1. Unified Modeling Language (UML)

## 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.)

## Modell és metamodell fogalma

Modell:

* A modell egy rendszer leírása, ahol a rendszer a lehető legszélesebb jelentésben értendeő (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 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.

## Szakterület-specifikus nyelvek, példák

* Szakterület-specifikus nyelv (domain specific language): Egy bizonyos fajta problémára koncentráló számítógépes nyelv, nem pedig egy általános célú nyelv tetszőleges fajta problémák megoldásához.
* Példák: BibTeX/LaTeX, CSS, DOT, Gradle DSL, Make, PlantUML, SQL, …
* A MOF (Meta Object Facility) egy nyílt és platformfüggetlen metaadat kezelő keretrendszert és kapcsolódó metaadat szolgáltatásokat biztosít, melyek lehetővé teszik modell- és metaadat vezérelt rendszerek fejlesztését és együttműködését.

## UML modellelemek: osztályozók, csomagok, függőségek, kulcsszavak, megjegyzések

Osztályozók:

* Az osztályozó egy modellelem, mely közös jellemzőkkel (tulajdonságokkal, műveletekkel) rendelkező példányok egy halmazát ábrázolja.
* Hierarcihába 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.

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.
  + Lásd például az osztályokat és interfészeket.
* Megadásuk francia idézőjelek – és karakterek – között.
  + Ha a használt betűkészletben nem állnak rendelkezésre a francia idézőjelek, akkor a ‘és’ karakterekkel helyettesíthetők.
* Egy modellelemre több kulcsszó is vonatkozhat.
  + A kulcsszavak felsorolhatók egymás után, mindegyik külön határolók közé zárva.
  + Több kulcsszó is megadható a határolók között vessző katakterekkel elválasztva.

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.

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.

## Osztálydiagramok, osztálydiagramok fajtái

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

Osztálydiagram fajtái:

* 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 konstrukcióival ekvivalensek.

## Osztály

* Jelölésmód
* Név – Attribútumok – Műveletek

## Láthatóság

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

## 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.
    - A 0…\* számosság helyett használható az ekvivalens \* jelölés.

## Tulajdonságok

* Egy tulajdonság egy attribútumot vagy egy asszociációvéget ábrázol.
* Jelölésmód:
  + [^] [láthatóság] [/] név [: típus] [számosság] [=alapérték] [{ módosító [, módosító]\*}]
    - A ^ azt jelzi, hogy a tulajdonság örökölt.
    - A / azt jelzi, hogy a tulajdonság származtatott.
    - A számosság elhagyásakor az alapértelmezés 1.
    - Módosító: például readOnly, ordered, unordered, unique, …

## 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.

## Statikus attribútumok és műveletek

* A statikus attribútumokat és műveleteket aláhúzás jelöli.

## Absztrakt osztályok

* Nem példányosítható osztály (osztályozó)
* Jelölésmód:
  + Szedjük az osztály 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!

## 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 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 assszociáció szimbólumához megadható név (ne legyen túl közel egyik véghez sem).
  + Folytonos vonallal ábrázolt bináris asszociáció neve mellett vagy helyén elhelyezhető egy tömör háromszög, mely a vonal mentén az egyik vég felé mutat és az olvasási irányt jelzi. Ez a jelölés csupán dokumentációs célokat szolgál.
* Asszociáció vég: az asszociációt ábrázoló vonal és egy osztályozót ábrázoló ikon (gyakran egy doboz) kapcsolata.
  + A vonal végének közelében elhelyezhető (egyik sem kötelező):
    - Név (gyakran szerepkörnek nevezik)
    - Számosság (ha nincs megadva, akkor semmilyen feltevéssel nem élhetünk a számosságról)
    - Módosító (lásd a tulajdonságoknál)
    - Láthatóság
  + A vonal végén egy nyílt nyílhegy azt jelzi, hogy a vég navigálható, egy x pedig azt, hogy a vég nem navigálható.
* Egy asszociáció vég számosságának jelentése:
  + A példányok számát adja meg a végen arra az esetre, amikor a többi (n-1) vég mindegyikén egy-egy 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 az a mechanizmus, mely révén hatékony elérés történik.
  + Az osztályokhoz tartozó asszociációvégek mindig navigálhatók, az asszociációkhoz tartozók lehetnek navigálhatók és nem navigálhatók.

## Egész-rész kapcsolat

* A bináris asszociációk egész-rész kapcsolatot kifejező fajtái:
  + Aggregáció: Egy rész objektum egyidejűleg több aggregációs objektumhoz is tartozhat, a részek és az aggregációs objektum egymástól függetlenül is létezhetnek.
  + Kompozíció: Az aggregáció erősebb formája. Egy rész objektum törlésekor az összes rész objektum vele együtt törlődik.
* Egy bináris asszociáció egyik vége jelölhető meg csak aggregációként vagy kompozícióként.

## 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.

## 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.

2. Szoftvertesztelés

## 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 a használatba vétel előtt annak megmutatása, hogy egy szoftver azt csinálja, amit kell, valamint a programhibák felfedezése.

## Verifikáció és validáció

* Verifikáció (verification): annak ellenőrzése, hogy a szoftver megfelel-e a vele szemben támasztott (funkcionális és nem funkcionális) követelményeknek.
* Validáció (validation): annak ellenőrzése, hogy a szoftver megfelel-e az ügyefelek elvárásainak.

## Hibát leíró szakkifejezések

* 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 megkövetelt funkciót a megszabott határok között.

## Tesztelési alapelvek

* *A szoftvertesztelés hét alapelve:*
  + A tesztelés a hibák jelenlétét mutatja meg, nem a hiányukat
  + Lehetetlen a kimerítő tesztelés
  + A korai tesztelés időt és pénzt takarít meg
  + A hibák csoportosulnak
  + Óvakodj a kártevőírtó paradoxontól
  + A tesztelés környezetfüggő
  + A hibamentesség egy tévhit

## Teszteset, teszadatok

* *Magas szintű teszteset:* Teszteset, mely absztrakt előfeltételekkel, bemeneti 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.
* A tesztadatok a tesztvégrehajtáshoz szükséges adatokat jelentik.

## Tesztelési szintek

* *Egységtesztelés:*
  + A függetlenül tesztelhető komponensekre összpontosít.
  + Az egységtesztelést 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.
* *Integrációs tesztelés:*
  + Komponensek vagy rendszerek közötti kommunikációra összpontosít.
  + Az integrációs teszteknek magára az integrációra kell koncentrálnia, nem pedig az egyes komponensek/rendszerek működésére.
  + Komponens integrációs tesztelés: az integrált komponensek közötti kommunikációra és interfészekre összpontosít.
  + Rendszerintegrációs tesztelés: rendszerek közötti kommunikációra és interfészekre összpontosít.
* *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.
  + 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 kiadását elérhetővé teszik kipróbálásra ügyfelek és felhasználók egy nagyobb csoportjának. A felhasználók helyén történik. Főleg olyan szoftvertermékekhez alkalmazzák, melyeket sok különböző környezetben használnak.

## Teszttípusok

* *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 teszek.
  + 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.
  + **Megerősítő tesztelés:** célja annak megerősítése, 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 akartalan mellékhatások érzékelése.

## 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 teszeknek logikai kimenete kell, hogy legyen. Vagy átmennek, vagy megbuknak.
* *Jól időzített (Timely):* A teszteket kellő időben kell megírni, közvetlenül a tesztelendő kód előtt.

## Egységtesztek szervezése: az AAA minta

* *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.

## JUnit

* *Tesztosztály:* bármely felsőszintű osztály, statikus tagosztály vagy @Nested osztály, mely legalább egy tesztmetódust tartalmaz.
* *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.
* Deklarálhatók az aktuális tesztosztályon belül lokálisan, örökölhetők ősosztályból vagy interfészektől.
* Nem lehetnek absztraktak és nem adhatnak vissza értéket.
* *Teszteredmény:*
  + *Siker (success):* amikor a teszt végrehajtásakor minden tényleges eredmény megegyezik a várt végeredménnyekkel.
  + *Bukás (failure):* amikor a teszt végrehajtásakor a tényleges eredmény nem egyezik meg a várt végeredménnyel.
  + *Hiba (error):* amikor a teszt végrehajtásakor egy hiba következik be, mely megakadályozza a befejeződést.

## Kódlefedettség

* Egy kódlefedettségi metrika (röviden lefedettségi metrika) a végrehajtott forráskód mennyiségét méri százalékosan kifejezve egy tesztkészlet futtatásakor.
* Utasítás lefedettség/sor lefedettség:
  + Utasítás lefedettség = Végrehajtott utasítások / Összes utasítás száma
  + Sor lefedettség = Végrehajtott kódsorok / Összes sor száma
  + Minél tömörebb a kód, annál jobb az utasítás/sor lefedettség, mivel az utasítások/sorok nyers számán alapul.
* 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
* *Mi az ésszerű lefedettségi szám?:*
  + Veszélyes egy bizonyos érték elérésének megcélzása egy lefedettségi metrikánál, mivel könnyen ez válhat a fő céllá.
  + Inkább a megfelelő egységtesztelésre kell koncentrálni.
* *Ökölszabályok:*
  + Jó, ha egy rendszer fő részeinél nagy a lefedettség.
  + Nem jó ezt magas szintű követelménnyé tenni.

# 3. Objektumorientált tervezési alapelvek

## Statikus kódelemzés

* A statikus kódelemzés (static code analysis) a programkód elemzésének folyamata, mely a kód végrehajtása nélkül történik.
* Statikus kódelemző (eszköz) (static code analyzer, static code analysis tool): statikus kódelemzést végző automatikus eszköz.
* 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 eszközök:
  + Coala
  + Infer
  + PMD
  + Semgrep

## DRY

* Ne ismételd magad (Don't Repeat Yourself)
* 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.

## KISS

* Keep it simple, stupid
* Kelly Johnson (1910–1990) repülőmérnöknek tulajdonítják a kifejezést.
* Az egyszerűségre való törekvés

## YAGNI

* A „You Aren't Gonna Need It” („nem lesz rá szükséged”) rövidítése.
* „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

## Csatoltság

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

## Demeter törvénye

* Demeter törvényét (Law of Demeter) Ian Holland javasolta 1987-ben.
* Más néven: ne beszélgess idegenekkel (Don't Talk to Strangers)
* A metódusok üzenetküldési szerkezetét korlátozza.
* Célja az osztályok közötti függőségek szervezése és csökkentése.
* Alkalmazása növeli a karbantarthatóságot és az érthetőséget.

## GoF

* 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)
* *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.

## SOLID

* **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:*
  + Robert C. Martin által megfogalmazott elv: – „A class should have only one reason to change.”
  + Egy osztálynak csak egy oka legyen a változásra.
  + 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.
* *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
  + 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): Az ésszerűen szükségesnél több tagfüggvénnyel és baráttal rendelkező interfész.
  + 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.
* *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.
  + 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.

## 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.
* A függőség befecskendezés alkalmazható DI konténer nélkül
* Tiszta DI: függőség befecskendezés alkalmazásának gyakorlata DI konténer nélkül.
* *A függőség befecskendezés előnyei:*
  + Kiterjeszthetőség
  + Karbantarthatóság
  + Tesztelhetőség: a függőség befecskendezés támogatja az egységtesztelést.

4. Minták a szoftverfejlesztésben

* A fogalom Christopher Alexander (1936–2022) építész nevéhez fűződik.
* Szinte azonnal átvették az elgondolást a szoftvertervezők.
* Széles körben ismertté és elterjedtté vált a „négyek bandájaként” ismert szerzők könyve révén a szoftveriparban.

# 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.
* Minden minta egy három részből álló szabály, mely egy bizonyos környezet, egy probléma és egy megoldás közötti kapcsolatot fejez ki.
* *Részei:*
* **Környezet** (Context):
  + Mely helyzetekben fordul elő a probléma.
* **Probléma** (Problem):
  + Az adott környezetben ismétlődően felmerülő probléma.
  + Erő (Force): Olyan szempontot jelent, melyet a megoldás során figyelembe kell venni.
    - Például: a megoldással szemben támasztott követelmények, megszorítások, a megoldás kívánatos jellemzői.
    - Az erők különböző nézőpontokból elemzik a problémát. Kiegészíthetik egymás vagy ellentmondhatnak egymásnak. – Példa ellentmondó erőkre: a rendszer kiterjeszthetősége és a kód méretének minimalizálása.
* **Megoldás** (Solution):
  + Hogyan oldjuk meg az ismétlődő problémát? Hogyan ellensúlyozzuk ki az erőket?
  + Egy megoldási sémát ad, nem egy részletes tervet.
  + Mentális építőkocka.

# Mintakatalógusok és mintanyelvek

* Mintakatalógus
  + Minták egy tetszőleges csoportja.
  + A gyűjtemény tartalmát tekintve lehet heterogén vagy fókuszálhat egy adott területre, problémára vagy absztrakciós szintre.
  + Szervezése történhet strukturálatlanul vagy strukturáltan.
  + A minták leírása többé-kevésbé egymástól függetlenül történik.
  + Példa: a GoF-féle katalógus
* Mintanyelv
  + Egymással összefüggő olyan minták egy gyűjteménye, melyek együtt meghatároznak egy szisztematikus folyamatot szoftverfejlesztési problémák megoldására.

# Architekturális minták

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.

# A modell-nézet vezérlő (MVC) architekturális minta(példa)

* **Név**: Modell-nézet vezérlő (Model-View-Control)
* **Környezet**: Rugalmas ember-gép felülettel rendelkező interaktív alkalmazások.
* Probléma: Különösen gyakori az igény a felhasználói felületek változtatására.
  + **Erők**:
    - Ugyanaz az információt különböző módon jelenik meg különböző helyeken (például oszlop- vagy kördiagramon).
    - Az alkalmazás megjelenítésének és viselkedésének azonnal tükröznie kell az adatokon végzett műveleteket.
    - A felhasználói felület könnyen változtatható kell hogy legyen, akár futásidőben is.
    - Különböző look and feel szabványok támogatása vagy a felhasználói felület portolása nem érintheti az alkalmazás magjának kódját.
* **Megoldás**: Az interaktív alkalmazás három részre osztása:
  + A modell komponens az adatokat és a funkcionalitást csomagolja be, független a kimenet ábrázolásmódjától vagy az input viselkedésétől.
  + A nézet komponensek jelenítik meg az információkat a felhasználónak.
  + A vezérlő fogadja a bemenetet, melyet szolgáltatáskérésekké alakít a modell vagy a nézet felé.

# Példa architekturális mintára

* A modell elválasztása a nézet komponenstől több nézetet is lehetővé tesz ugyanahhoz a modellhez.
* Ugyanazok az adatok többféle módon is megjeleníthetők.
* A nézet elválasztása a vezérlő komponenstől kevésbé fontos.
* Lehetővé tesz ugyanahhoz a nézethez akár több vezérlőt is.
* A klasszikus példa ugyanahhoz a nézethez szerkeszthető és nem szerkeszthető viselkedés támogatása két vezérlővel.
* A gyakorlatban sokszor csak egy vezérlő van nézetenként
* 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)

# Tervezési minták

* 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 testreszabott formában valamilyen általános tervezési problémát oldanak meg egy bizonyos összefüggésben.
* A 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.

# Példa tervezési mintára: egyke

* **Cél**: Egy osztályból csak egy példányt engedélyezni, és ehhez globális hozzáférési pontot ad meg.
* **Indíték**: Egyes osztályok esetében fontos, hogy pontosan egy példány legyen belőlük.
* **Alkalmazhatóság**: Az egyke mintát a következő esetben használjuk:
  + Pontosan egy példányra van szükség valamelyik osztályból, és annak elérhetőnek kell lennie az ügyfelek számára a jól ismert elérési pontokból.

# Tesztelési minták

* **Osztályonkénti teszteset osztály** (Testcase Class per Class)
* Egy adott osztályt tesztelő összes tesztmetódust helyezzük egy teszteset osztályba.
* Ha nincs túl sok tesztmetódus vagy csak most kezdtünk el teszteket írni a tesztelendő rendszerhez.
* Ahogy nő a tesztek száma és jobban megértjük a tesztadat (fixture) követelményeinket, a teszteset osztályunkat több osztályra vághatjuk szét (lásd: Testcase Class per Fixture, Testcase Class per Feature).
* Képességenkénti teszteset osztály (Testcase Class per Feature)
  + A tesztmetódusokat teszteset osztályokba csoportosítjuk az alapján, hogy a tesztelt rendszer melyik tesztelhető képességét mozgatják meg. – Mikor használjuk?
  + Amikor jelentős számú tesztmetódus van és nyilvánvalóbbá akarjuk tenni a tesztelt rendszer minden egyes képességének specifikációját.
  + Sajnos ez nem teszi a tesztmetódusokat egyszerűbbé vagy könnyebben érthetővé.
  + A képességek nagy száma annak lehetőségét jelző „szag”, hogy az osztálynak túl sok felelőssége van.

# Felhasználói felület tervezési minták

* Mintasablon (Tidwell):
  + Mit: Helyezz egy képet vagy színátmenetet az oldal hátterébe, mely vizuálisan visszaugrik az előtér elemek mögött.
  + Mikor használd: Az oldalnak hangsúlyos vizuális elemei vannak (mint például szövegblokkok, vezérlő elemek csoportjai vagy ablakok), és nem nagyon sűrű az elrendezés. Azt szeretnéd, hogy az oldal jellegzetes és szemrevaló legyen, vizuális márkaépítési stratégiára is gondolhatsz. Valami érdekesebb oldalhátteret szeretnél egy sima fehérnél vagy szürkénél.
  + Miért: A lágy fókuszú, színátmenetes és más mélységjelzőkkel rendelkező hátterek hátrahúzódnak az előterükben lévő élesebben definiált tartalom mögött. A tartalom ilyen módon „lebegni” látszik a háttér előtt. Ez a pszeudo-3D kinézet egy erős figura-alap hatást eredményez.
  + Hogyan: Használj olyan hátteret, mely rendelkezik ezen jellemzők közül eggyel vagy többel: lágy fókusz, színátmenetek, mélységjelzők, nincsenek erős fókuszpontok.

# 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
* Egy nagyon hasonló fogalom a tervezési szag (design smell):
  + A tervezési szagok olyan struktúrák a tervezésben, melyek alapvető tervezési elvek megsértését jelzik és negatív hatással vannak a tervezés minőségére.

# Szoftverfejlesztési antiminta: 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.
* **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

# 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.”

● 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

# Létrehozási minták (GoF)

* Elvont gyár (Abstract Factory)
* Építő (Builder)
* Gyártó metódus (Factory Method)
* Prototípus (Prototype)
* Egyke (Singleton)

# Elvont gyár

Cél: Kapcsolódó vagy egymástól függő objektumok családjának létrehozására szolgáló felületet biztosít a konkrét osztályok megadása nélkül.

# Építő

Cél: Az összetett objektumok felépítését függetleníti az ábrázolásuktól, így ugyanazzal az építési folyamattal különböző ábrázolásokat hozhatunk létre.

# Gyártófüggvény

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.

# Prototípus

Cél: Prototípus példány használatával határozza meg, hogy milyen típusú objektumokat kell létrehozni, az új objektumokat pedig ennek a prototípusnak a lemásolásával állítja elő.

# Egyke

Cél: Egy osztályból csak egy példányt engedélyez, és ehhez globális hozzáférési pontot ad meg.

# További létrehozási minták

* Függőség befecskendezés (Dependency Injection)
* Többke (Multiton)
* Objektumkészlet (Object Pool)
* Érték objektum (Value Object)

# Szerkezeti minták (GoF)

* Illesztő (Adapter)
* Híd (Bridge)
* Összetétel (Composite)
* Díszítő (Decorator)
* Homlokzat (Facade)
* Pehelysúlyú (Flyweight)
* Helyettes (Proxy)

# Illesztő

Cél:

– Az adott osztály interfészét az ügyfelek által igényelt interfésszé alakítja.

– E módszerrel az egyébként összeférhetetlen interfészű osztályok együttműködését biztosíthatjuk.

# Díszítő

Cél:

– Az objektumokhoz dinamikusan további felelősségi köröket rendel.

– A kiegészítő szolgáltatások biztosítása terén e módszer rugalmas alternatívája az alosztályok létrehozásának.

# Viselkedési minták (GoF)

● Felelősséglánc (Chain of Responsibility)

● Parancs (Command)

● Értelmező (Interpreter)

● Bejáró (Iterator)

● Közvetítő (Mediator)

● Emlékeztető (Memento)

● Megfigyelő (Observer)

● Állapot (State)

● Stratégia (Strategy)

● Sablonfüggvény (Template Method)

● Látogató (Visitor)

# Felelősséglánc

Cél:

– A minta arra szolgál, hogy elkerüljük a kérelem küldőjének a fogadóhoz való kötését.

– Ezt úgy érjük el, hogy több objektumnak is jogot adunk a kérelem kezelésére.

– A fogadó objektumokat láncba állítjuk, amelyen a kérelem addig halad, amíg el nem ér egy objektumot, ami képes a kezelésére.

# Bejáró

Cél: Az összetett objektumok elemeinek soros elérését a háttérben megbúvó ábrázolás felfedése nélkül biztosító módszer kialakítása.

# 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.

# Látogató

Cél: Egy objektumszerkezet elemein végrehajtandó műveletet ábrázolni: a Látogató minta segítségével anélkül határozhatunk meg egy új műveletet, hogy a benne részt vevő elemek osztályát meg kellene változtatnunk.

# 5. Tiszta Kód

# 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.

● 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.

# 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.

● A függvények csak egy dolgot csináljanak, de azt jól.

● Függvényenként egy absztrakciós szint.

● Argumentumok:

– 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

● Mellékhatásmentesség:

– A függvények legyenek mellékhatásmentesek.

● Csak azt csinálják, amit ígérnek.

– Kerülni kell output argumentumok használatát.

● A try/catch blokkokat emeljük ki önálló függvényekbe.

● Strukturált programozás:

– Használható egy függvényben akár több return utasítás, ciklusokban break és continue utasítás.

# Megjegyzések

● A legjobb esetben is szükséges rosszak.

● A megjegyzések helyénvaló használata ellensúlyozza hiányosságainkat az önmagunk kóddal történő kifejezésében.

● 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

● A pontatlan megjegyzések még rosszabbak, mint a megjegyzések teljes hiánya.

Jó megjegyzések

● Jogi megjegyzések

● Informatív megjegyzések

● Szándékot magyarázó megjegyzés

● Tisztázó megjegyzés

● Következményekre figyelmeztető megjegyzés

● TODO megjegyzés

● Megerősítő megjegyzés

● Javadoc megjegyzés nyilvános API-ban

# Hibakezelés

● Hibakódok visszaadása helyett részesítsük előnyben a kivételeket.

● Használjunk nem ellenőrzött kivételeket.

● Ne adjunk át/vissza null-t.