I. UNIFIED MODELING LANGUAGE (UML)

1. Mi az UML?

AZ OMG Egységes Modellező Nyelve szoftverrendszerek modelljeinek – ideértve azok felépítését és tervezését – részletes leírását, megjelenítését és dokumentálását segítő nyelv. (Üzleti modellezéshez és más nem szoftver rendszerek modellezéséhez is használható.)

2. Modell és metamodell fogalma

Modell: A modell egy rendszer leírása, ahol a rendszer a lehető legszélesebb jelentésben értendő (például szoftverek, szervezetek, folyamatok, …). A rendszert egy bizonyos nézőpontból írja le érintettek egy bizonyos csoportjának (például a rendszer tervezői vagy felhasználói) számára egy bizonyos absztrakciós szinten. Teljes abban az értelemben, hogy az egész rendszert lefedi, bár csak azon aspektusai kerülnek ábrázolásra benne, melyek lényegesek céljának szempontjából.

Metamodell: Egy modell modellje. Az UML-ben a metamodell egy olyan modell, mely önmagát modellezi. Nem csupán saját maga, hanem más modellek és metamodellek modellezésére is használható. Például a MOF modell egy metamodell.

3. Szakterület-specifikus nyelvek, példák

Domain-Specific language, DSL: Egy bizonyos fajta problémára koncentráló számítógépes nyelv, nem pedig egy általános célú nyelv tetszőleges fajta problémák megoldásához.

Példák: BibTeX/LaTeX, CSS, DOT (Graphviz), Gradle DSL, Make, PlantUML, SQL, …

4. UML modellelemek: osztályozók, csomagok, függőségek, kulcsszavak, megjegyzések

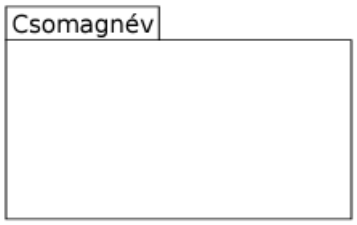
Osztályozók:

- Az osztályozó egy modellelem, mely közös jellemzőkkel (tulajdonságokkal, műveletekkel) rendelkező példányok egy halmazát ábrázolja.
- Hierarchiába szervezhetők az általánosítás révén.
- Specializációi: adattípus (DataType), asszociáció (Association), interfész (Interface), osztály (Class), …
- Jelölésmód: mint az osztályoké, a nevük megjelenítéséhez félkövér betűtípust kell használni.

Csomagok:

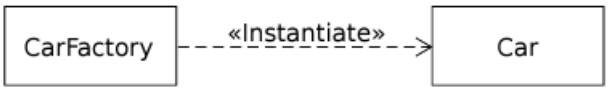
- A csomag egy modellelemek csoportosítására szolgáló konstrukció, mely egy névteret határoz meg a tagjai számára.

- A tartalmazott elemekre csomagnév::elemnév formájú minősített nevekkel lehet hivatkozni (például pkg::Point, pkg::Shape).



Függőségek:

- Modellelemek közötti szolgáltató-kliens kapcsolatot jelent, ahol egy szolgáltató módosításának hatása lehet a kliens modellelemekre.
- Jelölésmód: Két modellelem közötti szaggatott nyíl jelöli. A nyíl a függő (kliens) modellelemtől a szolgáltató modellelem felé mutat. A függőséghez megadható egy kulcsszó vagy sztereotípia.

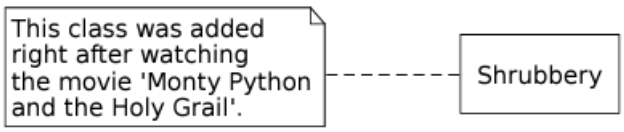


Kulcsszavak:

- Az UML jelölésmód szerves részét képző fenntartott szó.
- Szöveges annotációként jelenik meg egy UML grafikus elemhez kapcsolva vagy egy UML diagram egy szövegsorának részeként.
- Minden egyes kulcsszóhoz elő van írva, hogy hol jelenhet meg.
- Lehetővé teszi azonos grafikus jelölésű UML fogalmak (metaosztályok) megkülönböztetését.
- Megadásuk francia idézőjelek – « és » karakterek – között. Egy modellelemre több kulcsszó is vonatkozhat.

Megjegyzések:

- Nincs jelentése, a modell olvasója számára hordozhat hasznos információt.
- Jelölésmód: A jobb felső sarkában „szamárfüles” téglalap ábrázolja. A téglalap tartalmazza a megjegyzés törzsét. Szaggatott vonal kapcsolja a magyarázandó elem(ek)hez. A vonal elhagyható, ha egyértelmű a környezetből vagy nem fontos a diagramon.



5. Osztálydiagramok, osztálydiagramok fajtái

Osztálydiagram: Egy osztálydiagram az objektumok típusait írja le egy rendszerben és a köztük fennálló különféle statikus kapcsolatokat. Az osztálydiagramok mutatják az osztályok tulajdonságait és műveleteit is, valamint azokat a megszorításokat, melyek az objektumok összekapcsolására vonatkoznak.

Osztálydiagramok fajtái (Störrle):

- Elemzési: Az elemzési szinten az osztályok az alkalmazási szakterület fogalmai, az osztálydiagram a szakterület felépítését modellezi.

- Tervezési: Megjelennek az osztályokban a megvalósítás módjának technikai aspektusai.
- Megvalósítási: Az osztályok egy implementációs nyelv (például C++, Java, …) konstrukcióival ekvivalensek.

6. Osztályok

Jelölésmód:

7. Láthatóság

- + (nyilvános)
- - (privát)
- # (védett)
- ~ (csomagszintű)

8. Számosság

Megszorítást fejez ki egy kollekció elemeinek számára.

- Az elemek száma nem lehet kisebb az adott alsó korlátnál.
- Az elemek száma nem lehet nagyobb az adott felső korlátnál, ha az nem *.

Jelölésmód: [alsó_korlát ..] felső_korlát

- Az alsó korlát nemnegatív egész, a felső korlát nemnegatív egész vagy a „korlátlan” jelentésű *.
- Ha az alsó és felső korlát egyenlő, akkor használható önmagában csak a felső korlát. –Például 1 ekvivalens az 1..1 számossággal, hasonlóan 5 ekvivalens az 5..5 számossággal.
- A 0..* számosság helyett használható az ekvivalens * jelölés.

9. Tulajdonságok

Egy tulajdonság egy attribútumot vagy egy asszociációvéget ábrázol.

Jelölésmód: [^] [láthatóság] [/] név [: típus] [[számosság]] [= alapérték] [{ módosító [, módosító]* }]

- A ^ azt jelzi, hogy a tulajdonság örökölt (UML 2.5).
- A / azt jelzi, hogy a tulajdonság származtatott.
- A számosság elhagyásakor az alapértelmezés 1.
- Módosító: például readOnly, ordered, unordered, unique, …

10. Műveletek

Jelölésmód: [^] [láthatóság] név ([paraméterlista])

[: típus] [[számosság]]

[{ tulajdonság [, tulajdonság]* }]

Tulajdonság: nonunique, ordered, query, redefines név, seq/sequence, unique, unordered, megszorítás

query: azt jelenti, hogy a művelet nem változtatja meg a rendszer állapotát.

11. Statikus attribútumok és műveletek

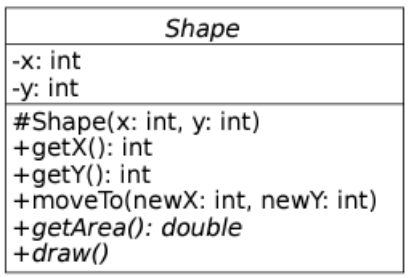
A statikus attribútumokat és műveleteket aláhúzás jelöli.

12. Absztrakt osztályok

Nem példányosítható osztály (osztályozó).

Jelölésmód:

- Szedjük az osztály (osztályozó) nevét dőlt betűvel és/vagy a név után vagy alatt adjuk meg az {abstract} szöveges annotációt.
- Az UML 2.5.1 nem rendelkezik az absztrakt műveletek jelölésmódjáról! (Személyes vélemény: ez valószínűleg hiba.)



13. Asszociációk

Szemantikus viszonyt jelent, mely osztályozók példányai között állhat fenn.

Azt fejezi ki az asszociáció, hogy kapcsolatok lehetnek olyan példányok között, melyek megfelelnek az asszociált típusoknak vagy implementálják azokat.

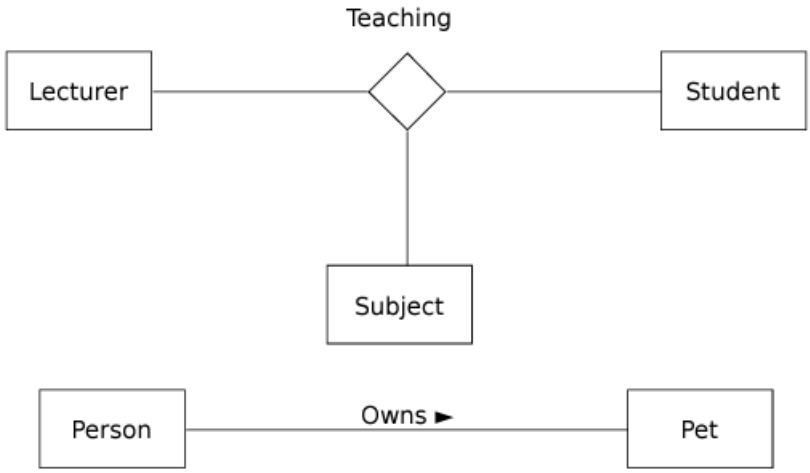
Legalább két végük van. Két végű asszociáció: bináris asszociáció.

Egy kapcsolat (link) egy asszociáció egy példánya. Azaz egy olyan n-es, mely minden véghez a vég típusának egy példányát tartalmazza.

Jelölésmód:

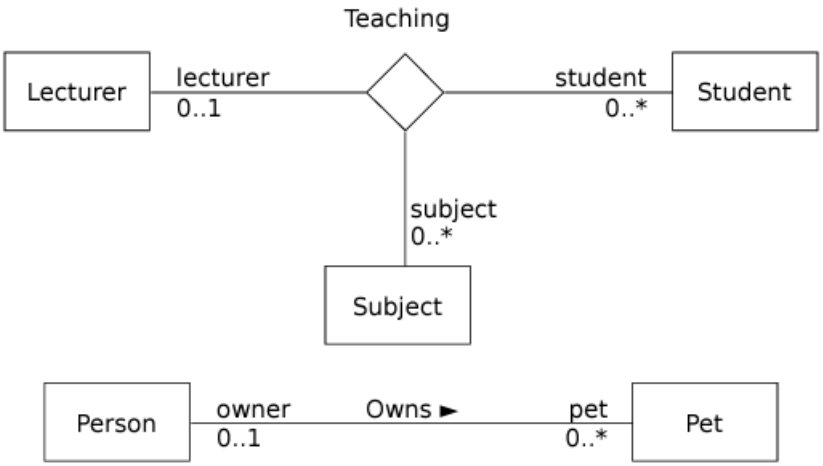
- Bármely asszociáció ábrázolható egy csúcsára állított rombusszal, melyet minden egyes vég esetén egy folytonos vonal köt össze azzal az osztályozóval, mely a vég típusa. Kettőnél több végű asszociáció csak így ábrázolható.
- Egy bináris asszociációt általában két osztályozót összekötő folytonos vonal ábrázol, vagy egy osztályozót önmagával összekötő folytonos vonal.

Az asszociáció szimbólumához megadható név (ne legyen túl közel egyik véghez sem).



Asszociáció vég: az asszociációt ábrázoló vonal és egy osztályozót ábrázoló ikon (gyakran egy doboz) kapcsolata.

Egy asszociáció vég számosságának jelentése: A példányok számát adja meg a végen arra az esetre, amikor a többi (n - 1) vég mindegyikén egyegy értéket rögzítünk.



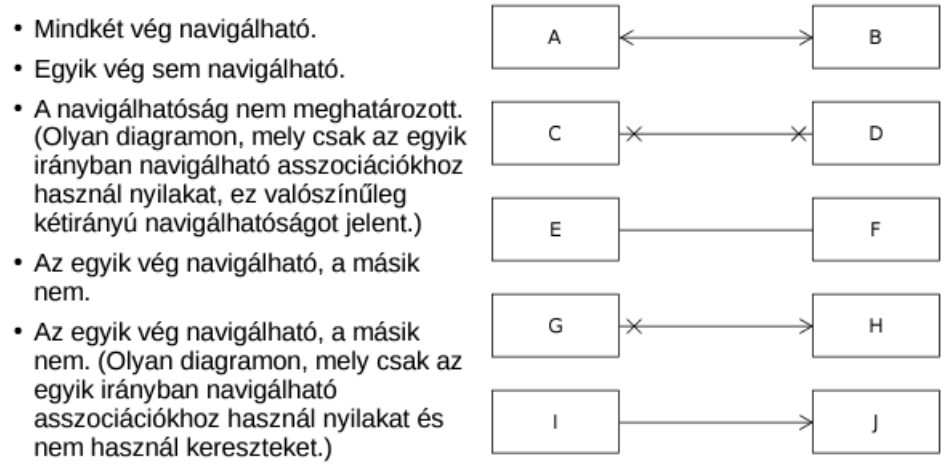
Az osztályozó és a vonal érintkezési pontjában elhelyezhető egy kis tömör kör (a továbbiakban pontnak nevezzük).

- A pont azt mutatja, hogy a modell tartalmaz egy tulajdonságot, melynek típusát a pont által érintett osztályozó ábrázolja. Ez a tulajdonság a másik végen lévő osztályozóhoz tartozik. Ebben az esetben szokás a tulajdonságot elhagyni az osztályozó attribútum rekeszéből.
- A pont hiánya azt jelzi, hogy a vég magához az asszociációhoz tartozik.

A navigálhatóság azt jelenti, hogy a kapcsolatokban résztvevő példányok futásidőben hatékonyan érhetők el az asszociáció többi végén lévő példányokból.

- Implementáció-specifikus azt a mechanizmus, mely révén hatékony elérés történik.

- Az osztályokhoz tartozó asszociációvégek mindig navigálhatók, az asszociációkhoz tartozók lehetnek navigálhatók és nem navigálhatók.



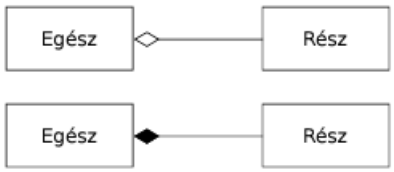
14. Egész-rész kapcsolat

A bináris asszociációk egész-rész kapcsolatot kifejező fajtái:

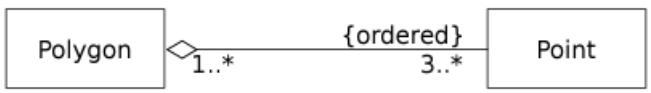
- Aggregáció (shared aggregation, aggregation): Egy rész objektum egyidejűleg több aggregációs objektumhoz is tartozhat, a részek és az aggregációs objektum egymástól függetlenül is létezhetnek.
- Kompozíció (composite aggregation, composition): Az aggregáció erősebb formája. Egy rész objektum legfeljebb egy kompozit objektumhoz tartozhat. A kompozit objektum törlésekor az összes rész objektum vele együtt törlődik.

Egy bináris asszociáció egyik vége jelölhető meg csak aggregációként vagy kompozícióként.

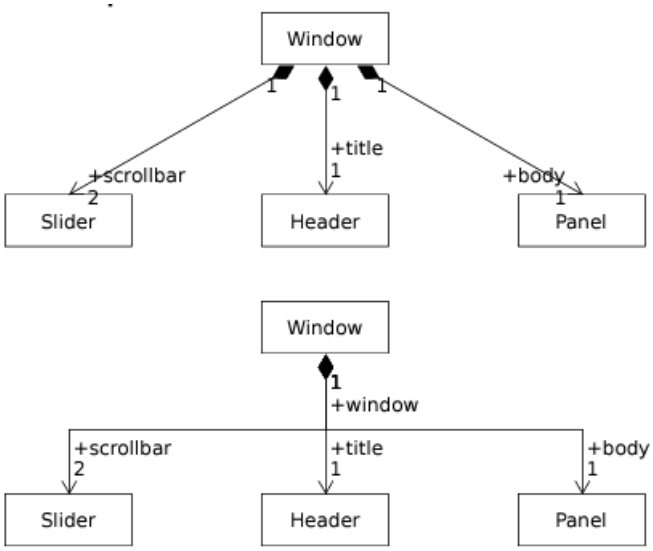
Jelölésmód:



Példa aggregációra:



Példa kompozícióra:



15. Általánosítás

Az általánosítás egy általánosítás/specializáció kapcsolatot határoz meg osztályozók között. Egy speciális osztályozót kapcsol össze egy általánosabb osztályozóval.

Az általánosítás/specializáció reláció tranzitív lezártja szerint értelmezzük egy osztályozó általánosításait és specializációit.

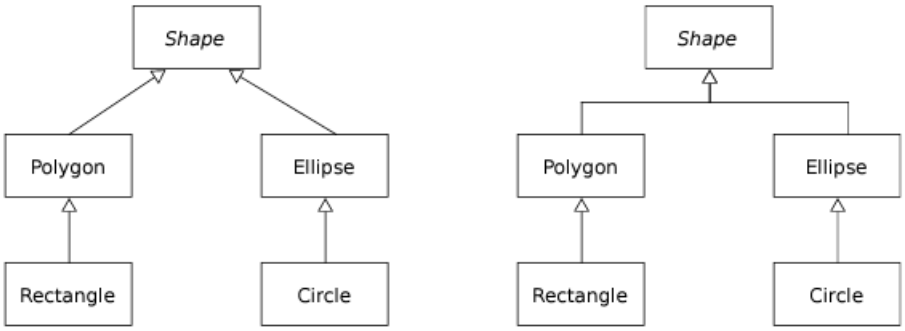
A közvetlen általánosításokat a speciális osztályozó szülőjének nevezzük, osztályok esetén ősosztálynak.

Egy osztályozó egy példánya minden általánosításának példánya.

A speciális osztályozó örökli az általános osztályozó bizonyos tagjait.

Jelölésmód:

Példa:

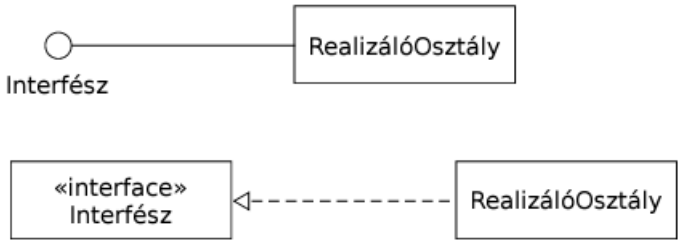


16. Interfészek

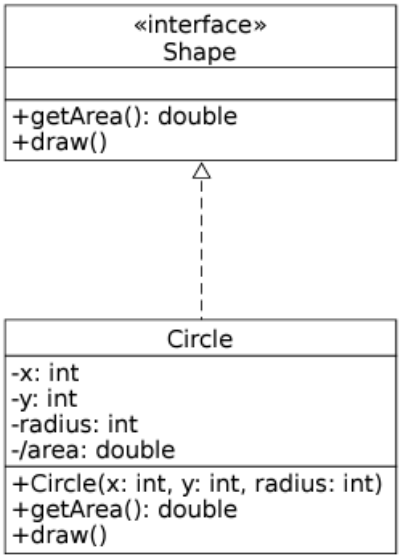
Az interfész egy olyan fajta osztályozó, mely nyilvános jellemzőket és kötelezettségeket deklarál, melyek együtt egy koherens szolgáltatást alkotnak. Az interfész egy szerződést határoz meg, az interfészt realizáló bármely osztályozó eleget kell, hogy tegyen a szerződésnek.

Az interfészek nem példányosíthatók. Osztályozók implementálják vagy realizálják az interfész specifikációt, mely azt jelenti, hogy az interfész specifikációnak megfelelő nyilvános felületet nyújtanak.

Jelölésmód:



Példa:



17. UML osztálydiagramok olvasása és értelmezése

EZ NEM TUDOM HOL A GECIBE VAN.

II. SZOFTVERTESZTELÉS

1. Mi a szoftvertesztelés?

IEEE: A szoftvertesztelés annak dinamikus verifikálását jelenti, hogy egy program várt módon viselkedik tesztesetek egy véges halmazán, melyek alkalmas módon kerülnek kiválasztásra egy általában végtelen végrehajtási tartományból.

ISTQB: A szoftvertesztelés egy megoldás a szoftver minőségének megállapításához és a szoftver működés közbeni meghibásodási kockázatának csökkentésére.

Sommerville: A tesztelés célja a használatba vétel előtt annak megmutatása, hogy egy szoftver azt csinálja, amit kell, valamint a programhibák felfedezése.

A szoftvertesztelés egy tágabb folyamat, a szoftver verifikáció és validáció (V&V) része.

2. Verifikáció és validáció fogalma

Verifikáció (verification): annak ellenőrzése, hogy a szoftver megfelel-e a vele szemben támasztott (funkcionális és nemfunkcionális) követelményeknek.

Are we building the product right?

Validáció (validation): annak ellenőrzése, hogy a szoftver megfelel-e az ügyefelek elvárásainak.

Are we building the right product?

3. Hibát leíró szakkifejezések: tévedés/tévesztés, hiba, meghibásodás

A szoftverfejlesztésben számos szakkifejezést használnak hibák leírására:

Tévedés/tévesztés (error/mistake): rossz eredményt adó emberi tevékenység.

Hiba (defect/fault/bug): tökéletlenség vagy hiányosság egy munkatermékben, melynél nem teljesülnek a követelmények vagy előírások.

Meghibásodás (failure): olyan esemény, melynél egy komponens vagy rendszer nem lát egy meg követelt funkciót a megszabott határok között.

Egy személy egy tévedést/tévesztést követ el, mely egy hibát vezethet be a szoftver kódjába vagy valamely más kapcsolódó munkatermékbe.

Ha végrehajtásra kerül a hiba a kódban, akkor az egy meghibásodást okozhat, de nem szükségszerűen minden esetben.

4. Tesztelési alapelvek

A szoftvertesztelés hét alapelve:

1. A tesztelés a hibák jelenlétét mutatja meg, nem a hiányukat
2. Lehetetlen a kimerítő tesztelés
3. A korai tesztelés időt és pénzt takarít meg
4. A hibák csoportosulnak
5. Óvakodj a kártevőírtó paradoxontól
6. A tesztelés környezetfüggő
7. A hibamentesség egy tévhit

5. Teszteset és tesztadat fogalma

Teszteset:

- Magas szintű teszteset: Teszteset, mely absztrakt előfeltételekkel, bementi adatokkal, elvárt eredményekkel, utófeltételekkel és (adott esetben) lépésekkel rendelkezik.

- Alacsony szintű teszteset: Teszteset, mely konkrét előfeltételekkel, bemeneti adatokkal, elvárt eredményekkel, utófeltételekkel és (adott esetben) a lépések részletes leírásával rendelkezik.

Tesztadat: A tesztadatok a teszt végrehajtáshoz szükséges adatokat jelentik. Az ilyen konkrét értékek a használatukra vonatkozó világos útmutatásokkal együtt végrehajtható alacsony szintű tesztesetekké teszik a magasszintű teszteseteket. Ugyanaz a magas szintű teszteset különböző tesztadatokat használhat különböző végrehajtásoknál.

6. Tesztelési szintek: egységtesztelés, integrációs tesztelés, rendszertesztelés, elfogadási tesztelés (alfa és béta tesztelés)

Egységtesztelés:

- A függetlenül tesztelhető komponensekre összpontosít.
- Az egységtesztelés általában az a fejlesztő végzi, aki a kódot írja, de legalább a tesztelt kódhoz való hozzáférés szükséges.
- A fejlesztők gyakran egy komponens kódjának megírása után írnak és hajtanak végre egységteszteket, azonban az automatikus egységtesztek megírása megelőzheti az alkalmazáskód megírását, lásd például a tesztvezérelt fejlesztést.

Integrációs tesztelés: Komponensek vagy rendszerek közötti kommunikációra összpontosít. Az integrációs tesztnek magára az integrációra kell koncentrálnia, nem pedig az egyes komponensek/rendszerek működésére.

Két különböző szintje van:

- *Komponens integrációs tesztelés:* az integrált komponensek közötti kommunikációra és interfészekre összpontosít. Az egységtesztelés után végzik és általában automatizált. A komponens integrációs tesztelés gyakran a fejlesztők felelősége.
- *Rendszerintegrációs tesztelés:* rendszerek közötti kommunikációra és interfészekre összpontosít. Kiterjedhet külső szervezetekkel és általuk szolgáltatott interfészekkel (például webszolgáltatásokkal) való interakciókra. Történhet a rendszertesztelés után vagy a folyamatban lévő rendszertesztelési tevékenységekkel párhuzamosan. A rendszerintegrációs tesztelés általában a tesztelők felelősége.

Rendszertesztelés: A rendszer egészének (funkcionális és nem funkcionális) viselkedésére összpontosít. Jellemzően független tesztelők végzik jelentős mértékben specifikációkra támaszkodva.

Elfogadási tesztelés:

- Annak meghatározására összpontosít, hogy a rendszer kész-e a telepítésre és az ügyfél (végfelhasználó) általi használatra.
- Gyakran az ügyfél vagy a rendszerüzemeltetők felelőssége, de más érintettek is bevonhatók.
- A szoftver kiadása előtt azt néha odaadják potenciális felhasználók egy kis kiválasztott csoportjának kipróbálásra (alfa tesztelés) és/vagy reprezentatív felhasználók egy nagyobb halmazának (béta tesztelés).

Alfa tesztelés:

- Felhasználók és fejlesztők együtt dolgoznak egy rendszer tesztelésén a fejlesztés közben.
- A fejlesztő szervezet telephelyén történik.

Béta tesztelés:

- Akkor történik, amikor egy szoftverrendszer egy korai, néha befejezetlen kiadást elérhetővé tesik kipróbálásra az ügyfelek és felhasználók egy nagyobb csoportjának.
- A felhasználók helyén történik.
- Főleg olyan szoftvertermékekhez alkalmazzák, melyeket sok különböző környezetben használnak.
- A marketing egyik formája is.

7. Teszttípusok

A tesztelés átfogó célja szerint az alábbi teszttípusokat különböztetjük meg:

- Funkcionális tesztelés
- Nem funkcionális tesztelés
- Fehér dobozos tesztelés
- Változással kapcsolatos tesztelés
  - Megerősítő tesztelés (confirmation testing)
  - Regressziós tesztelés (regression testing)

Bármely teszttípus alkalmazható bármely tesztelési szinten.

Funkcionális tesztelés: A rendszer által nyújtott funkciók tesztelése. Más szóval annak tesztelése, amit a rendszer csinál. Funkcionális teszteket minden tesztelési szinten ajánlott végezni.

Nem funkcionális tesztelés: Rendszerek olyan jellemzőinek értékelése, mint például a használhatóság, teljesítmény vagy biztonság. Más szóval annak tesztelése, hogy a rendszer mennyire jól teszi a dolgát.

Fehér dobozos tesztelés: A rendszer belső felépítésén vagy megvalósításán alapuló tesztek. A belső szerkezetbe beleérthető kód, architektúra vagy a rendszeren belüli munkafolyamatok.

Változással kapcsolatos tesztelés: Teszteket kell végezni, amikor módosítások történnek egy rendszerben egy hiba kijavításához vagy új funkcionalitás hozzáadásához/létező funkcionalitás módosításához. Két fajtája van:

- *Megerősítő tesztelés:* célja annak megerősítse, hogy az eredeti hiba sikeresen kijavításra került.
- *Regressziós tesztelés:* Lehetséges, hogy egy változás a kód egy részében, akár egy javítás vagy másfajta módosítás, véletlenül hatással van a kód más részeinek viselkedésére. A regressziós tesztelés célja a változások által okozott akaratlan mellékhatások érzékelése.

8. A jó egységtesztek ismertetőjegyei: FIRST

A jó egységtesztek ismertetőjegyei: FIRST

- Gyors (Fast): A tesztek gyorsak kell, hogy legyenek. Gyorsan kell, hogy lefussanak.
- Független (Independent): A tesztek nem függhetnek egymástól.

- Megismételhető (Repeatable): A tesztek bármely környezetben megismételhetők kell, hogy legyenek.
- Önérvényesítő (Self-Validating): A tesztnek logikai kimenete kell, hogy legyen. Vagy átmennek, vagy megbuknak.
- Jól időzített (Timely): A teszteket kellő időben kell megírni, közvetlenül a tesztelendő kód előtt.

9. Egységtesztek szervezése: az AAA minta

A tesztnek az alábbi három része ajánlott, hogy legyen:

- Elrendez (Arrange): ez a rész felelős a tesztelt rendszer és függőségei egy kívánt állapotba állításáért.
- Cselekszik (Act): ez a rész szolgál a tesztelt rendszer metódusainak meghívására, az előkészített függőségek átadására és a kimeneti érték elkapására (ha van).
- Kijelent (Assert): ez a szakasz szolgál a kimenetel ellenőrzésére. A kimenetel ábrázolható a visszatérési értékkel vagy a tesztelt rendszer végső állapotával.

10. Junit: tesztosztályok és tesztmetódusok, teszt végrehajtási életciklus, teszteredmények

Tesztosztály: bármely felsőszintű osztály, statikus tagosztály vagy @Nested osztály, mely legalább egy tesztmetódust tartalmaz. Nem lehet absztrakt és egyetlen konstruktora kell, hogy legyen.

Tesztmetódus: a @Test, @RepeatedTest, @ParameterizedTest, @TestFactory vagy @TestTemplate annotációval megjelölt bármely példánymetódus.

Életciklus metódus: a @BeforeAll, @AfterAll, @BeforeEach vagy @AfterEach annotációval megjelölt bármely metódus.

Nem szükséges, hogy a tesztosztályok, tesztmetódusok és életciklus metódusok nyilvánosak legyenek, de nem lehetnek privát láthatóságúak.

A tesztmetódusok és életciklus metódusok deklarálhatók az aktuális tesztosztályon belül lokálisan, örökölhetők ősosztályból vagy interfészektől. Nem lehetnek absztraktak és nem adhatnak vissza értéket.

A tesztosztály konstruktoroknak és metódusoknak is meg van engedve, hogy paramétereik legyenek, mely lehetővé teszi a függőség befecskendezést.

Teszt végrehajtási életciklus: Alapértelmezésben a JUnit egy új példányt hoz létre minden egyes tesztosztályból az egyes tesztmetódusok végrehajtás előtt, mely lehetővé teszi a tesztmetódusok izoláltan történő végrehajtását. Ez a viselkedés megváltoztatható, az összes tesztmetódus ugyanazon a teszt példányon történő végrehajtásához a tesztosztályt a @TestInstance(Lifecycle.PER_CLASS) annotációval kell megjelölni.

Teszteredmények:

- *Siker (Success):* amikor a teszt végrehajtásakor minden tényleges eredmény megegyezik a várt végeredménnyel. Ekkor azt mondjuk a teszt átmegy (passes).
- *Bukás (Failure):* amikor a teszt végrehajtásakor a tényleges eredmény nem egyezik meg a várt eredménnyel. A bukást egy elbukó okozza. Ekkor azt mondjuk, hogy a teszt megbukik (fails).

- *Hiba (Error):* amikor a teszt végrehajtásakor egy hiba következik be, mely megakadályozza a befejeződést. A hibát egy váratlan kivétel vagy hiba okozza.

11. Kódlefedettségi metrikák: utasítás lefedettség/sor lefedettség, ág lefedettség, mi az az ésszerű lefedettségi szám?

Utasítás lefedettség/sor lefedettség: A leggyakrabban használt lefedettségi metrikák az utasítás lefedettség (statement coverage) és a sorlefedettség (line coverage).

- Utasítás lefedettség = Végrehajtott utasítások/Összes utasítás száma
- Sorlefedettség = Végrehajtott kódsorok/Összes sorszáma

Minden egyes végrehajtott utasítást/sort egyszer számolunk.

A sorlefedettség meghatározásakor csak a végrehajtható kódot tartalmazó sorok kerülnek számolásra.

Vegyük észre, hogy a sorlefedettség függ a forráskód formázástól.

Minél tömörebb a kód, annál jobb az utasítás/sorlefedettség, mivel az utasítások/sorok nyers számán alapul.

Ág lefedettség: Az ág lefedettség (branch coverage) egy lefedettségi mérték, mely az olyan vezérlési szerkezeteken alapul, mint az if és a switch. A végrehajtott ágak arányát méri egy tesztkészlet futtatásakor az összes ág számához viszonyítva.

Az ág lefedettség kiszámítása: Ág lefedettség = Végrehajtott ágak/Összes ág száma

Ésszerű lefedettségi szám: Veszélyes egy bizonyos érték elérésének megcélzása egy lefedettségi metrikánál, mivel könnyen ez válhat a főcéllá. Inkább a megfelelő egységtesztelésre kell koncentrálni.

Ökölszabályok:

- Jó, ha egy rendszer főrészeinél nagy a lefedettség.
- Nem jó ezt magasszintű követelménnyé tenni.

III. OBJEKTUMORIENTÁLT TERVEZÉSI ALAPELVEK

1. Statikus kódelemzés fogalma, példák statikus kódelemző eszközökre

A statikus kódelemzés (static code analysis) a programkód elemzésének folyamata, mely a kód végrehajtása nélkül történik. Az elemezés irányulhat: hibák észlelésére; annak ellenőrzésére, hogy a kód megfelel-e egy kódolási szabványnak, …

Statikus kódelemző (eszköz) (static code analyzer, static code analysis tool): statikus kódelemzést végző automatikus eszköz.

Példák statikus kódelemző eszközökre:

- C#: InferSharp, Roslyn Analyzers, Roslynator
- C++: Cppcheck
- ECMAScript/Javascript: ESLint, JSHint, JSLint, RSLint
- Java: Checkstyle, Error Prone, NullAway, SpotBugs
- Python: Prospector

- Több nyelvet támogató eszközök: Coala, Infer, PMD, Semgrep

2. A DRY elv

Ne ismételd magad (**D**on't **R**epeat **Y**ourself).

„Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.”

Az ellenkezője a WET.

„We enjoy typing”, „write everything twice”, „waste everyone's time”, …

Az ismétlések fajtái:

- *Kényszerített ismétlés (imposed duplication)*: a fejlesztők úgy érzik, hogy nincs választásuk, a környezet láthatólag megköveteli az ismétlést.
- *Nem szándékos ismétlés (inadvertent duplication):* a fejlesztők nem veszik észre, hogy információkat duplikálnak.
- *Türelmetlen ismétlés (impatient duplication):* a fejlesztők lustaságából fakad, az ismétlés látszik a könnyebb útnak.
- *Fejlesztők közötti ismétlés (interdeveloper duplication):* egy csapatban vagy különböző csapatokban többen duplikálnak egy információt.

Kódismétlés: A kódismétlés (duplicate code) azonos (vagy nagyon hasonló) forráskódrész, mely egynél többször fordul elő egy programban.

3. A KISS elv

Keep **i**t **s**imple, **s**tupid.

- 1960-as évek, amerikai haditengerészet.
- Kelly Johnson (1910–1990) repülőmérnöknek tulajdonítják a kifejezést.

4. A YAGNI elv

You **A**ren't **G**onna **N**eed **I**t.

Az extrém programozás (XP) egy alapelve.

„Mindig akkor implementálj valamit, amikor tényleg szükséged van rá, soha ne akkor, amikor csak sejted, hogy kell.”

A YAGNI alapelv csak azon képességekre vonatkozik, melyek egy feltételezett lehetőség támogatásához kerülnek beépítésre a szoftverbe, nem vonatkozik a szoftver módosítását könnyítő törekvésekre.

A YAGNI csak akkor járható stratégia, ha a kód könnyen változtatható.

5. Csatoltság, laza és szoros csatoltság

Csatoltság (coupling): egy szoftvermodul függésének mértéke egy másik szoftvermodultól.

- Más szóval, a szoftvermodulok közötti csatoltság annak mértéke, hogy mennyire szoros a kapcsolatuk.
- A csatoltság laza vagy szoros lehet.

Szoros csatoltság:

- A bonyolultságot növeli, mely megnehezíti a kód módosítását, tehát a karbantarthatóságot csökkenti.
- Az újrafelhasználhatóságot is csökkenti.

Laza csatoltság:

- Lehetővé teszi a fejlesztők számára a nyitva zárt elvnek megfelelő kód írását, azaz a kódot kiterjeszthetővé teszi.
- Kiterjeszthetővé teszi a kódot, a kiterjeszthetőség pedig karbantarthatóvá.
- Lehetővé teszi a párhuzamos fejlesztést.

6. GoF alapelvek

Interfészre programozzunk, ne implementációra!

Részesítsük előnyben az objektum-összetételt az öröklődéssel szemben!

A két leggyakoribb módszer az újrafelhasználásra az objektumorientált rendszerekben:

- Öröklődés (fehér dobozos újrafelhasználás)
- Objektum-összetétel (fekete dobozos újrafelhasználás)

A fehér/fekete dobozos jelző a láthatóságra utal.

Az öröklődés előnyei:

- Statikusan, fordítási időben történik, és használata egyszerű, mivel a programozási nyelv közvetlenül támogatja.
- Az öröklődés továbbá könnyebbé teszi az újrafelhasznált megvalósítás módosítását is.

Az öröklődés hátrányai:

- Először is, a szülőosztályoktól örökölt megvalósításokat futásidőben nem változtathatjuk meg, mivel az öröklődés már fordításkor eldől.
- Másodszor – és ez általában rosszabb –, a szülőosztályok gyakran alosztályaik fizikai ábrázolását is meghatározzák, legalább részben.
- Az implementációs függőségek gondot okozhatnak az alosztályok újrafelhasználásánál.

Objektum-összetétel:

- Az objektum-összetétel dinamikusan, futásidőben történik, olyan objektumokon keresztül, amelyek hivatkozásokat szereznek más objektumokra.
- Az összetételhez szükséges, hogy az objektumok figyelembe vegyék egymás interfészét, amihez gondosan megtervezett interfészek kellenek, amelyek lehetővé teszik, hogy az objektumokat sok másikkal együtt használjuk.

Az objektum összetétel előnyei:

- Mivel az objektumokat csak az interfészükön keresztül érhetjük el, nem szegjük meg az egységbe zárás elvét.
- Bármely objektumot lecserélhetünk egy másikra futásidőben, amíg a típusaik egyeznek.
- Az öröklődéssel szemben segít az osztályok egységbe zárásában és abban, hogy azok egy feladatra összpontosíthassanak.
- Az osztályok és osztályhierarchiák kicsik maradnak, és kevésbé valószínű, hogy kezelhetetlen szörnyekké duzzadnak.

Az objektum összetétel hátrányai:

- Másrészről az objektum-összetételen alapuló tervezés alkalmazása során több objektumunk lesz (még ha osztályunk kevesebb is), és a rendszer viselkedése ezek kapcsolataitól függ majd, nem pedig egyetlen osztály határozza meg.

7. SOLID alapelvek: egyszeres felelősség elve, nyitva zárt elv, Liskov-féle helyettesítési elv, interfész szétválasztási elv, függőség megfordítási elv

- **S**ingle Responsibility Principle (SRP) – Egyszeres felelősség elve
- **O**pen/Closed Principle (OCP) – Nyitva zárt elv
- **L**iskov Substitution Principle (LSP) – Liskov-féle helyettesítési elv
- **I**nterface Segregation Principle (ISP) – Interfész szétválasztási elv
- **D**ependency Inversion Principle (DIP) – Függőség megfordítási elv

Egyszeres felelősség elve:

- Egy felelősség egy ok a változásra.
- Minden felelősség a változás egy tengelye. Amikor a követelmények változnak, a változás a felelősségben történő változásként nyilvánul meg.
- Ha egy osztálynak egynél több felelőssége van, akkor egynél több oka van 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.
- A szoftverek aktorok kielégítése céljából változnak. Az „aktor” kifejezést itt emberek (például felhasználók) egy olyan csoportjára használjuk, akik azt akarják, hogy a szoftver ugyanúgy változzon.
- Az elv tehet így fogalmazható újra: Egy modul egy, és csak egyetlen aktornak van alárendelve.

Nyitva zárt elv:

- Bertrand Meyer által megfogalmazott alapelv.
- A szoftver entitások (osztályok, modulok, függvények, …) legyenek nyitottak a bővítésre, de zártak a módosításra.
- Kapcsolódó tervezési minták: gyártó metódus, helyettes, stratégia, sablonfüggvény, látogató.
- Az elvnek megfelelő modulnak két fő jellemzője van:
  - Nyitott a bővítésre: azt jelenti, hogy a modul viselkedése kiterjeszthető.
  - Zárt a módosításra: azt jelenti, hogy a modul viselkedésének kiterjesztése nem eredményezi a modul forrás- vagy bináris kódjának változását.

Liskov-féle helyettesítési elv:

- Barbara Liskov által megfogalmazott 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.

Interfész szétválasztási elv:

- Robert C. Martin által megfogalmazott elv.
- *„Classes should not be forced to depend on methods they do not use.”*
- Vastag interfész (fat interface) (Bjarne Stroustrup): *„An interface with more member functions and friends than are logically necessary.”*

- Az interfész szétválasztási elv a vastag interfészekkel foglalkozik.
- A vastag interfészekkel rendelkező osztályok interfészei nem koherensek, melyekben a metódusokat olyan csoportokra lehet felosztani, melyek különböző klienseket szolgálnak ki.
- Az ISP elismeri azt, hogy vannak olyan objektumok, melyekhez nem koherens interfészek szükségesek, de azt javasolja, hogy a kliensek ne egyetlen osztályként ismerjék őket.
- *Interfész szennyezés (interface pollution):* Egy interfész szennyezése szükségtelen metódusokkal.
- Amikor egy kliens egy olyan osztálytól függ, melynek vannak olyan metódusai, melyeket a kliens nem használ, más kliensek azonban igen, akkor a többi kliens által az osztályra kényszerített változások hatással lesznek arra a kliense is.
- Ez a kliensek közötti nem szándékos csatoltságot eredményez.

Függőség megfordítási elv:

- Robert C. Martin által megfogalmazott elv.
- Magas szintű modulok ne függjenek alacsony szintű moduloktól. Mindkettő absztrakcióktól függjön.
- Az absztrakciók ne függjenek a részletektől. A részletek függjenek az absztrakcióktól.
- Az elnevezés onnan jön, hogy a hagyományos szoftverfejlesztési módszerek hajlamosak olyan felépítésű szoftvereket létrehozni, melyekben a magas szintű modulok függenek az alacsony szintű moduloktól.
- Kapcsolódó tervezési minta: illesztő
- A magas szintű modulok tartalmazzák az alkalmazás üzleti logikáját, ők adják az alkalmazás identitását. Ha ezek a modulok alacsony szintű moduloktól függenek, akkor az alacsony szintű modulokban történő változásoknak közvetlen hatása lehet a magas szintű modulokra, szükségessé tehetik azok változását is.
- Ez abszurd! A magas szintű modulok azok, melyek meg kellene, hogy határozzák az alacsony szintű modulokat.
- A magas szintű modulokat szeretnénk újrafelhasználni. Az alacsony szintű modulok újrafelhasználására elég jó megoldást jelentenek a programkönyvtárak.
- Ha magas szintű modulok alacsony szintű moduloktól függenek, akkor nagyon nehéz az újrafelhasználásuk különféle helyzetekben.
- Ha azonban a magas szintű modulok függetlenek az alacsony szintű moduloktól, akkor elég egyszerűen újrafelhasználhatók.
- Minden egyes magasabb szintű interfész deklarál az általa igényelt szolgáltatásokhoz egy interfészt.
- Az alacsonyabb szintű rétegek realizálása ezekből az interfészekből történik.
- Ilyen módon a felsőbb rétegek nem függenek az alsóbb rétegektől, hanem pont fordítva.
- Függés absztrakcióktól:
  - Ne függjön a program konkrét osztályoktól, hanem inkább csak absztrakt osztályoktól és interfészektől.
  - Egyetlen változó se hivatkozzon konkrét osztályra.
  - Egyetlen osztály se származzon konkrét osztályból.
  - Egyetlen metódus se írjon felül valamely ősosztályában implementált metódust.
  - A fenti heurisztikát a legtöbb program legalább egyszer megsérti.

- ❖ Nem túl gyakran változó konkrét osztályok esetén (például String) megengedhető a függés.

8. Függőség befecskendezés

A függőség befecskendezés (DI – dependency injection) kifejezés Martin Fowlertől származik.

A vezérlés megfordítása (IoC – inversion of control) nevű architekturális minta alkalmazásának egy speciális esete.

Definíció (Seemann): *„Dependency Injection is a set of software design principles and patterns that enable us to develop loosely coupled code."*

A lazán csatoltság kiterjeszthetővé teszi a kódot, a kiterjeszthetőség pedig karbantarthatóvá.

Egy objektumra egy olyan szolgáltatásként tekintünk, melyet más objektumok kliensként használnak.

Az objektumok közötti kliens-szolgáltató kapcsolatot függésnek nevezzük. Ez a kapcsolat tranzitív.

Függőség (dependency): egy kliens által igényelt szolgáltatást jelent, mely a feladatának ellátásához szükséges.

Függő (dependent): egy kliens objektum, melynek egy függőségre vagy függőségekre van szüksége a feladatának ellátásához.

Objektum gráf (object graph): függő objektumok és függőségeik egy összessége.

Befecskendezés (injection): egy kliens függőségeinek megadását jelenti.

DI konténer (DI container): függőség befecskendezési funkcionalitást nyújtó programkönyvtár. Az Inversion of Control (IoC) container kifejezést is használják rájuk.

A függőség befecskendezés alkalmazható DI konténer nélkül.

Tiszta DI: függőség befecskendezés alkalmazásának gyakorlata DI konténer nélkül.

A függőség befecskendezés objektum gráfok hatékony létrehozásával, ennek mintáival és legjobb gyakorlataival foglalkozik.

A DI keretrendszerek lehetővé teszik, hogy a kliensek a függőségeik létrehozását és azok befecskendezését külső kódra bízzák.

A függőség befecskendezés előnyei:

- Kiterjeszthetőség
- Karbantarthatóság
- Tesztelhetőség: a függőség befecskendezés támogatja az egységtesztelést.

IV. MINTÁK A SZOFTVERFEJLESZTÉSBEN

1. Mi a minta?

„Minden minta olyan problémát ír le, ami újra és újra felbukkan a környezetünkben, s aztán leírja hozzá a megoldás magját, oly módon, hogy a megoldás milliószor

felhasználható legyen, anélkül, hogy valaha is kétszer ugyanúgy csinálnánk.” – Christopher Alexander

„Minden minta egy három részből álló szabály, mely egy bizonyos környezet, egy probléma és egy megoldás közötti kapcsolatot fejez ki.” – Christopher Alexander

„A minta egy olyan ötlet, mely egy gyakorlati környezetben már hasznosnak bizonyult, és várhatóan más környezetekben is hasznos lesz.” - Martin Fowler

„A minta egy gyakori probléma vagy kérdés általános megoldásának leírása, melyből meghatározható egy konkrét probléma részletes megoldása.” - Scott W. Ambler

2. Architekturális minták, a modell-nézet vezérlő (MVC) architekturális minta

Az architekturális minták szoftverrendszerek alapvető szerkezeti felépítésére adnak sémákat. Ehhez előre definiált alrendszereket biztosítanak, meghatározzák ezek felelősségi köreit, valamint szabályokat és irányelveket tartalmaznak a köztük lévő kapcsolatok szervezésére vonatkozólag.

MVC: 1978-ban Trygve Reenskaug, egy norvég számítógéptudós (Oslói Egyetem) dolgozta ki.

- A modell elválasztása a nézet komponenstől több nézetet is lehetővé tesz ugyanahhoz a modellhez.
- A nézet elválasztása a vezérlő komponenstől kevésbé fontos.
- A modell a szakterületeit valamilyen információját ábrázoló objektum, mely adatokat csomagol be.
    - ❖ Rendelkezik alkalmazás-specifikus feldolgozást végző eljárásokkal, melyeket a vezérlők hívnak meg a felhasználó nevében.
    - ❖ Függvényeket biztosít az adatokhoz való hozzáféréshez, melyeket a nézetek használnak a megjelenítendő adatok eléréséhez.
    - ❖ Regisztrálja a függő objektumokat (nézeteket és vezérlőket), melyeket értesít az adatokban történő változásokról.
- Változatok:
    - ❖ Hierarchikus modell-nézet-vezérlő (HMVC)
    - ❖ Model-view-presenter (MVP)
    - ❖ Model-view-viewmodel (MVVM)

3. Tervezési minták, tervezési minta osztályozása

A tervezési minták középszintű minták, kisebb léptékűek az architekturális mintáknál.

Alkalmazásuknak nincs hatása egy szoftverrendszer alapvető felépítésére, de nagyban meghatározhatják egy alrendszer felépítését.

Függetlenek egy adott programozási nyelvtől vagy programozási paradigmától.

GoF: A tervezési minták egymással együttműködő objektumok és osztályok leírásai, amelyek testre- szabott formában valamilyen általános tervezési problémát oldanak meg egy bizonyos összefüggésben.

A tervezési minták osztályozása céljuk szerint (GoF):

- *Létrehozási minták (creational patterns):* az objektumok létrehozásával foglalkoznak.
- *Szerkezeti minták (structural patterns):* azzal foglalkoznak, hogy hogyan alkotnak osztályok és objektumok nagyobb szerkezeteket.
- *Viselkedési minták (behavioral patterns):* az osztályok vagy objektumok egymásra hatását, valamint a felelősségek elosztását írják le.

Létrehozási minták:

- *Építő:* Az összetett objektumok felépítését függetleníti az ábrázolásuktól, így ugyanazzal az építési folyamattal különböző ábrázolásokat hozhatunk létre.
- *Egyke:* Egy osztályból csak egy példányt engedélyez, és ehhez globális hozzáférési pontot ad meg.

Szerkezeti minták:

- *Díszítő:* Az objektumokhoz dinamikusan további felelősségi köröket rendel. A kiegészítő szolgáltatások biztosítása terén e módszer rugalmas alternatívája az alosztályok létrehozásának.
- *Illesztő:* Az adott osztály interfészét az ügyfelek által igényelt interfésszé alakítja. E módszerrel az egyébként összeférhetetlen interfészű osztályok együttműködését biztosíthatjuk.

Viselkedési minták:

- *Bejáró:* Az összetett objektumok elemeinek soros elérését a háttérben megbúvó ábrázolás felfedése nélkül biztosító módszer kialakítása.
- *Sablonfüggvény:* 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.
- *Megfigyelő:* Objektumok között egy sok-sok függőségi kapcsolatot létrehozni, így amikor az egyik objektum állapota megváltozik, minden tőle függő objektum értesül erről és automatikusan frissül.

4. Programozási idiómák/implementációs minták

Egy idióma egy programozási nyelvre jellemző alacsony szintű minta. Az idiómák jelentik a legalacsonyabb szintű mintákat.

Egy idióma leírja, hogy hogyan valósítsuk meg komponensek és kapcsolataik bizonyos vonatkozásait az adott nyelv eszközrendszerével.

A legtöbb idióma nyelvspecifikus, létező programozási tapasztalatot hordoznak.

5. Antiminták, a massza és a spagetti kód antiminta

Antiminták:

- Az antiminta kifejezést Andrew Koenig alkotta meg: *„Egy antiminta pont olyan, mint egy minta, kivéve azt, hogy megoldás helyet valami olyat ad, ami látszólag megoldásnak néz ki, de nem az.”*
- Egy problémára adott általánosan előforduló megoldások, melyek kifejezetten negatív következményekkel járnak.
- Bármely szinten megjelenhetnek.
- Nézőpont szerint az alábbi három kategória:

- ❖ Szoftverfejlesztési antiminták
- ❖ Szoftver architekturális antiminták
- ❖ Szoftverprojekt vezetési antiminták

A massza:

- **Antiminta neve:** a massza (The Blob)
- **Más néven:** Winnebago, az Isten osztály (The God Class)
- **Leggyakoribb előfordulási szint:** alkalmazás
- **Újragyártott megoldás neve:** a felelősségek újraosztása
- **Újragyártott megoldás típusa:** szoftver
- **Kiváltó okok:** lustaság, sietség
- **Kiegyensúlyozatlan erők:** a funkcionalitás, a teljesítmény és a bonyolultság kezelése
- **Anekdotaszerű példa:** „Ez az osztály az architektúránk szíve."
- **Általános alak:**
  - ❖ A massza olyan tervezésnél fordul elő, ahol a feldolgozást egyetlen osztály sajátítja ki magának, a többi osztály pedig elsősorban adatokat zár egységbe.
  - ❖ Olyan osztálydiagram jellemzi, mely egyetlen bonyolult vezérlő osztályból és azt körülvevő egyszerű adat osztályokból áll.
  - ❖ A massza általában procedurális tervezésű, habár reprezentálható objektumokkal és implementálható objektumorientált nyelven.
  - ❖ Gyakran iteratív fejlesztés eredménye, ahol egy megvalósíthatósági példakódot (proof–of–concept code) fejlesztenek idővel egy prototípussá, végül pedig egy éles rendszerré.
- **Tünetek és következmények:**
  - ❖ Egyetlen osztály nagyszámú attribútummal, művelettel vagy mindkettővel. Általában a massza jelenlétét jelzi egy 60-nál több attribútummal és művelettel rendelkező osztály.
  - ❖ Egymáshoz nem kapcsolódó attribútumok és műveletek bezárása egyetlen osztályba.
  - ❖ Egy ilyen osztály túl bonyolult újrafelhasználáshoz vagy teszteléshez.– Költséges lehet egy ilyen osztály a memóriába való betöltése. Még egyszerű műveletekhez is sok erőforrást használ.
- **Tipikus okok:**
  - ❖ Az objektumorientált architektúra hiánya.
  - ❖ (Bármilyen) architektúra hiánya.
  - ❖ Az architektúra kikényszerítésének hiánya.
  - ❖ Túl korlátozott beavatkozás.
  - ❖ Kódolt katasztrófa (rossz követelmény specifikáció).
- Ismert kivételek: A massza antiminta elfogadható kompatibilitási okokból megtartott korábbi rendszer becsomagolásakor.
- **Újragyártott megoldás:** A megoldás kódújraszervezéssel jár.
  - ❖ Összetartozó attribútumok és műveletek csoportjainak azonosítása.
  - ❖ Természetes helyet kell keresni ezen funkcionalitáscsoportok számára és oda kell áthelyezni őket.
  - ❖ A redundáns asszociációk eltávolítása.
- **Változatok:**
  - ❖ Viselkedési forma: osztály, mely tartalmaz egy központi folyamatot, mely interakcióban van a rendszer legtöbb más részével („központi agy osztály").

- Adat forma: osztály, mely tartalmaz olyan adatokat, melyeket a rendszer legtöbb más objektuma használ („globális adat osztály”).

- **Alkalmazhatóság más nézőpontokra és szintekre:** Az architekturális és a vezetési nézőpontnak is kulcsszerepe van a massza antiminta megelőzésében.
- **Alkalmazhatóság más nézőpontokra és szintekre:** Az architekturális és vezetői nézőpontok is kulcsszerepet játszanak a massza antiminta megelőzésében.

A spagetti kód:

- **Antiminta neve:** spagetti kód (Spaghetti Code)
- **Leggyakoribb előfordulási szint:** alkalmazás
- **Újragyártott megoldás neve:** kódújraszervezés, kódtisztítás
- **Újragyártott megoldás típusa:** szoftver
- **Kiváltó okok:** tudatlanság, lustaság
- **Kiegyensúlyozatlan erők:** a bonyolultság, a változás kezelése
- **Anekdotaszerű példa:** „Ó! Micsoda zűrzavar!”, „Ugye tisztában vagy vele, hogy a nyelv egynél több függvényt támogat?”, „Könnyebb újraírni ezt a kódot, mint megpróbálni módosítani.”
- **Háttér:** Klasszikus, a leghíresebb antiminta, mely egyidős a programozási nyelvekkel.
- **Általános alak:**
  - Strukturálatlan, nehezen átlátható programkódként jelenik meg.
  - Objektumorientált nyelvek esetén kevés osztály jellemzi, melyeknél a metódusok megvalósítása nagyon hosszú.
- **Tünetek és következmények:**
  - A metódusok nagyon folyamat-orientáltak, az objektumokat gyakran folyamatoknak nevezik.
  - A végrehajtást az objektum implementáció határozza meg, nem pedig az objektum kliensei.
  - Kevés kapcsolat van az objektumok között.
  - Sok a paraméter nélküli metódus, melyek osztályszintű és globális változókat használnak.
  - Nehéz a kód újrafelhasználása. Sok esetben nem is szempont az újrafelhasználhatóság.
  - Elvesznek az objektumorientáltság előnyei, nem kerül felhasználásra az öröklődés és a polimorfizmus.
  - A további karbantartási erőfeszítések csak súlyosbítják a problémát.– Költségesebb a létező kódbázis karbantartása, mint egy új megoldás kifejlesztése a semmiből.
- **Tipikus okok:**
  - Tapasztalatlanság az objektumorientált tervezés terén.
  - Nincs mentorálás, nem megfelelő a kódátvizsgálás.
  - Nincs az implementálást megelőző tervezés.
  - A fejlesztők elszigetelten dolgoznak.
- **Ismert kivételek:** Ésszerűen elfogadható, ha az interfészek következetesek és csak az implementáció spagetti.
- **Újragyártott megoldás:** A megoldás kódújraszervezés.
- **Kapcsolódó megoldások:** analízis-paralízis, lávafolyás

V. TISZTA KÓD

SZOFTVERFEJLESZTÉS II. ZH JEGYZET

1. Értelmes nevek

- Olyan neveket használjunk a kódban, melyekből kiderül a szándék.
- Rossz gyakorlat:
  - int d; // elapsed time in days
- Jó gyakorlat:
  - int elapsedTimeInDays;
  - int daysSinceCreation;
  - int daysSinceModification;
  - int fileAgeInDays;
- Kerüljük a félrevezető neveket. Félrevezető például az accountList név, ha nem ténylegesen egy listáról van szó. Ha ez a helyzet, sokkal jobb például az accountGroup vagy az accounts név.
- Értelmesen megkülönböztethető neveket használjunk.
- Ne használjunk olyan túl általános zajszavakat a nevekben, mint például Data, Info vagy Object.
  - A nevük alapján nem világos, hogy mi a különbség például a Product, ProductData és ProductInfo nevű osztályok között.
  - Soha ne használjuk a variable szót változó nevében.
- Használjunk kiejthető neveket.
- Használjunk kereshető neveket.
  - Például egybetűs nevek esetén problémás lehet a keresés.
  - Egybetűs neveket csak lokális változókhoz használjunk rövid metódusokban.
- Egy név hossza meg kell, hogy feleljen a hatásköre méretének.
  - Minél nagyobb a hatáskör mérete, célszerűbb annál hosszabb nevet választani.
  - Ha egy változót vagy konstanst a kódban több helyen is használunk, akkor érdemes neki kereshető nevet adni.
- Kerüljük a kódolásokat a nevekben, melyek a típusról vagy a hatáskörről szolgáltatnak információkat.
  - Magyar jelölés (Hungarian notation)
  - Tagok nevében m_ előtag használata
  - I karakter az interfészek nevének elején
- 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.
- Szükség esetén használjuk az adott problématerület neveit.
- Nevek kontextusba helyezése:
  - Vannak olyan nevek, melyek önmagukban értelmesek. A legtöbb azonban nem ilyen, ezeket az érthetőséghez az olvasó számára egy kontextusba kell helyezni, bezárva őket egy osztályba, függvénybe vagy névtérbe.
  - Nyilvánvaló például, hogy a street, city, zipCode, state nevű változók egy címet alkotnak.
  - Ha azonban csak a state változót látjuk egy metódusban, nem feltétlenül arra gondolunk, hogy ez egy cím része.
  - Ne vezessünk be azonban feleslegesen kontextusokat. Például ne használjunk osztályok nevének elején az alkalmazásra utaló előtagot.

2. Függvények

- A függvények nagyon rövidek kell, hogy legyenek.
- Nem szabad, hogy 100 sorosak legyenek. Nagyon ritkán legyenek 20 sorosak. Legyenek inkább 2–4 sorosak.
- Utasításblokkok (for, if, while, …) egyetlen sornyi kódot kell, hogy tartalmazzanak, mely várhatólag egy függvényhívás.
    - ❖ Ez nem csupán röviden tartja a befoglaló függvényt, hanem dokumentációs értékkel is bír, mivel a meghívott függvény neve beszédes.
    - ❖ Ez azt is jelenti, hogy a függvények bekezdési szintjeinek (indent level) száma nem lehet több 2-nél. Ez a függvényeket könnyen olvashatóvá és érthetővé teszi.
- A függvények csak egy dolgot csináljanak, de azt jól.
- Függvényenként egy absztrakciós szint:
    - ❖ Annak biztosításához, hogy a függvények egy dolgot csináljanak, az utasítások azonos absztrakciós szintűek kell, hogy legyenek bennük.
    - ❖ Stepdown szabály: felülről lefelé haladva csökkenjen a függvények absztrakciós szintje. A kód felülről lefelé haladva olvasható.

<u>Argumentumok:</u>

- *A függvények megkülönböztetése az argumentumok száma szerint:*
    - ❖ **Niladikus (niladic):** argumentum nélküli, ez az ideális
    - ❖ **Monadikus (monadic):** egyargumentumú
    - ❖ **Diadikus (diadic):** kétargumentumú
    - ❖ **Triadikus (triadic):** három argumentumú, lehetőleg kerülni kell
    - ❖ **Poliadikus (polyadic):** háromnál több argumentumú, soha ne használjuk
- Az olvasó számára megnehezítik a kód megértését.
- A tesztelést is bonyolítják.
- Rossz gyakorlat jelző argumentumok használata. Egy jelző argumentum egy olyan logikai típusú argumentum, melynek értékétől függ a függvény viselkedése. Egy ilyen argumentummal rendelkező függvény egynél több dolgot csinál: egyet akkor, ha az argumentum értéke igaz, egy másikat akkor, ha hamis.
- Kettőnél vagy háromnál több argumentum esetén bizonyosakat érdemes lehet becsomagolni egy osztályba.
- Változó argumentumszámú függvényekre lista argumentumú függvényekként tekinthetünk.
- Mellékhatásmentesség:
    - ❖ A függvények legyenek mellékhatásmentesek.
    - ❖ 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.
- Hibakódok visszaadása helyett részesítsük előnyben a kivételeket. Így egyszerű további kivételek bevezetése, az új kivételek egy kivételosztály leszármazottjai lesznek.
- A try/catch blokkokat emeljük ki önálló függvényekbe.
    - ❖ Összezavarják a kód szerkezetét, mivel keverik a szabályos feldolgozást és a hibakezelést.
    - ❖ A függvények egy dolgot kell, hogy csináljanak, a hibakezelés is egy dolog.

- Strukturált programozás: Használható egy függvényben akár több return utasítás, ciklusokban break és continue utasítás.

3. Mi a baj a megjegyzésekkel?

- A legjobb esetben is szükséges rosszak.
- A megjegyzések helyénvaló használata ellensúlyozza hiányosságainkat az önmagunk kóddal történő kifejezésében.
- Ha elég kifejezőek lennének a programozási nyelvek és mesteri módon tudnánk velük bánni a szándékaink kifejezéséhez, akkor nem nagyon lenne szükség megjegyzésekre.
- Azért nemkívánatosak a megjegyzések, mert nem mindig, és nem szándékosan, de túl gyakran közölnek pontatlan vagy valótlan információt.
- A kód változik, fejlődik, melyet nem minden esetben követnek a megjegyzések.
  - ❖ Nem életszerű a karbantartásuk.
  - ❖ Minél régebbi egy megjegyzés, annál valószínűbb, hogy rossz.
- A pontatlan megjegyzések még rosszabbak, mint a megjegyzések teljesen hiánya.
- A megjegyzések írásának egyik gyakori oka a rossz kód. Érdemesebb inkább rendbe tenni a rossz kódot, mint megjegyzésekkel megtűzdelni.
- A megjegyzések írásának egyik gyakori oka a rossz kód. Érdemesebb inkább rendbe tenni a rossz kódot, mint megjegyzésekkel megtűzdelni.

4. Jó és rossz megjegyzések fajtái

Jó megjegyzések fajtái:

- Jogi megjegyzések
- Informatív megjegyzések
- Szándékot magyarázó megjegyzés
- Tisztázó megjegyzés
- Következményekre figyelmeztető megjegyzés
- TODO megjegyzés
- Megerősítő megjegyzés
- Javadoc megjegyzés nyilvános API-ban

Rossz megjegyzések fajtái:

- Motyogás
- Fölösleges megjegyzés
- Félrevezető megjegyzés
- Kötelező megjegyzés
- Napló megjegyzés
- Zaj-megjegyzés
- Pozíciójelző/szalagcím megjegyzés
- Záró kapcsos zárójel megjegyzés
- Szerző neve megjegyzésben
- Megjegyzésbe tett kód
- HTML megjegyzés
- Nem lokális megjegyzés
- Túl sok információt tartalmazó megjegyzés
- A kódhoz nem nyilvánvalóan kapcsolódó megjegyzés
- Javadoc megjegyzés nem nyilvános kódban

5. Forráskód formázás

- A forráskódot úgy kell formázni, hogy az jól olvasható legyen.

- Az olvasó első benyomást a kódról a formázás alapján szerez.
- Egyszerű szabályokat kell választani a formázáshoz és azokat következetesen kell alkalmazni.
- Csapatban történő fejlesztés esetén meg kell állapodni egy kódolási konvencióban és mindenkinek azt kell követni.
- Függőleges és vízszintes formázás.

Függőleges formázás:

- Egy-egy üres sor jelöljön minden új fogalmat.
- Válasszuk el egymástól a csomagdeklarációt, az import deklarációkat, a függvényeket, …
- A szorosan kapcsolódó programsorok sűrűn kell, hogy megjelenjenek.
  - ❖ Ne legyenek közöttük megjegyzések vagy üres sorok.
- A szorosan kapcsolódó fogalmakat tartsuk függőlegesen egymáshoz közel.
  - ❖ Csak nagyon indokolt esetben kerüljenek külön állományokba.
  - ❖ Függőleges távolságuk tükrözze azt, hogy mennyire fontos az egyik a másik megértéséhez.
- A változókat deklaráljuk a használatuk helyéhez a lehető legközelebb.
  - ❖ Mivel a függvények rövidek, a lokális változókat az elejükön deklaráljuk.
  - ❖ A ciklusváltozókat lehet a ciklusokban.
- A példányváltozókat az osztályok elején kell deklarálni.
- Ha egy függvény meghív egy másikat, akkor függőlegesen közel kell, hogy legyenek egymáshoz, és ha ez egyáltalán lehetséges, akkor a hívó előzze meg a hívottat.
- A sorok ne legyenek túl hosszúak. Ne legyenek hosszabbak például 120 karakternél.
- Használjunk szóközöket a gyengén összetartozó elemek között.
  - ❖ Például értékadó operátor két oldalán. Ne tegyünk viszont szóközt például egy függvény neve és az azt követő nyitó zárójel karakter közé.
- Nem érdemes a deklarációkban és értékadásokban a neveket és a kifejezéseket igazítani.
- Nagyon fontos a megfelelő behúzás.
  - ❖ Használható szóköz és tabulátor is.
  - ❖ Ne szegjük meg a behúzási szabályt rövid if utasítások, ciklusok vagy függvények kedvéért sem.
- Ha for vagy while ciklus törzseként üres utasítást kell használni, ezt célszerű egy új sorba elhelyezni.

6. Újság metafora

- Egy jól megírt újságcikket felülről lefelé haladva olvasunk.
- A tetején egy olyan cím van, mely alapján az olvasó eldöntheti, hogy érdekli-e egyáltalán. Az első bekezdés a teljes történet egy összefoglalását adja. Lefelé haladva a bekezdések egyre több és több részletet tartalmaznak.
- Egy forrásállomány is legyen olyan, mint egy újságcikk.
- A neve legyen egyszerű és beszédes. Az eleje magas szintű fogalmakat és algoritmusokat tartalmazzon. Lefelé haladva egyre nagyobb hangsúlyt kapnak a részletek. A végén legyenek a legalacsonyabb szintű függvények.

7. Hibakezelés, ellenőrzött és nem ellenőrzött kivételek

Hibakezelés:

- Hibakódok visszaadása helyett részesítsük előnyben a kivételeket.

- Használjunk nem ellenőrzött kivételeket.
- Ne adjunk át/vissza null-t.

Ellenőrzött és nem ellenőrzött kivételek:

- Java-ban a kivételek ellenőrzöttek (checked) vagy nem ellenőrzöttek (unchecked).
- Nyelvek, melyekben nincsenek ellenőrzött kivételek: C#, C++, Kotlin, Python, Scala, …
- Az ellenőrzött kivételek használata megsértheti a nyitva zárt elevet.

Ellenőrzött kivételek:

- A metódusok deklarálják az ellenőrzött kivételeket, melyek bekövetkezhetnek a végrehajtásuk során, mely lehetővé teszi a fordításidejű ellenőrzést annak biztosításához, hogy a kivételek kezelésre kerüljenek.
- A throws záradék szolgál azoknak az ellenőrzött kivételeknek a jelzésére, melyeket egy metódus vagy konstruktor törzs utasításai dobhatnak.
- Az ellenőrzött kivételek arra kényszerítik a programozót, hogy foglalkozzon velük, mivel el kell kapni őket, így a kód megbízhatóságát növelik.
- Kritikus programkönyvtárak számára ajánlott ellenőrzött kivételek dobása.

Nem ellenőrzött kivételek:

- Bárhol dobhatók egy metódus vagy konstruktor törzsében.
- A nem ellenőrzött kivételosztályok a java.lang.RuntimeException vagy a java.lang.Error osztály alosztályai.