzon velük, mivel el kell kapni őket, így a kód megbízhatóságát növelik.

**Nem ellenőrzött kivételek:**

* Bárhol dobhatók egy metódus vagy konstruktor törzsében (pl. java.lang.RuntimeException, java.lang.Error alosztályai)

**Null értékek át/visszaadása:**

* Ne
* null referenciák NullPointerException kivételeket okoznak.
* Ha kellene, vegyük helyette fontolóra inkább egy kivétel dobását vagy egy speciális eset objektum (üres lista) visszaadását.
* Visszaadás rossz, átadás megrosszabb. Amikor lehetséges, el kell kerülni.

1. **A Java SE/JDK új lehetőségei**

**Boilerplate kód:** olyan láthatólag gyakran ismétlődő kód, mely újra és újra felbukkan valamilyen eredmény elérése céljából, és amelyről látszik, hogy sokkal egyszerűbbnek kellene lennie. Például: A képen szöveg, képernyőkép, Betűtípus látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

A boilerplate kód mennyiségének csökkentése Java-ban:

* Lombok

@lombok.Data  
public class Name {

String firstName;  
 String lastName;

}

* Rekordok

public record Name(String firstName, String lastName) {}

**Előzetes lehetőségek:**

* Egy előzetes lehetőség a Java nyelv, a virtuális gép vagy a Java SE API egy új lehetősége, mely pontosan meghatározott, teljesen implementált, de még nem végleges.
* Fajtái:
  + előzetes nyelvi lehetőségek
  + előzetes VM lehetőségek
  + előzetes API-k
* Soha nem kísérleti, kockázatos, hiányos vagy instabil.
* --enable-preview parancssori opciót kell megadni az előzetes lehetőségek engedélyezéséhez a JDK parancssori eszközöknek (java, javac (--release <n> vagy --source <n> valamelyike is kell), jshell).
* Mavennél:

**A képen szöveg, Betűtípus, képernyőkép látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

**Switch (Java SE 14):**

* Utasításként és kifejezésként is használható, mindkét formánál használhatók hagyományos case címkék (áteséssel), vagy új case cimkék (átesés nélkül)
* Rendelkezésre áll egy új utasítás (yield) is, mellyel egy érték adható vissza a switch kifejezésből.
* Vagy értékkel, vagy kivétel dobással kell befejeződnie.
* Switch címke új formája: case … -> <kifejezés, blokk, throw>
* default záradék szükséges, kivétel az összes enum konstanst lefedő enum switch kifejezéseknél

A képen szöveg, Betűtípus, képernyőkép, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.A képen szöveg, Betűtípus, képernyőkép, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.A képen szöveg, képernyőkép, Betűtípus látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

**Szövegblokkok (Java SE 15):** egy többsoros sztring literál, mely bárhol használható, ahol egy közönséges sztring literál.

A képen szöveg, képernyőkép, Betűtípus látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

**Mintaillesztés az instanceof operátorhoz (Java SE 16):**

* Lehetővé teszi egy programban komponensek objektumokból történő feltételes kinyerésének tömörebb és biztonságosabb kifejezését.
* A minta-érzékeny konstrukciók változókat vezethetnek be egy kifejezés közepén.
* Hagyományos mód:

A képen szöveg, Betűtípus, kézírás, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

Mintaillesztéssel:

A képen szöveg, Betűtípus, sor, fehér látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

* equals() hagyományosan:

A képen szöveg, Betűtípus, képernyőkép, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

equals() implementálása mintaillesztéssel:

A képen szöveg, képernyőkép, Betűtípus látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

**Rekordok (Java SE 16):**

* Nem módosítható adatokat becsomagoló újfajta osztályok.
* A rekord példányok rekord komponenseknek nevezett rögzített értékek halmazát ábrázolják.
* Minden egyes komponenshez van egy implicit módon deklarált lekérdező metódusa (getter). Van implicit módon deklarált konstruktora, equals(), hashCode() és toString() metódusa is.

**A képen szöveg, képernyőkép, Betűtípus, szám látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

**A képen szöveg, képernyőkép, Betűtípus, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**