



Szoftverfejlesztés

II. Zárthelyi Dolgozat

Tartalom

1. Unified Modeling Language

- UML modellelemek
- Osztálydiagramok
- Osztályok
- Számosság
- Tulajdonságok
- Műveletek
- Statikus attribútumok és műveletek
- Absztrakt osztályok
- Asszociációk
- Egész-rész kapcsolat
- Általánosítás
- Interfészek

2. Szoftvertesztelés

- Mi a szoftvertesztelés?
- Verifikáció és validáció fogalma
- Hibát leíró szakkifejezések
- Tesztelési alapelvek
- Teszteset és tesztadat fogalma
- Tesztelési szintek
- Teszt típusok
- A jó egységteszt ismertetőjegyei: FIRST
- Egységtesztek szervezése: az AAA minta
- JUnit
- Kódlefedettségi metrikák
- Mi a tesztvezérelt fejlesztés (TDD)

3. Objektumorientált tervezési alapelvek

- Statikus kódelemzés fogalma
- A DRY elv
- A KISS elv
- A YAGNI elv
- Csatoltság, laza és szoros csatlakozás
- GoF alapelvek
- SOLID alapelvek
- Függőség befecskendezés

4. Minták a szoftverfejlesztésben

5. Tiszta kód

1. Unified Modeling Language (UML)

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

osztályozók

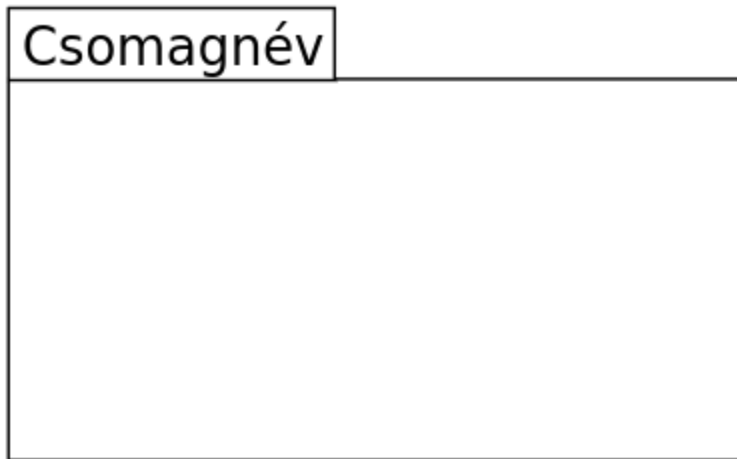
Az osztályozó egy modellelem, mely közös jellemzőkkel műveletekkel rendelkező példányok egy halmazát ábrázolja.

- **Hierarchiába** szervezhetők az általánosítás révén.
- **Specializációi:** *DataType, Association, Interface, Class*
- **Jelölésmód:** mint az osztályoké, a nevük megjelenítéséhez félkövér betűtípust kell használni.

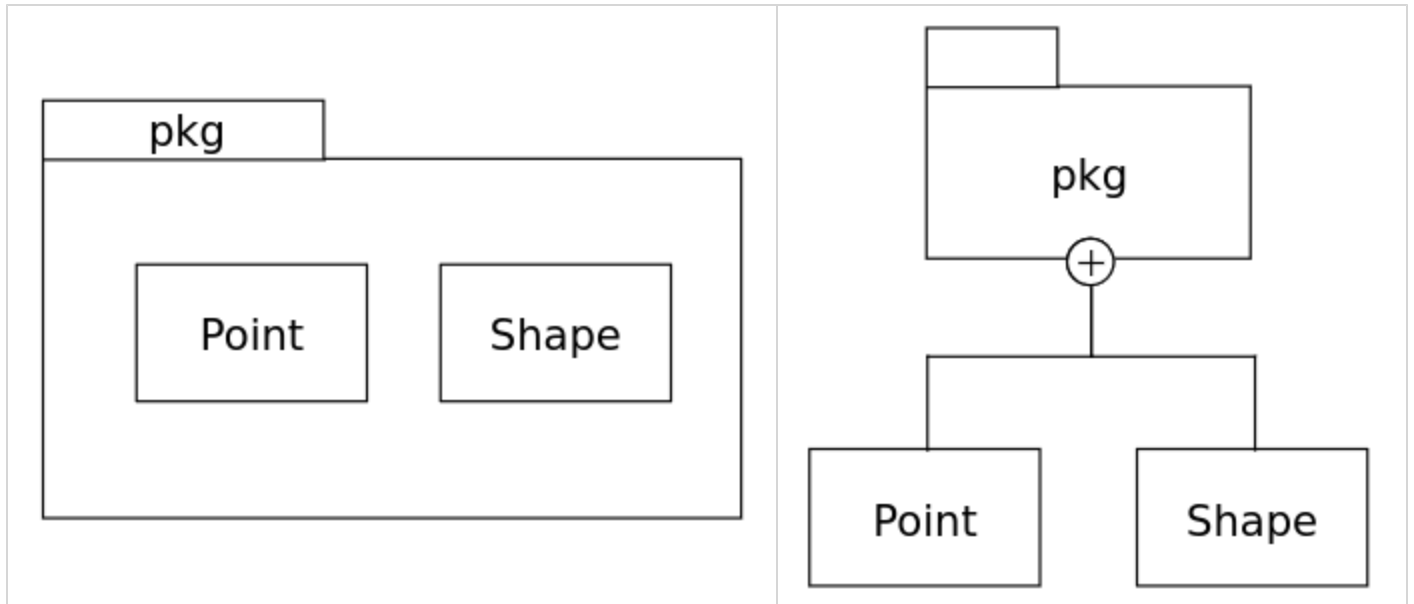
csomagok

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

Jelölésmód:



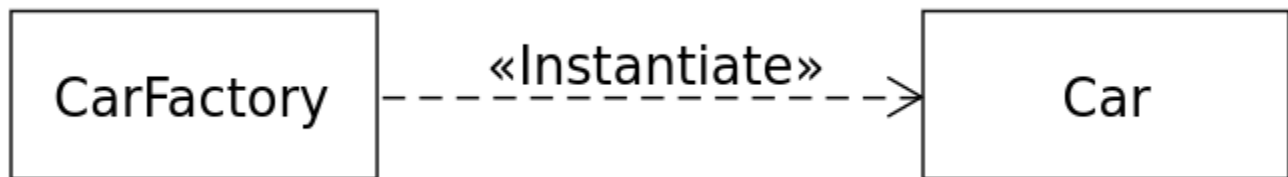
A tartalmazott elemekre csomagnév::elemnév formájú minősített nevekkel lehet hivatkozni (például `pkg::Point` , `pkg::Shape`).



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 képező 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 (`«interface»`).

Megadásuk francia idézőjelek, `«` és `»` karakterek között.

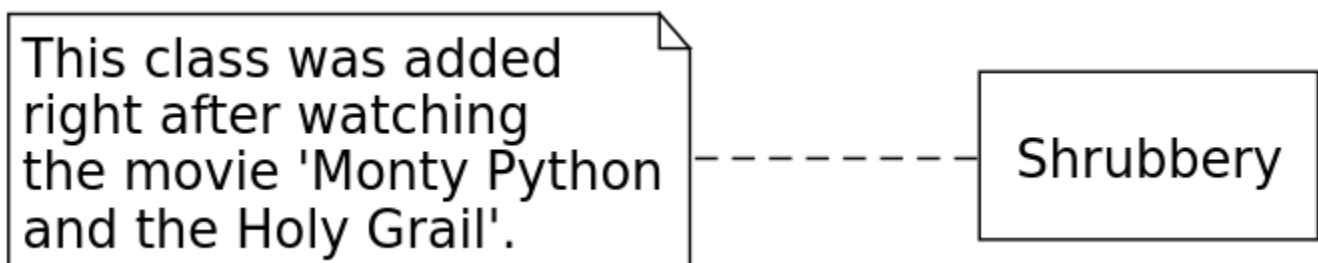
- Ha a használt betűkészletben nem állnak rendelkezésre a francia idézőjelek, akkor a `>>` és `<<` karakterekkel helyettesíthetők.
- Egy modellelemre több kulcsszó is vonatkozhat.
- A kulcsszavak felsorolhatók egymás után, mindegyik külön határolók közé zárva.
- Több kulcsszó is megadható a határolók között vessző karakterekkel 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.



Osztálydiagramok, osztálydiagramok fajtái

Osztálydiagramok

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

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.

Osztályok

Jelölésmód:

Név
Atribútumok
Műveletek

Láthatóság

Jelölés	Láthatóság
+	nyilvános
-	privát
#	védett
~	csomagszintű

Számosság

Megszorítást fejez ki egy kollekció elemeinek számára.

- Az elemek száma nem lehet kisebb az alsó, illetve nagyobb az felső korlátnál.

Jelölésmód:

[alsó_korlát] [...] felső_korlát

- Például 1..2
- A 0..* számosság helyett használható a * jelölés.

Az alsó korlát **nemnegatív egész**, a felső korlát **nemnegatív egész** vagy a "korlátlan" jelentésű *.

Ha az alsó és felső korlát egyenlő, akkor használható önmagában a felső korlát.

- Például 1..1 → 1 vagy 5..5 → 5

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 [, tulajdon
```

- **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.

Példa:

Singleton
<u>-instance: Singleton</u>
<u>-Singleton()</u> <u>+getInstance(): Singleton</u>

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.)

<i>Shape</i>
-x: int -y: int
#Shape(x: int, y: int) +getX(): int +getY(): int +moveTo(newX: int, newY: int) <i>+getArea(): double</i> <i>+draw()</i>

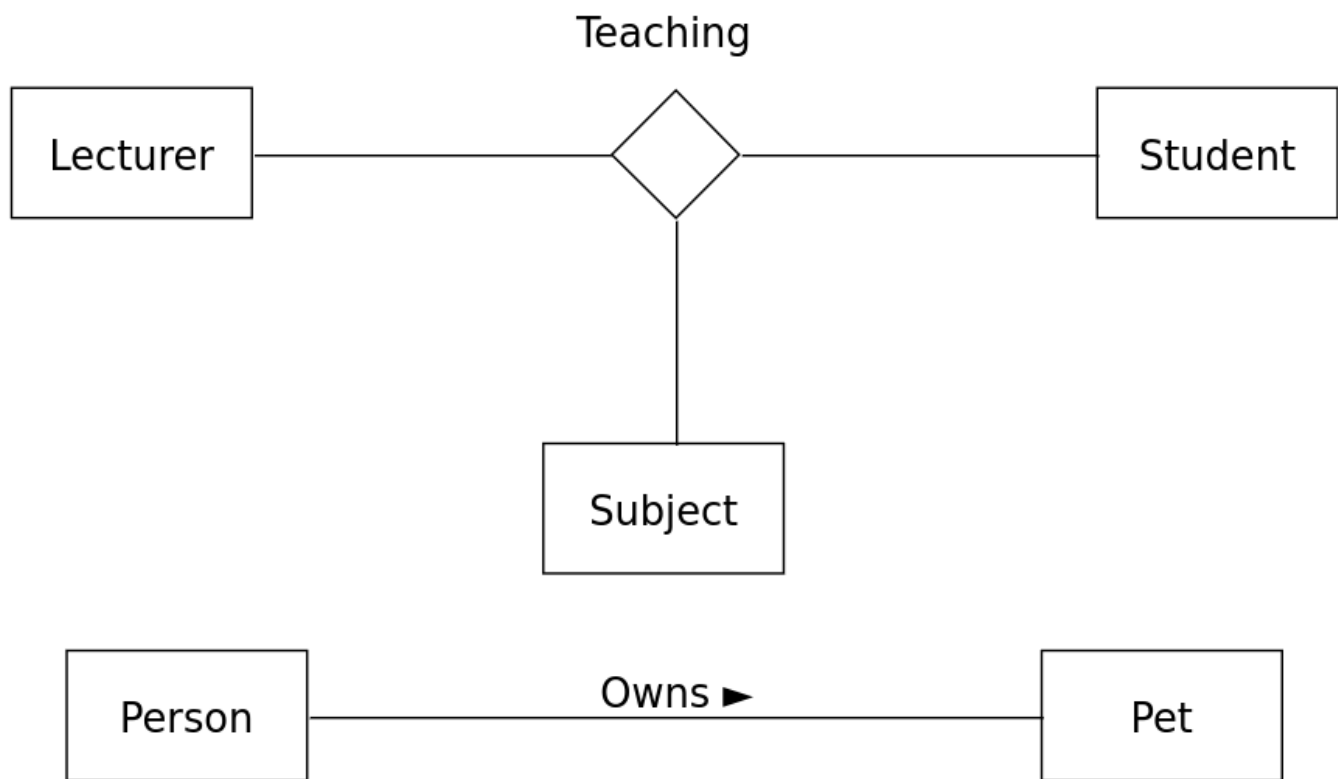
Asszociációk

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).
- 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.

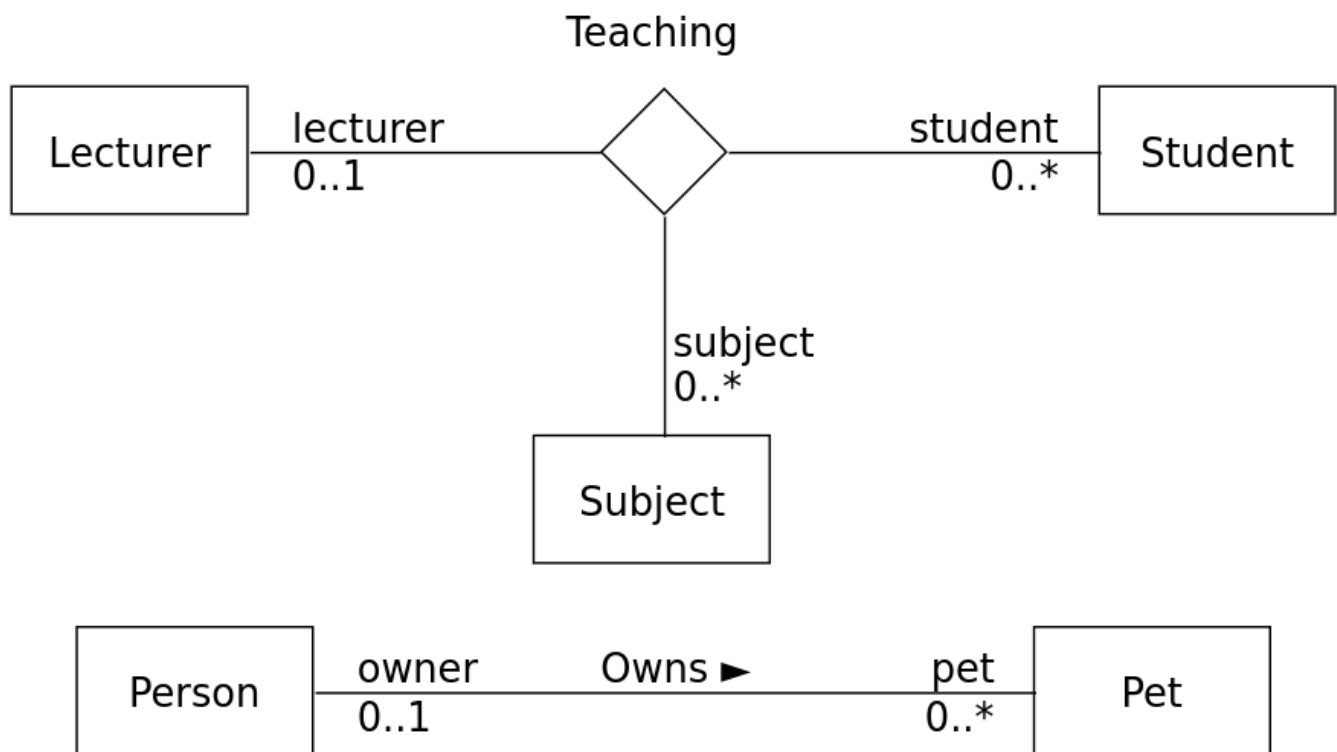
A vonal végének közelében elhelyezhető (egyik sem kötelező):

- 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 \times 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.







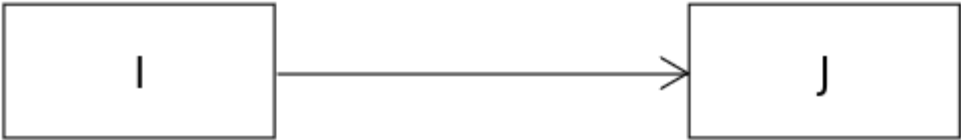
Az osztályozó és a vonal érintkezési pontjában elhelyezhető egy kis tömör kör (a továbbiakban pontnak nevezzük).

- 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 azt a mechanizmus, mely révén hatékony elérés történik.
- Az osztályokhoz tartozó asszociációvégek mindig navigálhatók, az asszociációkhoz tartozók lehetnek navigálhatók és nem navigálhatók.

-	-
Mindkét vég navigálható.	
Egyik vég sem navigálható.	
A navigálhatóság nem meghatározott.	
Az egyik vég navigálható, a másik nem.	
Az egyik vég navigálható, a másik nem.	

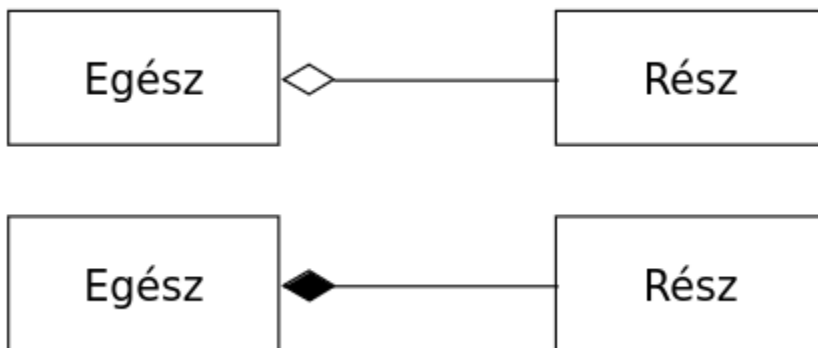
Egész-rész kapcsolat

A bináris asszociációk egész-rész kapcsolatot kifejező fajtái:

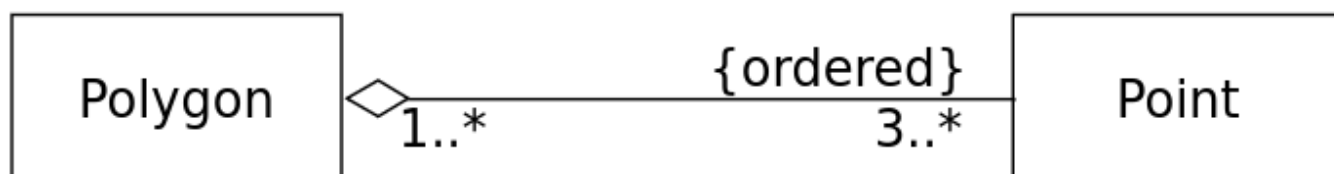
- **Aggregáció** (*shared aggregation, aggregation*): Egy rész objektum egyidejűleg több aggregációs objektumhoz is tartozhat, a részek és az aggregációs objektum egymástól függetlenül is létezhetnek.
- **Kompozíció** (*composite aggregation, composition*): Az aggregáció erősebb formája. Egy rész objektum 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.

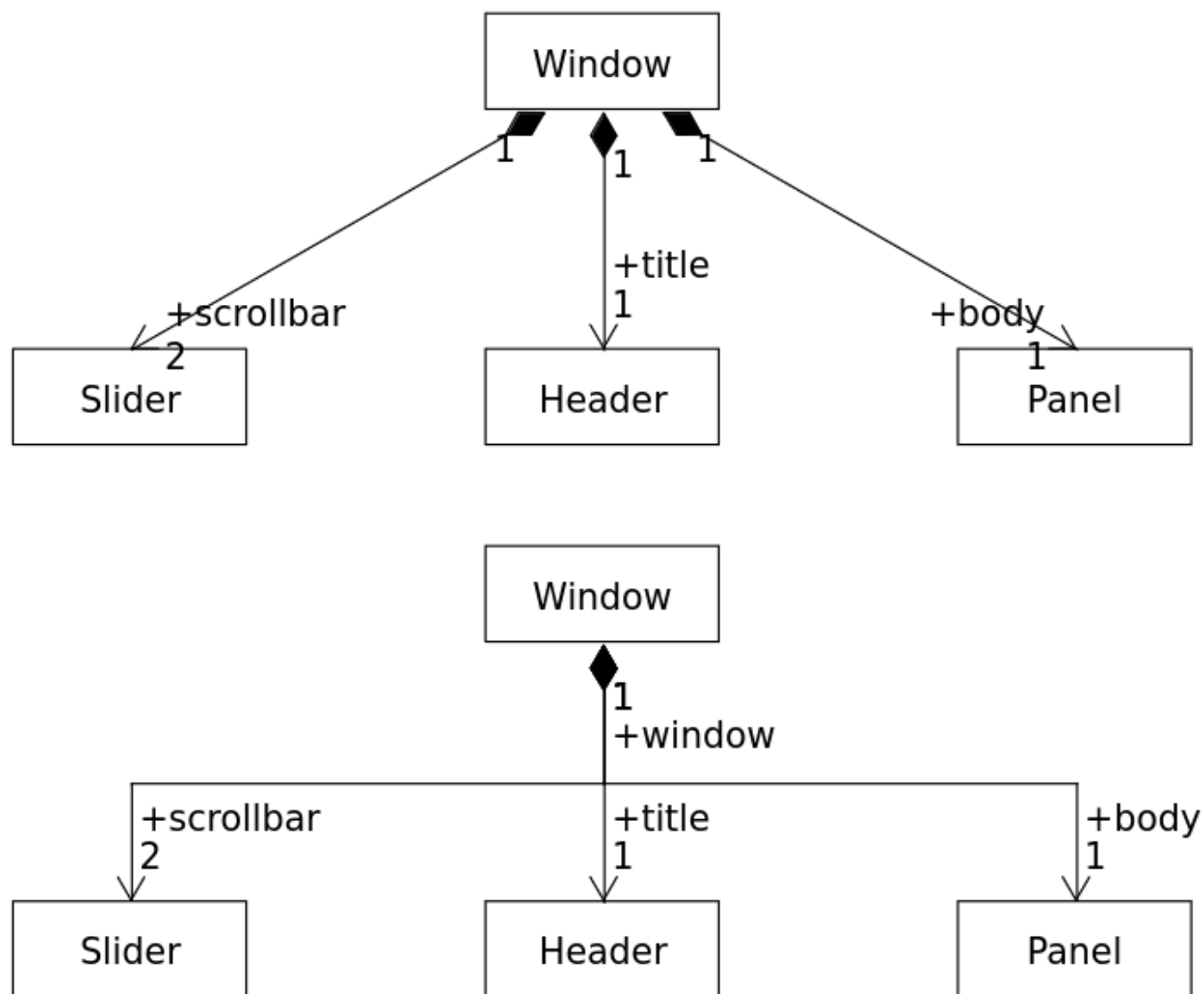
Jelölésmód:



Példa aggregációra:



Példa kompozícióra:



Általánosítás

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

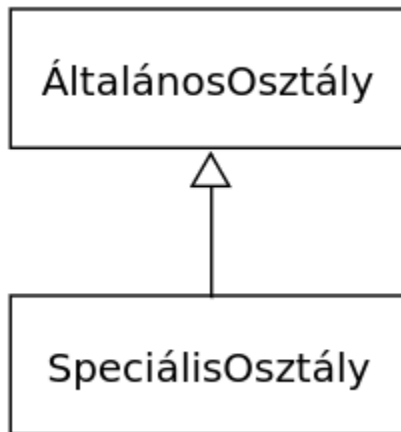
Az általánosítás/specializáció reláció tranzitív lezártja szerint értelmezzük egy osztályozó általánosításait és specializációit.

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

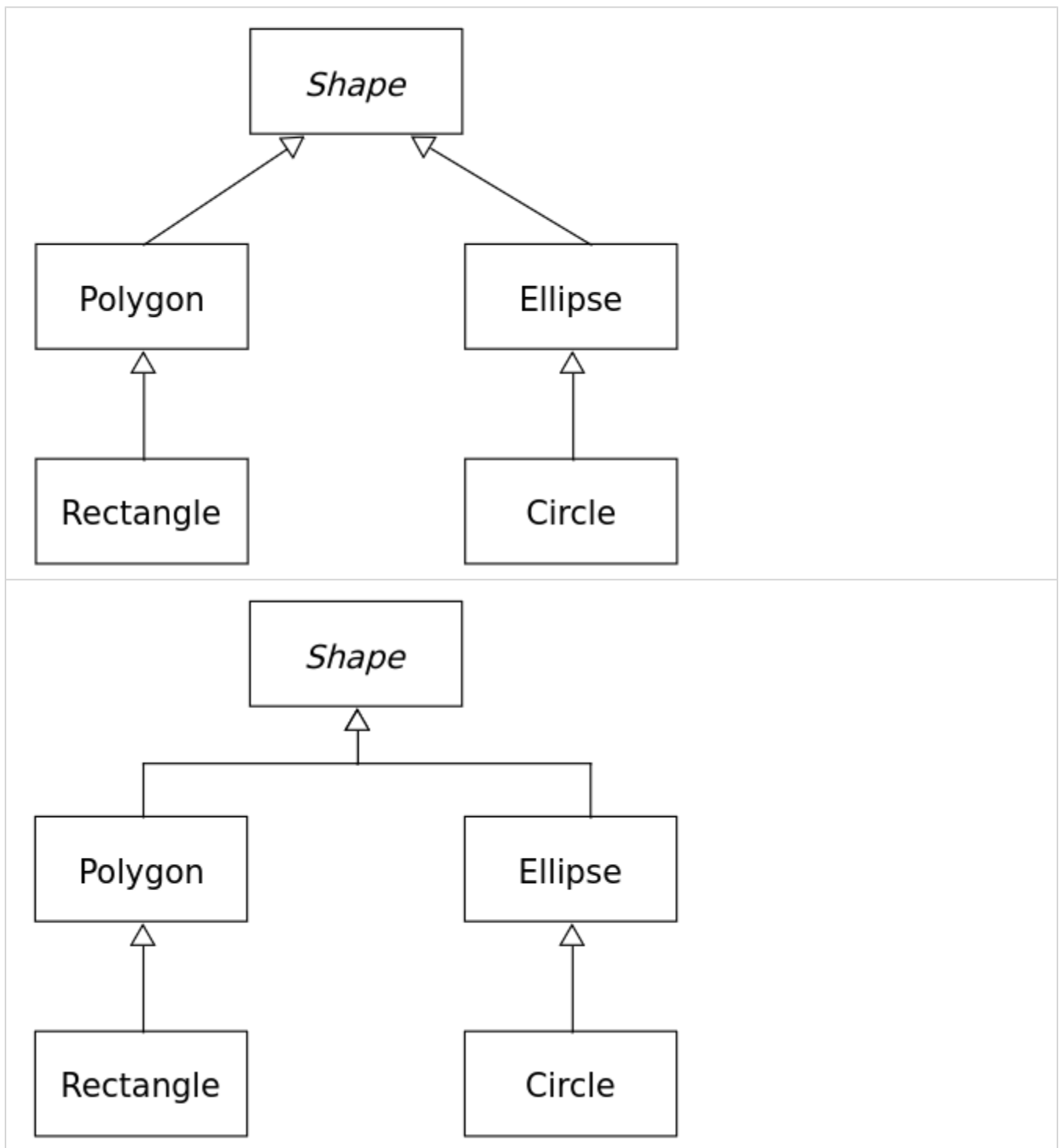
Egy osztályozó egy példánya minden általánosításának példánya.

A speciális osztályozó örökli az általános osztályozó bizonyos tagjait.

Jelölésmód:



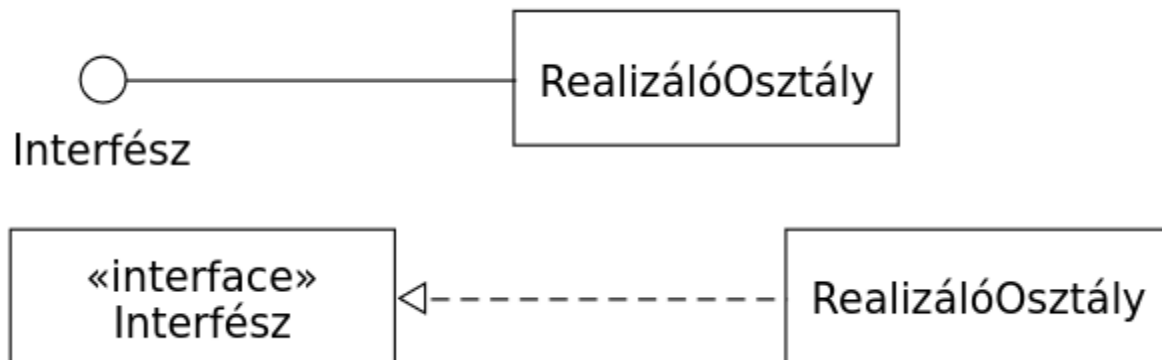
Példa:



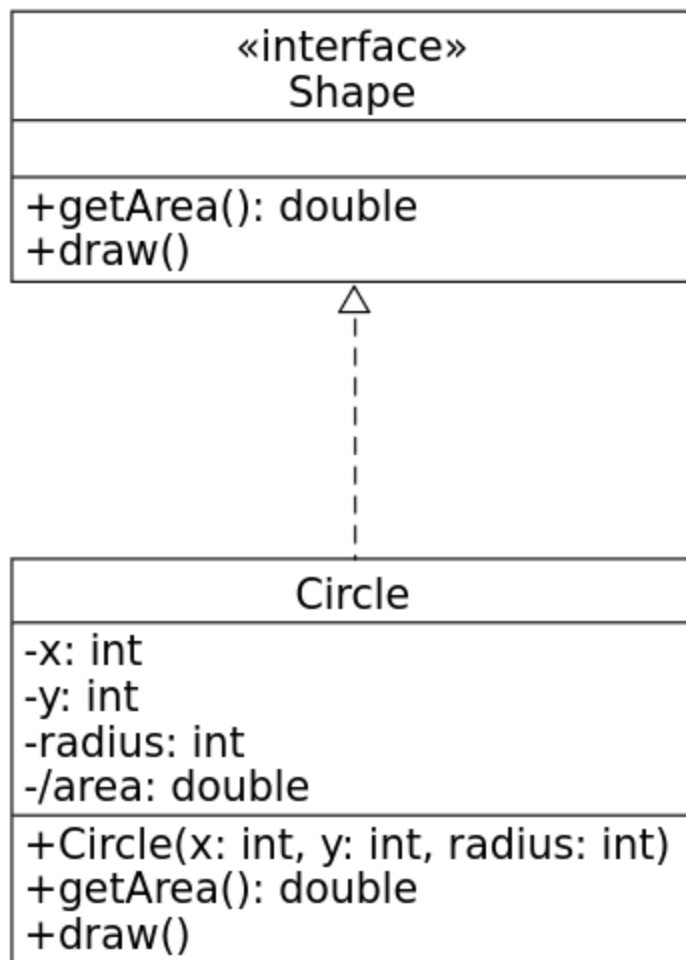
Interfészek

Az interfész egy olyan fajta osztályozó, mely nyilvános jellemzőket és kötelezettségeket deklarál. Az interfész egy szerződést határoz meg. Az interfészek nem példányosíthatók. Osztályozók implementálják vagy az interfész specifikációnak megfelelő nyilvános felületet nyújtanak.

Jelölésmód:



Példa:



2. Szoftvertesztelés

Mi a szoftvertesztelés?

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

Verifikáció és validáció fogalma

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 ügyfelek elvárásainak.

Hibát leíró szakkifejezések: tévedés/tévesztés, hiba, meghibásodás

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.

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.

Például bizonyos hibák nagyon sajátos bemenetek vagy előfeltételek mellett váltanak ki meghibásodást, melyek nagyon ritkán vagy sohasem fordulnak elő.

Nem minden meghibásodást a kódban lévő hibák okoznak, eredményezhetik őket környezeti feltételek is.

Tesztelési alapelvek

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. A tesztek elkopnak (óvakodj a kártevőírtó paradoxontól)
6. A tesztelés környezetfüggő
7. A hibamentesség egy tévhit

Teszteset és tesztadat fogalma

teszteset

ISTQB: Tesztfeltételek alapján meghatározott előfeltételek, bemenetek, tevékenységek (adott esetben), elvárt eredmények és utófeltételek halmaza.

Tesztfeltétel: Egy komponens vagy rendszer tesztelhető vonatkozás, melyet a tesztelés alapjául választunk.

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.

tesztadat

A tesztadatok a tesztvégrehajtáshoz szükséges adatokat jelentik.

Az ilyen konkrét értékek a használatukra vonatkozó világos útmutatásokkal együtt végrehajtható alacsony szintű tesztesetekké teszik a magas szintű teszteseteket. Ugyanaz a magas szintű teszteset különböző tesztadatokat használhat különböző végrehajtásoknál.

Tesztelési szintek: egységtesztelés, integrációs tesztelés, rendszertesztelés, elfogadási

Egységtesztelés/komponens tesztelés (unit testing/componenttesting)

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.

Azonban az automatikus egységtesztek megírása megelőzheti az alkalmazáskód megírását, lásd például a tesztvezérelt fejlesztést (TDD).

Integrációs tesztelés (integration testing)

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.

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 (system testing)

A rendszer egészének (funkcionális és nem funkcionális) viselkedésére összpontosít.

Jellemzően független tesztelők végzik jelentős mértékben specifikációkra támaszkodva.

Elfogadási tesztelés (acceptance testing)

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).

tesztelés (alfa és 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 kiadását elérhetővé teszik kipróbálásra ügyfelek és felhasználók egy nagyobb csoportjának.

A felhasználók helyén történik.

Főleg olyan szoftvertermékekhez alkalmazzák, melyeket sok különböző környezetben használnak.

A marketing egy formája is.

Teszttípusok

funkcionális tesztelés

A rendszer által nyújtott funkciók tesztelése.

Más szóval annak tesztelése, amit a rendszer csinál.

Funkcionális tesztek minden tesztelési szinten ajánlott végezni.

nem funkcionális tesztelés

Rendszerek olyan jellemzőinek értékelése, mint például ahasználhatóság, teljesítmény vagy biztonság.

Más szóval annak tesztelése, hogy a rendszer mennyire jól teszi adolgtát.

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.

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 teszteknek 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

Példa:

```
@Test
public void testPairOfMapEntry() {
    // Arrange:
    final HashMap<Integer, String> map = new HashMap<>();
    map.put(0, "foo");
    final Entry<Integer, String> entry = map.entrySet().iterator().next();

    // Act:
    final Pair<Integer, String> pair = MutablePair.of(entry);

    // Assert:
    assertEquals(entry.getKey(), pair.getLeft());
    assertEquals(entry.getValue(), pair.getRight());
}
```

JUnit

tesztosztályok és tesztmetódusok

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. A `@BeforeAll` és `@AfterAll` annotációkkal jelölt metódusok statikusak kell, hogy legyenek (kivéve azt az esetet, amikor az `@TestInstance(Lifecycle.PER_CLASS)` annotációt használjuk).

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

Tesztmetódusok és életciklus metódusok:

Deklarálhatók az aktuális tesztosztályon belül lokálisan, örökölhettek őssztályból vagy interfészekről.

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 tesztpéldányon történő végrehajtásához a tesztosztályt a `@TestInstance(Lifecycle.PER_CLASS)` annotációval kell megjelölni.

```

import org.junit.jupiter.api.*;

public class LifecycleTest {

    LifecycleTest() {
        System.out.printf("Constructor creates %s\n", this);
    }

    @BeforeAll
    static void beforeAll() { System.out.println("@BeforeAll static method invoked"); }

    @AfterAll
    static void afterAll() { System.out.println("@AfterAll static method invoked"); }

    @BeforeEach
    void beforeEach() { System.out.printf("@BeforeEach method invoked on %s\n", this); }

    @AfterEach
    void afterEach() { System.out.printf("@AfterEach method invoked on %s\n", this); }

    @Test
    void testMethod1() { System.out.printf("testMethod1() method invoked on %s\n", this); }

    @Test
    void testMethod2() { System.out.printf("testMethod2() method invoked on %s\n", this); }

}

```

```

@BeforeAll static method invoked
Constructor creates LifecycleTest@2145433b
  @BeforeEach method invoked on LifecycleTest@2145433b
  testMethod1() method invoked on LifecycleTest@2145433b
  @AfterEach method invoked on LifecycleTest@2145433b
Constructor creates LifecycleTest@fdefd3f
  @BeforeEach method invoked on LifecycleTest@fdefd3f
  testMethod2() method invoked on LifecycleTest@fdefd3f
  @AfterEach method invoked on LifecycleTest@fdefd3f
@AfterAll static method invoked

```

teszteredmények

Siker (success):

Amikor a teszt végrehajtásakor minden tényleges eredmény megegyezik a várt végeredményekkel. Ekkor azt mondjuk, hogy a teszt átmegy (passes).

Bukás (failure):

Amikor a teszt végrehajtásakor a tényleges eredmény nem egyezik meg a várt végeredménnyel. A bukást egy elbukó állítás 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

Kódlefedettségi metrikák

utasítás lefedettség/sor lefedettség

A leggyakrabban használt lefedettségi metrikák az utasítás lefedettség (*statement coverage*) és a sor lefedettség (*line coverage*)

Utasítás lefedettség = Végrehajtott utasítások / Összes utasítás száma

Sor lefedettség = Végrehajtott kódsorok / Összes sor száma

Minden egyes végrehajtott utasítást/sort egyszer számolunk.

A sor lefedettség meghatározásakor csak a végrehajtható kódot tartalmazó sorok kerülnek számolásra.

Vegyük észre, hogy a sor lefedettség függ a forráskód formázástól.

Példa:

```
public static boolean isLongString(String s) {  
    if (s.length() > 5) {  
        return true;  
    }  
    return false;  
}  
  
@Test  
void testIsLongString() {  
    assertFalse(isLongString("abc"));  
}
```

A kódlefedettség $2/4 = 0,5 = 50\%$

Példa:

```
public static boolean isLongString(String s) {  
    return s.length() > 5;  
}  
  
@Test  
void testIsLongString() {  
    assertFalse(isLongString("abc"));  
}
```

A kódlefedettség $1/1 = 1 = 100\%$.

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

Példa:

```
public static String middle(String s) {  
    int i = -1;  
    if ((s.length() & 1) == 1) {  
        i = s.length() / 2;  
    }  
    return s.substring(i, i + 1);  
}  
  
@Test  
void testMiddle() {  
    assertEquals("e", middle("voldemort"));  
}
```

Vegyük észre, hogy az utasítás/sor lefedettség 100%, noha hibás a `middle()` metódus implementációja.

,

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.

Ág lefedettség = Végrehajtott ágak / Összes ág száma

Példa:

```
public static boolean isLongString(String s) {  
    if (s.length() > 5) {  
        return true;  
    }  
    return false;  
}  
  
@Test  
void testIsLongString() {  
    assertFalse(isLongString("abc"));  
}
```

Az ág lefedettség $1/2 = 0,5 = 50\%$

Példa:

```
public static boolean isLongString(String s) {  
    return s.length() > 5;  
}  
  
@Test  
void testIsLongString() {  
    assertFalse(isLongString("abc"));  
}
```

Az ág lefedettség $1/2 = 0,5 = 50\%$.

Példa:

Az ág lefedettség becsapása (az ág lefedettség 100%!)

```
public static int someMethod(int a, int b) {
    int x = 0, y = 0;
    if (a != 0) {
        x = a + 10;
    }
    if (b > 0) {
        y = b / x;
    }
    return y;
}

@Test
void testSomeMethod() {
    assertEquals(2, someMethod(1, 22));
    assertEquals(0, someMethod(0, -15));
}
```

Példa:

Az ág lefedettség becsapása (az ág lefedettség 100%!)

```
public static int someMethod(int a, int b) {
    int x = 0, y = 0;
    if (a != 0) {
        x = a + 10;
    }
    if (b > 0) {
        y = b / x;
    }
    return y;
}
```

Vegyük észre, hogy a `someMethod(0, 10)` metódushívás egy `ArithmeticException` kivételt eredményez.

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.

Mi a tesztvezérelt fejlesztés (TDD)?

A tesztvezérelt fejlesztés (*test driven development, TDD*) egy szoftverfejlesztési folyamat, mely az automatikus tesztek megírását bármiféle kód megírása elé helyezi.

Egy iteratív megközelítés, mely egy automatizált teszt keretrendszer (például *JUnit*) használatán és a következő rövid fejlesztési ciklus ismétlésén alapul:

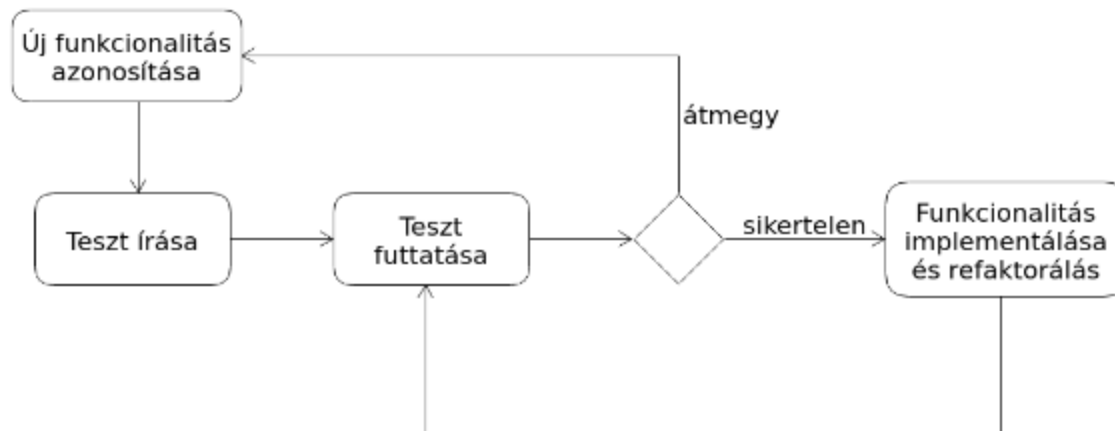
1. Írj egy tesztet (Write a test)
2. Érd el, hogy működjön (Make it run)
3. Javítsd ki (Make it right)

A TDD mantrája:

1. Vörös: írd meg egy nem működő kis tesztet, mely elsőre talán le sem fordul.
2. Zöld: gyorsan javítsd ki a tesztet, közben bármilyen bűnt elkövethetsz, ami szükséges.
3. Refaktorálj: távolítsd el minden ismétlődést, mely azért jött létre, hogy a teszt működjön.

A cél működő tiszta kód.

A TDD folyamat



A TDD előnyei

Megkönnyíti a kód írását:

A TDD segíti a programozót abban, hogy tisztázza a gondolatait arról, hogy mit kellene, hogy csináljon egy kódrész. Egy teszt írásához a feladat megértése szükséges, a megértés pedig könnyebbé teszi a szükséges kód megírását.

Kódlefedettség:

Minden kódrésznek kell, hogy legyen tesztje.

Könnyebb hibakeresés:

Amikor egy teszt sikertelen, nyilvánvaló kell, hogy legyen a probléma forrása.

Dokumentálás:

A tesztek maguk is egyfajta dokumentációnak tekinthetők, mely leírja, hogy mit kellene, hogy csináljon a kód.

Leginkább új szoftverek kifejlesztéséhez alkalmas.

3. Objektorientált tervezési alapelvek

Statikus kódelemzés fogalma, példák statikus kódelemző eszközökre

A statikus kódelemzés a programkód elemzésének folyamata mely a kód végrehajtása nélkül történik.

Az elemzés irányulhat hibák észlelésére; annak ellenőrzésére, hogy a kód megfelel-e egy kódolási szabványnak, ...

Statikus kódelemző: statikus kódelemzést végző automatikus eszköz.

Nyelv	Kódelemző
C#	<i>InferSharp, Roslynator</i>
C++	Cppcheck
ECMAScript / JavaScript	<i>ESLint, JSHint, JSLint, RSLint</i>
Java	<i>Checkstyle, Error Prone, NullAway, SpotBugs</i>
Python	<i>Prospector, Pylint</i>
Több nyelvet támogató eszközök	<i>Coala, Infer, PMD, Semgrep</i>

A DRY elv

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

- A tudás minden darabkájának egyetlen, egyértelmű, hiteles reprezentációja kell, hogy legyen egy rendszerben.
- Az ellenkezője a **WET**.
- „We enjoy typing”, „write everything twice”, „waste everyone's time”

Az ismétlések fajtái:

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

Kapcsolódó fogalom: kódismétlés (code duplication), copy-and-paste programming

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

Nem minden kódismétlés információ ismétlés!

PMD támogatás: Copy/Paste Detector (CPD)

- Finding duplicated code with CPD
- Támogatott programozási nyelvek: C/C++, C#, ECMAScript (JavaScript), Java, Kotlin, Python, ...

IntelliJ IDEA:

- Analyze duplicates

A DRY elv megsértései nem mindig kódismétlés formájában jelennek meg.

A DRY elv az információk megismétléséről szól. A tudás egy darabkája két teljesen eltérő módon is kifejezhető két különböző helyen.

Példa (Thomas & Hunt, 2019):

```
class Line {  
    Point start;  
    Point end;  
    double length; // a DRY elv megsértése  
}
```

Az elv megsértése kiküszöbölhető a length adattag egy **metódusra** való kicserélésével:

```
class Line {  
    Point start;  
    Point end;  
    double length() {  
        return start.distanceTo(end);  
    }  
}
```


A jobb teljesítmény érdekében választható a DRY elv megsértése. Ilyenkor az elv megszegését ajánlott a külvilág elől elrejteni.

```
class Line {
    private Point start;
    private Point end;
    private double length;

    public Line(Point start, Point end) {
        this.start = start;
        this.end = end;
        calculateLength();
    }
    public void setStart(Point p) {
        this.start = p;
        calculateLength();
    }
    public void setEnd(Point p) {
        this.end = p;
        calculateLength();
    }
    public Point getStart() { return start; }

    public Point getEnd() { return end; }

    public double getLength() { return length; }

    private void calculateLength() { this.length = start.distanceTo(end); }
}
```

Reprezentációs ismétlés (Thomas & Hunt, 2019):

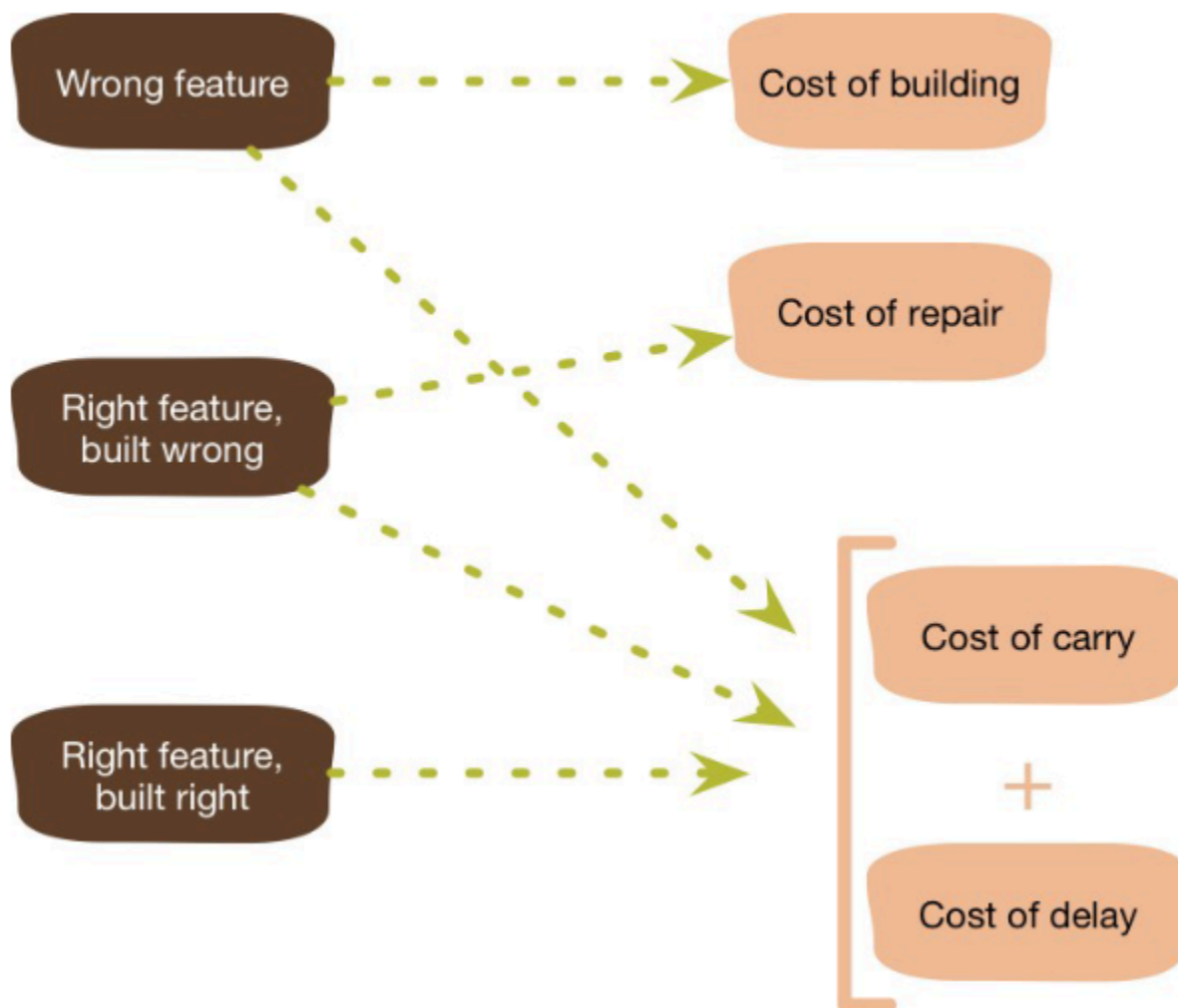
A kód gyakran függ a külvilágtól, például *API*-kon keresztül más programkönyvtáraktól, külső adatforrások adataitól, mely mindig a *DRY* elv valamiféle megsértését vonja maga után, a kódnak olyan tudással kell rendelkeznie, mely a külső dologban is ott van. Ismernie kell az *API*-t, a sémát, vagy a hibakódok jelentését. Ez az ismétlés elkerülhetetlen. Eszközök, melyek segítenek megbirkózni az ilyen fajta ismétlésekkel: Sémákból kódot generáló eszközök (például *JAXB*, *JPA*, *OpenAPI*)

A YAGNI elv

A „You Aren't Gonna Need It” („nem lesz rá szükséged”) rövidítése.
Az extrém programozás (XP) egy alapelve.

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

Egy olyan lehetőség kifejlesztésének költségei, mely jelenleg nem szükséges



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.

A KISS elv

Keep it simple, stupid

Az egyszerűségekre való törekvés:

„Everything should be made as simple as possible, but not simpler.”

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.

GoF alapelvek

A két GoF alapelv:

- Interfészre programozunk, ne implementációra.
- Lásd például a létrehozási mintákat.
- Részesítsük előnyben az objektum-összetételt az öröklődéssel szemben.

SOLID alapelvek (magyarázat)

- **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 osztálynak csak egy oka legyen a változásra.

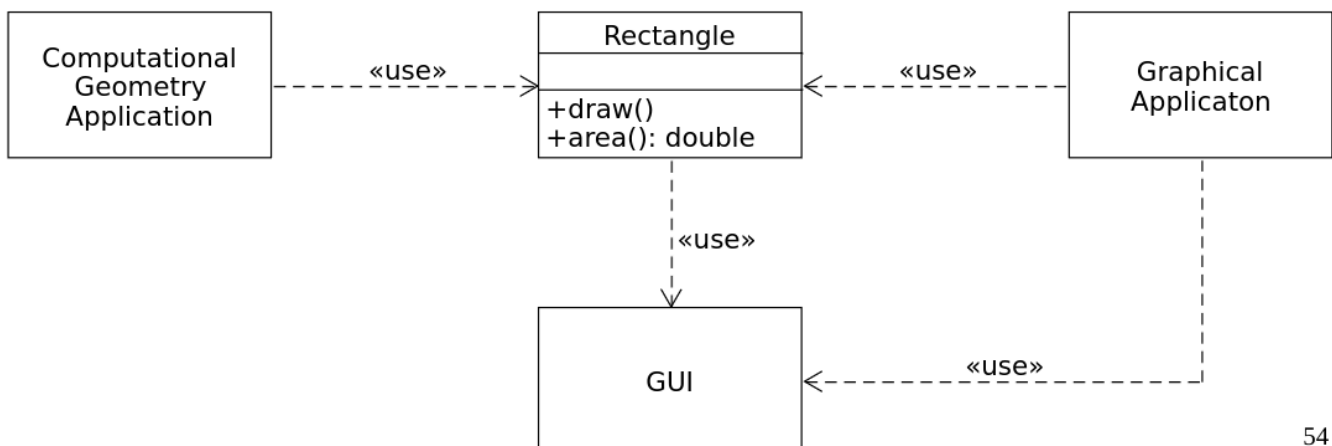
Egy felelősség egy ok a változásra.

- 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.

Példa az elv megsértésére:

A Rectangle osztály két felelőssége:

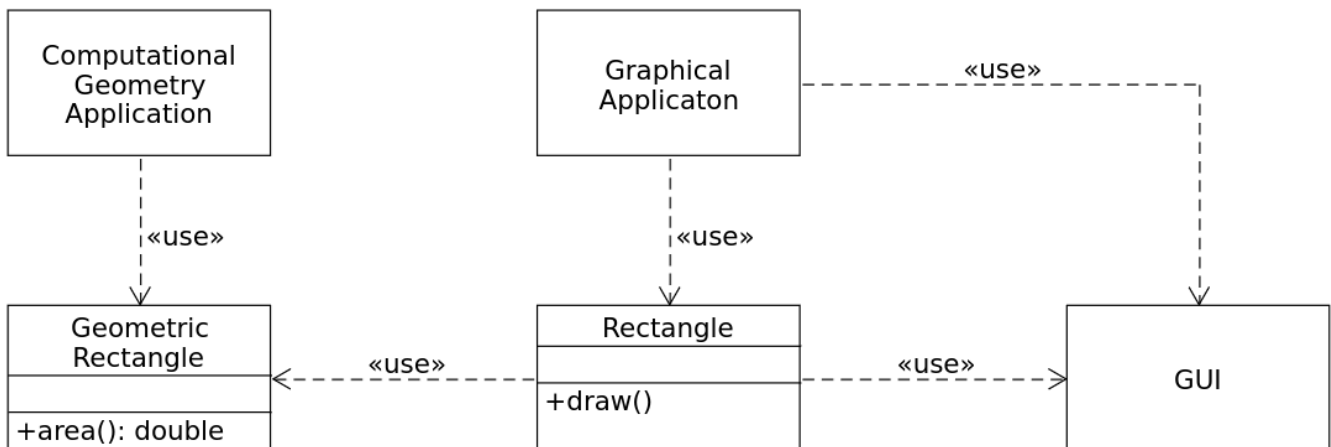
- Egy téglalap geometriájának matematikai modellezése.
- Téglalap megjelenítése a grafikus felhatalnáló felületen.



Problémák:

- A számítógépes geometriai alkalmazásnak tartalmaznia kell a grafikus felhasználói felületet.
- Ha a grafikus alkalmazás miatt változik a *Rectangle* osztály, az szükségessé teheti a számítógépes geometriai alkalmazás összeállításának, tesztelésének és telepítésének megismétlését (*rebuild*, *retest*, *redesploy*).

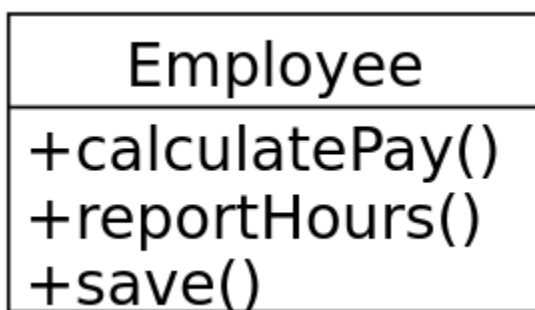
Az előbbi példa az elvnek megfelelő változata:



Példa (Robert C. Martin):

Az alábbi *Employee* osztály megszegi az egyszeres felelősség elvét, mivel a három metódus nagyon különböző aktoroknak van alávetve:

- `calculatePay()` : a bérosztály határozza meg
- `reportHours()` : a munkaügyi osztály határozza meg
- `save()` : az adatbázis adminisztrátorok határozzák meg



nyitva zárt elv

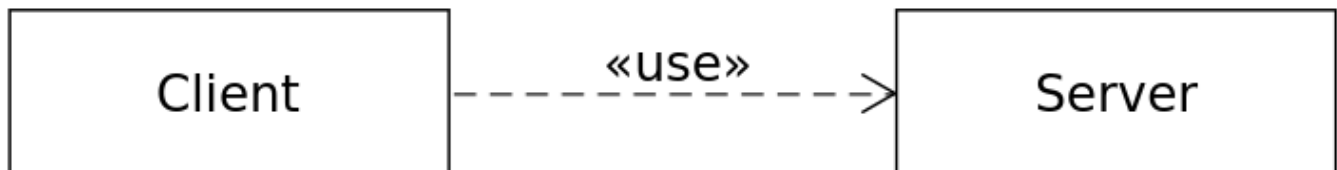
A szoftver entitások (*osztályok, modulok, függvények, ...*) legyenek nyitottak a bővítésre, de zártak a módosításra.

Az elvnek megfelelő modulnak két fő jellemzője van:

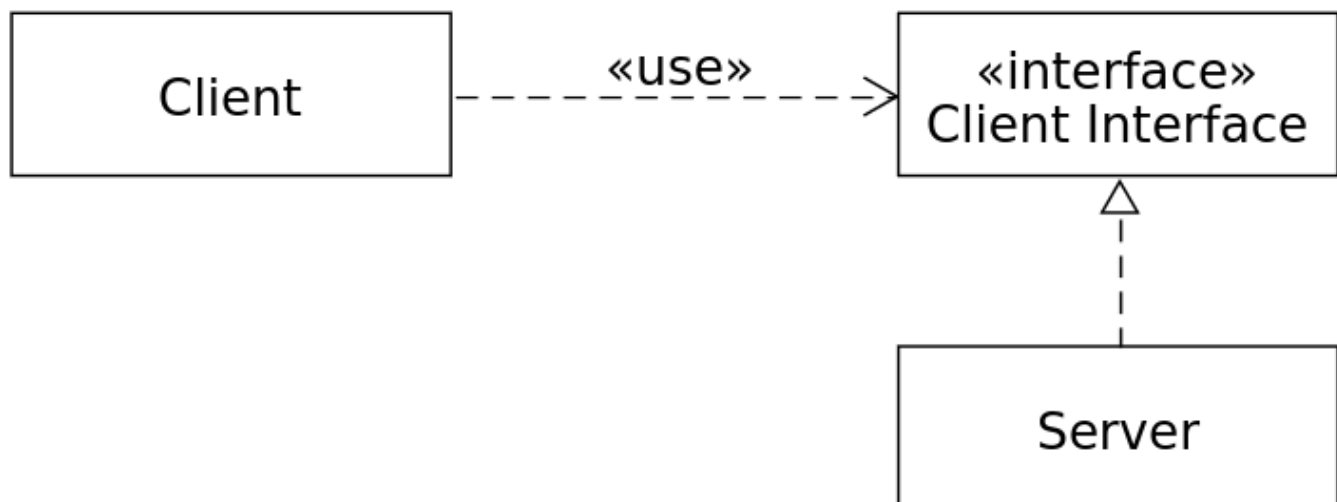
- Nyitott a bővítésre: azt jelenti, hogy a modul viselkedése kiterjeszthető.
- Zárt a módosításra: azt jelenti, hogy a modul viselkedésének kiterjesztése nem eredményezi a modul forrás- vagy bináris kódjának változását.

Példa az elv megsértésére:

A *Client* és a *Server* konkrét osztályok. A *Client* osztály a *Server* osztályt használja. Ha azt szeretnénk, hogy egy *Client* objektum egy különböző szerver objektumot használjon, a *Client* osztályban meg kell változtatni a szerver osztály nevét.



Az előbbi példa az elvnek megfelelő változata:



Liskov-féle helyettesítési elv

Ha az *S* típus a *T* típus altípusa, nem változhat meg egy program működése, ha benne a *T* típusú objektumokat *S* típusú objektumokkal helyettesítjük.

This means that every subclass or derived class should be substitutable for their base or parent class.

interfész szétválasztási elv

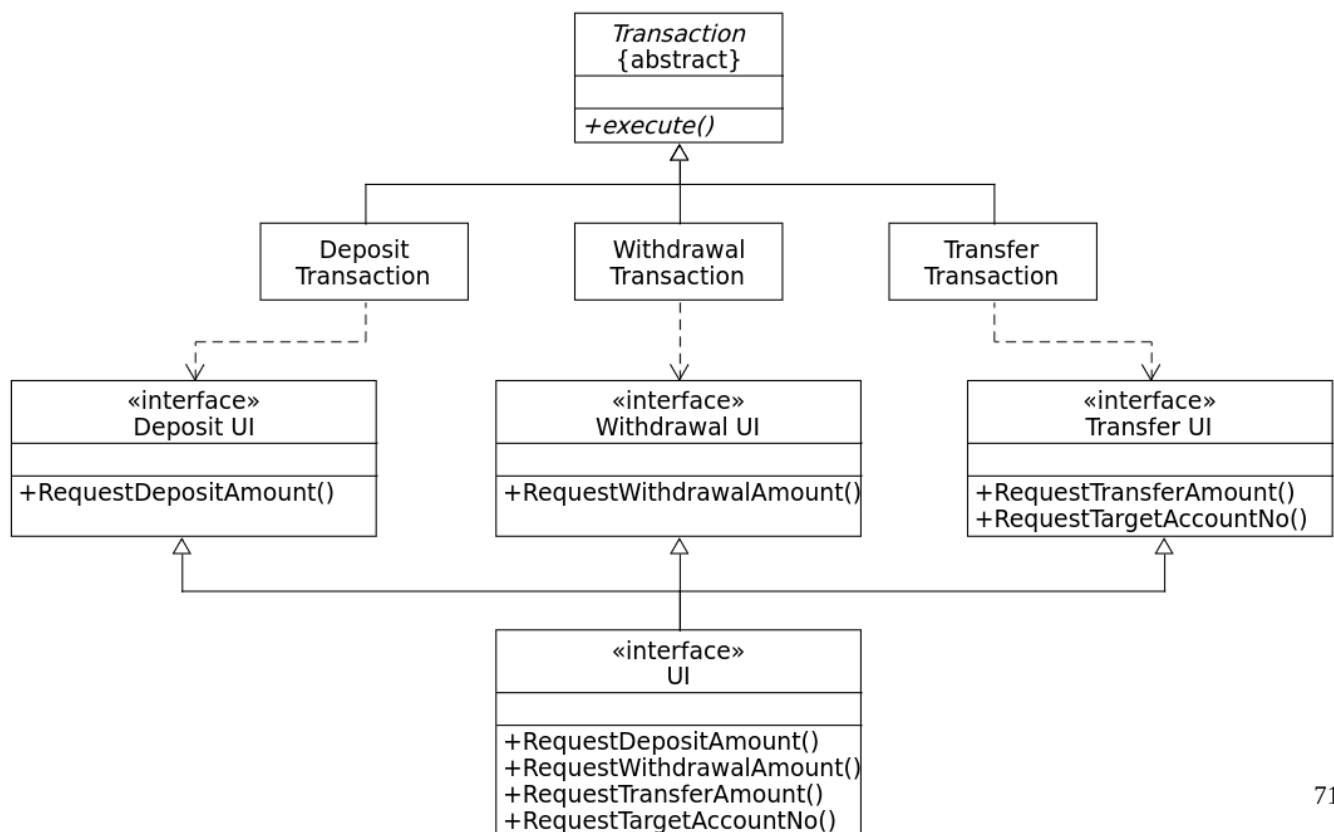
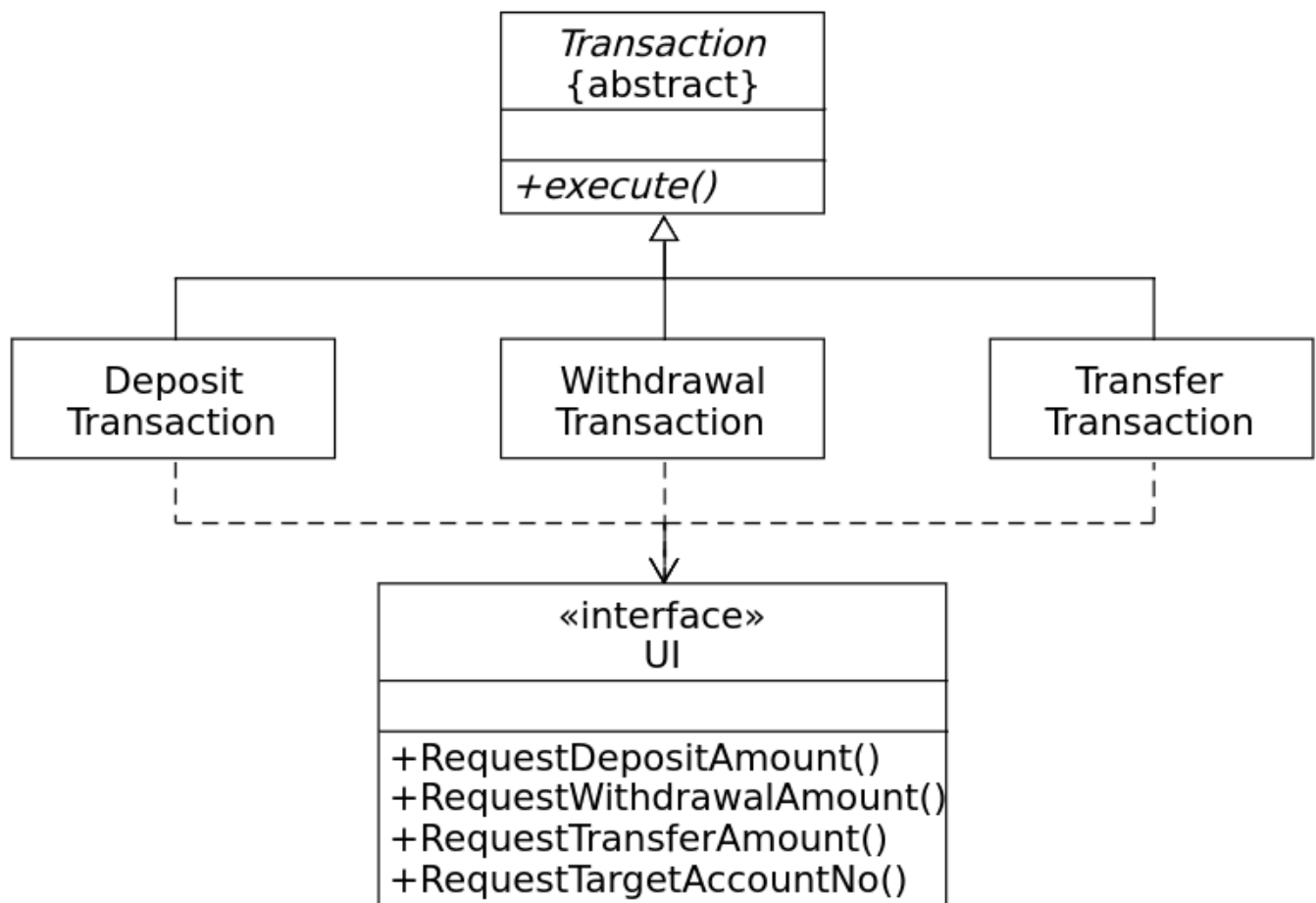
Nem szabad arra kényszeríteni az osztályokat, hogy olyan metódusoktól függjenek, melyeket nem használnak.

Vastag interfész (fat interface):

- Az ésszerűen szükségesnél több tagfüggvénnyel és baráttal rendelkező interfész.
- 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

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 függnek az alacsony szintű moduloktól.

A magas szintű modulok tartalmazzák az alkalmazás üzleti logikáját, ők adják az alkalmazás identitását.

Ha ezek a modulok alacsony szintű moduloktól függnek, akkor az alacsony szintű modulokban történő változásoknak közvetlen hatása lehet a magas szintű modulokra, szükségessé tehetik azok változását is.

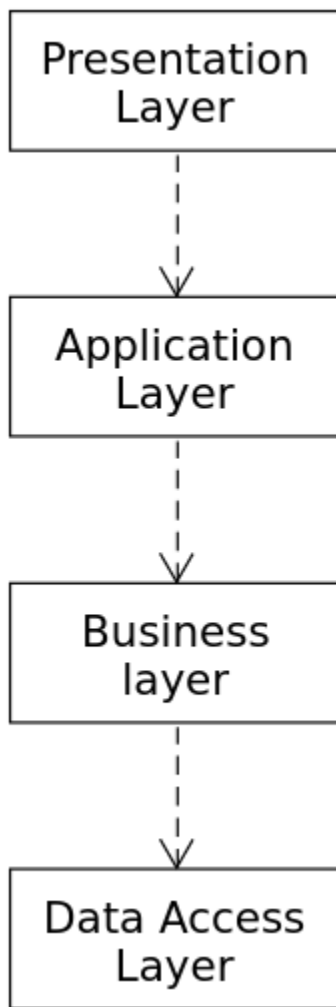
Ez abszurd! A magas szintű modulok azok, melyek meg kellene, hogy határozzák az alacsony szintű modulokat.

A magas szintű modulokat szeretnénk újrafelhasználni. Az alacsony szintű modulok újrafelhasználására elég jó megoldást jelentenek a programkönyvtárak.

Ha magas szintű modulok alacsony szintű moduloktól függnek, akkor nagyon nehéz az újrafelhasználásuk különféle helyzetekben.

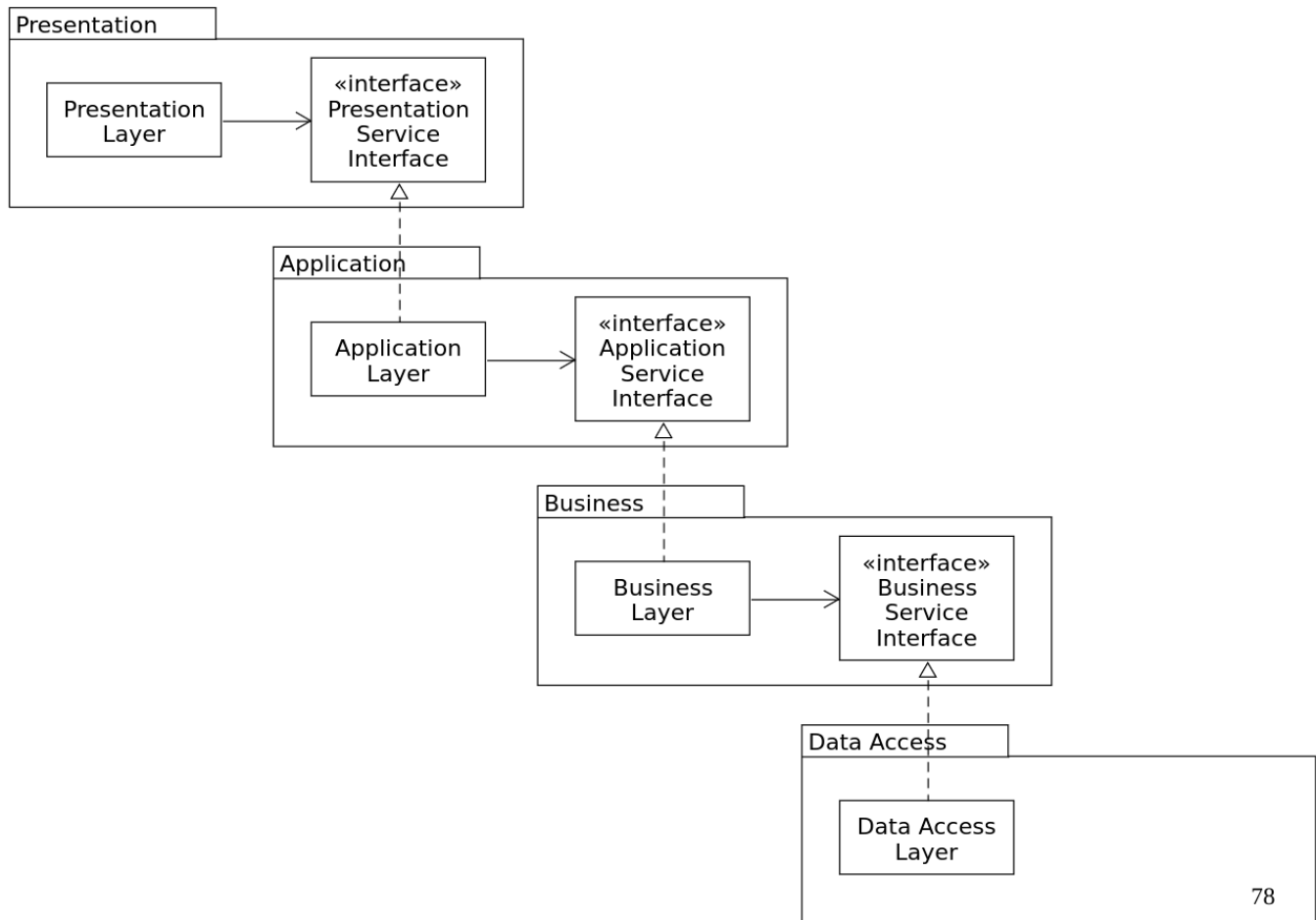
Ha azonban a magas szintű modulok függetlenek az alacsony szintű moduloktól, akkor elég egyszerűen újrafelhasználhatók.

Példa a rétegek architektúrális minta hagyományos alkalmazására:



Az előbbi példa az elvnek megfelelő változata:

- 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.



- Nem csupán a függőségek kerültek megfordításra, hanem az interfész tulajdonlás is (inversion of ownership).
- **Hollywood elv:** Ne hívj, majd mi hívunk. (Don't call us, we'll call you.)

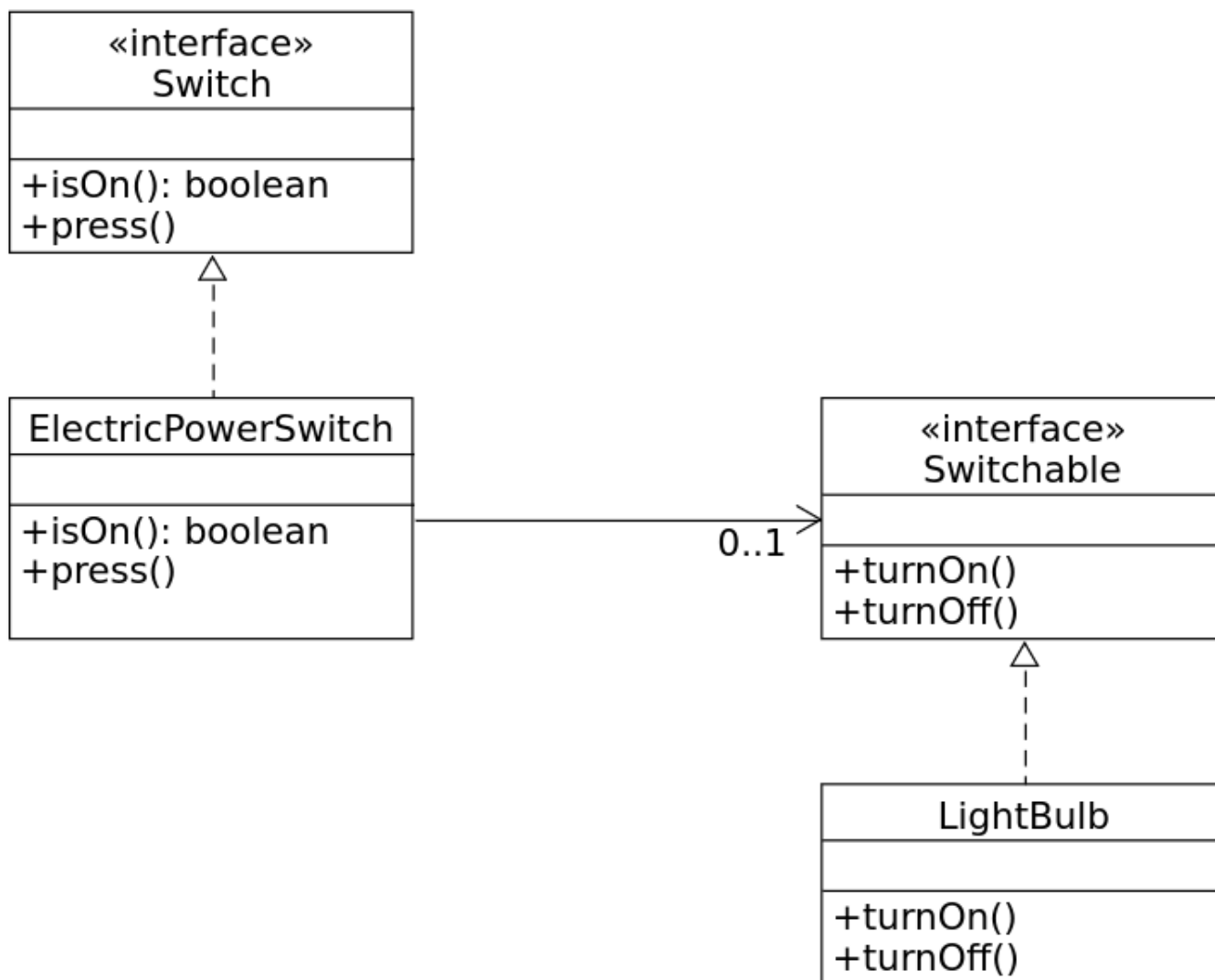
Függés absztrakcióktól:

- Ne függjön a program konkrét osztályoktól, hanem inkább csak absztrakt osztályoktól és interfészekről.
- 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.

Példa az elv megsértésére:



Az előbbi példa az elvnek megfelelő változata:



Függőség befecskendezés

A vezérlés megfordítása (*IoC - inversion of control*) nevű architektúrális 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 függőség befecskendezés olyan szoftvertervezési elvek és minták összessége, melyek lazán csatolt kód fejlesztését teszik lehetővé.
- 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.

Példa: (nincs függőség befecskendezés)

```
public interface SpellChecker {
    boolean check(String text);
}
public class TextEditor {
    private SpellChecker spellChecker;

    public TextEditor() {
        spellChecker = new HungarianSpellChecker();
    }
    // ...
}
```

Példa: (függőség befecskendezés konstruktorral)

```
public class TextEditor {
    private SpellChecker spellChecker;

    public TextEditor(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }
    // ...
}
```

Példa: (függőség befecskendezés beállító metódussal) (setter injection):

```
public class TextEditor {
    private SpellChecker spellChecker;
    public TextEditor() {}
    public void setSpellChecker(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }
    // ...
}
```

Példa: (függőség befecskendezés interfésszel)

```
public interface SpellCheckerSetter {
    void setSpellChecker(SpellChecker spellChecker);
}
public class TextEditor implements SpellCheckerSetter {
    private SpellChecker spellChecker;

    public TextEditor() {}

    @Override
    public void setSpellChecker(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }
    // ...
}
```

A függőség befecskendezés előnyei:

- Kiterjeszthetőség
 - Karbantarthatóság
 - Tesztelhetőség: a függőség befecskendezés támogatja az egységtesztelést.
 - Valós függőségek helyet a tesztelt rendszerbe befecskendezhetők „teszt dublőrök” (test doubles).
-

4. Minták a szoftverfejlesztésben

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.”

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

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

Architekturális minták, a modell-nézet vezérlő (MVC) architektúrális minta

Az architektúrális 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. [POSA1]

- Példák: mikrokernel, modell-nézet-vezérlő

MVC:

Név: Modell-nézet vezérlő (Model-View-Controller)

Környezet: Rugalmas ember-gép felülettel rendelkező interaktív alkalmazások.

Probléma: Különösen gyakori az igény a felhasználói felületek változtatására.

Erők:

Ugyanaz az információ 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

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.

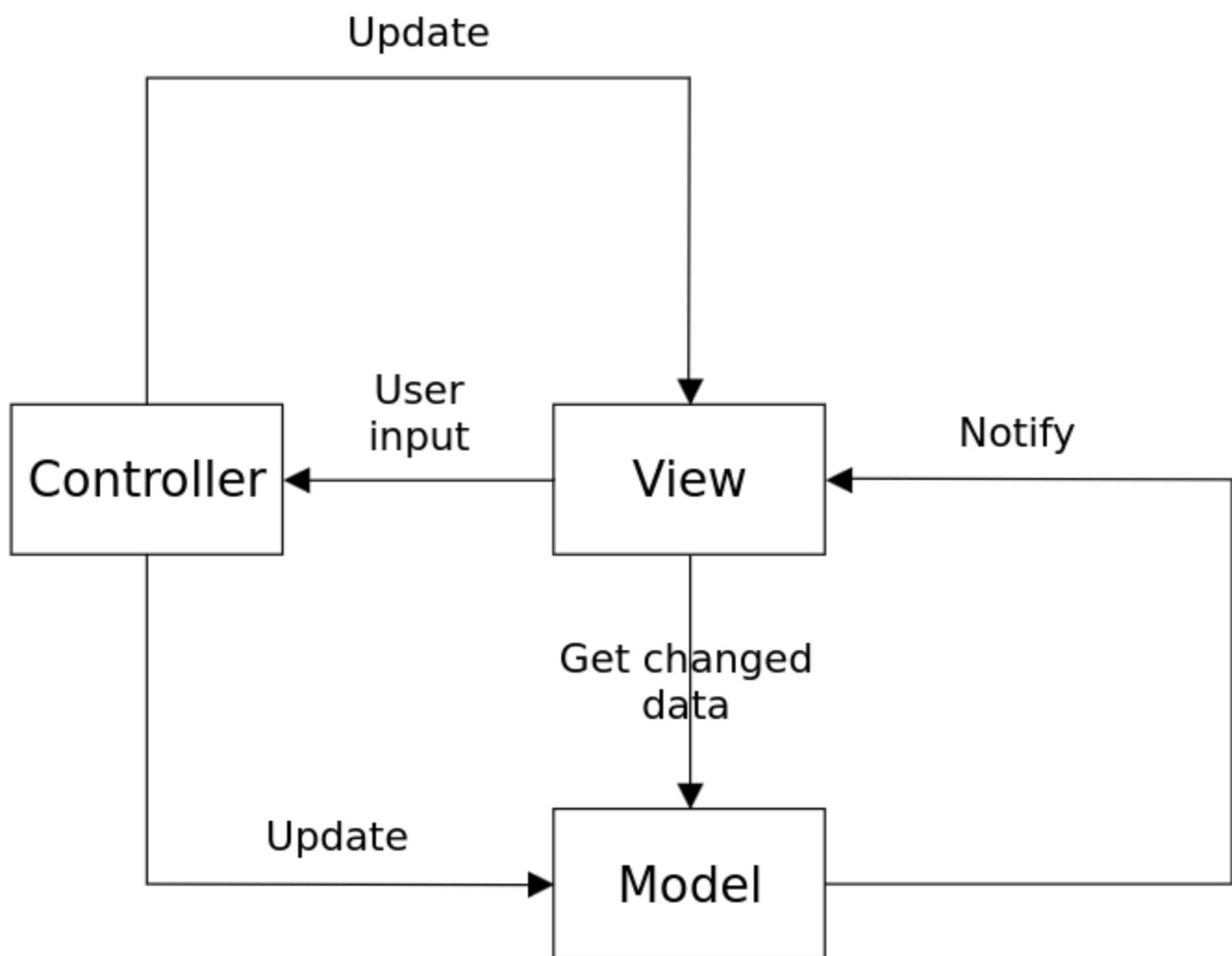
Lehetővé tesz ugyanahhoz a nézethez akár több vezérlőt is.

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, tervezési minták 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.

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

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.

A GoF könyv 23 tervezési mintát ír le, időközben számos további tervezési minta született.

- Például: függőség befecskendezés (dependency injection), tároló (repository), többke (multiton), iker (twin), folyékony interfész (fluent interface), ...

Új kategória: Konkurencia minták (Concurrency Patterns)

- Például: aktív objektum (active object), őrzött felfüggesztés (guarded suspension), szálkészlet (thread pool), ...

Mintasablon (GoF):

- Név és besorolás (Name and Classification):
- Cél (Intent):
- Más néven (Also Known As):
- Indíték (Motivation):
- Alkalmazhatóság (Applicability):
- Szerkezet (Structure):
- Részrtvevők (Participants):
- Együttműködés (Collaborations):
- Következmények (Consequences):
- Megvalósítás (Implementation):
- Példakód (Sample Code):
- Ismert felhasználások (Known Uses):
- Kapcsolódó minták (Related Patterns):

Létrehozási minták

elvont gyár (abstract factory):

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

Ismert felhasználások:

- `java.sql.Connection`
- `javax.xml.datatype.DatatypeFactory`
- `javax.xml.transform.TransformerFactory`
- `javax.xml.stream.XMLInputFactory`

egyke:

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

Ismert felhasználások:

- java.io.Console
- java.lang.Runtime
- java.time.chrono.IsoChronology

építő:

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

Ismert felhasználások:

- A java.lang.Appendable interfész minden implementációja
 - Például java.lang.StringBuilder
- java.lang.ProcessBuilder
- java.time.format.DateTimeFormatterBuilder

Példakód: java.util.StringJoiner

```
var s = new StringJoiner(",", "[", "]")
    .add("John")
    .add("Paul")
    .add("George")
    .add("Ringo")
    .toString(); // "[John,Paul,George,Ringo]"
```

objektumkészlet (object pool):

Cél: Inicializált objektumok egy halmazát tartja nyilván az igények kiszolgálásához, ahelyett, hogy létrehozná és megsemmisítené az objektumokat.

Ismert felhasználások:

- Apache Commons Pool (licenc: Apache License 2.0)
- Vibur Object Pool (licenc: Apache License 2.0)
- Adatbázis kapcsolatok gyorsítótárazása (connection pooling):
 - Apache Commons DBCP (licenc: Apache License 2.0)
 - Vibur DBCP (licenc: Apache License 2.0)

Szerkezeti minták: díszítő, illesztő

- 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 interfészerű osztályok együttműködését biztosíthatjuk.

Ismert felhasználások:

- java.io.InputStreamReader
- java.io.OutputStreamWriter
- jakarta.xml.bind.annotation.adapters.XmlAdapter

Példakód: java.io.InputStreamReader (OpenJDK 21)

```
public class InputStreamReader extends Reader {
    private final StreamDecoder sd;

    public InputStreamReader(InputStream in) {
        super(in);
        sd = StreamDecoder.forInputStreamReader(in, this,
            Charset.defaultCharset()); // ## check lock object
    }
    // ...
}
```


Díszítő

Cél:

- Az objektumokhoz dinamikusan további felelősségi köröket rendel.
- A kiegészítő szolgáltatások biztosítása terén e módszer rugalmas alternatívája az alosztályok létrehozásának.

Ismert felhasználások:

- `java.io.InputStream`
- `java.io.OutputStream`
- `java.util.Collections#unmodifiableXXX()`

Viselkedési minták: sablonfüggvény, megfigyelő

- 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)

Megfigyelő

Cél: Objektumok között egy sok-sok függőségi kapcsolatot létrehozni, így amikor az egyik objektum állapota megváltozik, minden tőle függő objektum értesül erről és automatikusan frissül.

Ismert felhasználások:

- `java.util.EventListener`
- `javafx.beans.Observable`
- `java.util.concurrent.Flow`

Sablonfüggvény

Cél:

- Egy adott művelet algoritmusának vázát elkészíteni, amelynek egyes lépéseit alosztályokra ruházzuk át.
- Így az alosztályok az algoritmus egyes lépéseit felülbírálnak, anélkül, hogy az algoritmus szerkezete módosulna.

Ismert felhasználások:

- `java.io.InputStream`
- `java.io.OutputStream`
- `java.util.ArrayList`
- `java.util.HashMap`
- `java.util.AbstractQueue`

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 eszköztárával.

A legtöbb idióma nyelvspecifikus, létező programozási tapasztalatot hordoznak.

Példa:

```
import java.util.Objects;

public final class Movie implements Cloneable {
    private String title;
    private int year;

    public Movie(String title, int year) {
        if (year < 1878) {
            throw new IllegalArgumentException();
        }
        this.title = title;
        this.year = year;
    }

    @Override
    public boolean equals(Object o) {
        if (o == this) {
            return true;
        }
        if (!(o instanceof Movie)) {
            return false;
        }
        Movie m = (Movie) o;
        return Objects.equals(title, m.title) && year == m.year;
    }

    // Error: no hashCode() and clone() methods are provided!
}
```

Példa: mintaillesztés az instanceof operátorhoz (Java SE 16)

```
import java.util.Objects;

public final class Movie implements Cloneable {
    private String title;
    private int year;

    public Movie(String title, int year) {
        if (year < 1878) {
            throw new IllegalArgumentException();
        }
        this.title = title;
        this.year = year;
    }

    @Override
    public boolean equals(Object o) {
        if (o == this) {
            return true;
        }
        return (o instanceof Movie m) && Objects.equals(title, m.title) && year == m.year;
    }

    // Error: no hashCode() and clone() methods are provided!
}
```

Antiminták, a massa és a spagetti kód antiminta

„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.

Antiminta sablon:

Antiminta neve: Az antiminta egyedi (pejoratív) neve.

Más néven: Az antiminta más ismert elnevezései.

Leggyakoribb előfordulási szint: A szoftvertervezési modell mely szintjén fordul elő jellemzően az antiminta. A következő kulcsszavak kerülhetnek ide: idióma, mikroarchitektúra, keretrendszer, alkalmazás, rendszer, vállalat, globális/ipar.

Újragyártott megoldás neve: Az újragyártott megoldási sémát azonosítja.

Újragyártott megoldás típusa: Azt jelzi, hogy milyen fajta tevékenységet jelent az antiminta megoldása. A következő kulcsszavak kerülhetnek ide: szoftver, technológia, folyamat, szerep

Kiváltó okok: Az antiminta kiváltó okait megadó kulcsszavak, melyek a következők lehetnek: sietség, fásultság, szűklátókörűség, lustaság, fösvénység, tudatlanság, büszkeség, elhanyagolt felelősség.

Kiegyensúlyozatlan erők: Azokat a tényezőket megadó kulcsszavak, melyeket figyelmen kívül hagytak, rosszul használtak vagy túl sokszor használtak a mintában. A választási lehetőségek közé tartoznak a következők: a funkcionalitás kezelése, a teljesítmény kezelése, a bonyolultság kezelése, a változás kezelése, az IT erőforrások kezelése, a technológia-transzfer kezelése.

Anekdotaszerű példa: Opcionális rész, mely az antimintához kötődő ismert mondásokat tartalmaz.

Háttér: Opcionális rész, mely további példákat tartalmazhat a probléma előfordulási helyeiről vagy további hasznos vagy érdekes általános háttérinformációkat.

Általános alak: Az antiminta általános leírása, mely gyakran tartalmaz ábrát is. Nem egy példa, hanem egy általános változat.

Tünetek és következmények: Az antiminta tüneteinek és az antiminta általa okozott következmények felsorolása.

Tipikus okok: Az antiminta okainak felsorolása.

Ismert kivételek: Azokat a kivételes eseteket írja le ez a rész, melyeknél az antiminta nem káros.

Újragyártott megoldás: Az általános alaknál megfogalmazott antimintára adott megoldást ismerteti ez a rész lépésekre bontva.

Változatok: Opcionális rész, mely az antiminta általános alakjának esetleges változatait sorolja fel és fejti ki.

Példa: Ez a rész szemlélteti a megoldás a problémára való alkalmazását. Rendszerint megjelenik itt a probléma sematikus ábrája, a probléma leírása, a megoldás sematikus ábrája, a megoldás leírása.

Kapcsolódó megoldások: Az adott antimintához szorosan kapcsolódó tervezési minták és antiminták kerülnek itt felsorolásra, valamint kifejtésre kerülnek a különbségek.

Alkalmazhatóság más nézőpontokra és szintekre: Hogyan befolyásolja a minta a többi nézőpontot: vezetési, architekturális, fejlesztési. Ugyancsak itt kerül leírásra, hogy milyen mértékben releváns a minta más szintek szempontjából.

massza

Antiminta neve: a massa (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 massa 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 massa 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 rendszeré.

Tünetek és következmények:

- Egyetlen osztály nagyszámú attribútummal, művelettel vagy mindkettővel. Általában a massa 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 massa antiminta elfogadható kompatibilitási okokból megtartott korábbi rendszer becsomagolásakor.

Újragyártott megoldás:

- A megoldás kódújrászervezéssel jár.
- Összetartozó attribútumok és műveletek csoportjainak azonosítása.
- Természetes helyet kell keresni ezen funkcionalitás- csoportok 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 massa antiminta megelőzésében.
- **Alkalmazhatóság más nézőpontokra és szintekre:** Az architekturális és vezetői nézőpontok is kulcsszerepet játszanak a massa antiminta megelőzésében.

spagetti kód

Antiminta neve: spagetti kód (Spaghetti Code)

Leggyakoribb előfordulási szint: alkalmazás

Újragyártott megoldás neve: kódújrászervezés, kódtisztítás

Újragyártott megoldás típusa: szoftver

Kiváltó okok: tudatlanság, lustaság

Kiegyensúlyozatlan erők: a bonyolultság, a változás kezelése

Anekdotaszerű példa:

- „Ó! Micsoda zűrzavar!”
- „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 kódújrászervezés. Kapcsolódó megoldások: analízis-paralízis, lávafolyás
-

5. Tiszta kód

Milyen a tiszta kód?

Szeretem, ha a kódom elegáns és hatékony. A logikája egyszerű kell, hogy legyen, hogy nehezen bújjanak meg a hibák a kódban. A függőségek legyenek minimálisak a könnyű karbantarthatóság érdekében. A hibakezelés legyen teljes, a teljesítmény pedig közel optimális, hogy ne kísértsen a kódot piszkító optimalizálásra. A tiszta kód egy dolgot csinál, és azt jól.

A tiszta kód egyszerű és közvetlen. Olyan a tiszta kódot olvasni, mint a jól megírt prózát. A tiszta kód soha nem homályosítja el a tervezője szándékát, hanem inkább éles absztrakciókkal és egyértelmű sorokkal teli.

A tiszta kód egy olyan fejlesztő számára is olvasható, továbbfejleszthető, aki nem az eredeti szerző. Vannak hozzá egység- és elfogadási tesztek. Értelmes nevei vannak. Egy dolog elvégzéséhez egy módot biztosít több helyett. Minimális függőségei vannak, melyek világosan meghatározottak. Tiszta és minimális API-t biztosít. A kód literate kell, hogy legyen, mivel a nyelvtől függően nem minden szükséges információ fejezhető ki világosan pusztán a kódban.

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;
```

Egy nagyon rossz példa:

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<int[]>();  
    for (int[] x : theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```

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.

Példa nem informatív nevekre:

```
public static void copyChars(char a1[],char a2[]) {  
    for (int i = 0; i < a1.length; i++) {  
        a2[i] = a1[i];  
    }  
}
```

A fenti kódban használjunk inkább például a `source` és `target` neveket az `a1` és `a2` helyett.

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.

Annak biztosításához, hogy a függvények egy dolgot csináljanak, az utasítások azonos absztrakciós szintűek kell, hogy legyenek bennük.

Zavaró egy függvényben a különböző absztrakciós szintek keverése, mert az olvasó nem tudja eldönteni egy kifejezésről, hogy az lényeges-e vagy csupán egy technikai részlet.

Stepdown szabály: felülről lefelé haladva csökkenjen a függvények absztrakciós szintje.

A kód felülről lefelé haladva olvasható.

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

Az olvasó számára megnehezítik a kód megértését.

Más absztrakciós szinten vannak, mint a függvény neve, és egy olyan részlet ismeretére kényszerítenek, mely az adott ponton nem különösebben érdekes.

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.

Például `Point p = new Point(0,0);` tökéletesen elfogadható, az argumentumok egyetlen érték komponensei, az argumentumok sorrendje pedig a komponensek sorrendjének felel meg.

Viszont például az `assertEquals(expected, actual)` *JUnit* függvénynél nagyon könnyű összekeverni az argumentumokat.

Példa elfogadható három argumentumú függvényre (*JUnit*):

```
assertEquals(double expected, double actual, double epsilon)
```

Kettőnél vagy háromnál több argumentum esetén bizonyosakat érdemes lehet becsomagolni egy osztályba.

Példa:

- Circle `makeCircle(double x, double y, double radius);`
- Circle `makeCircle(Point center, double radius);`
- Változó argumentumszámú függvényekre lista argumentumú függvényekként tekinthetünk.
- Lásd például: `java.lang.String.format(String format, Object... args)`

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.
- Vagy változtassa meg egy objektum állapotát, vagy adjon vissza információt az objektumról.

Rossz gyakorlat:

Az alábbi függvény, mely beállítja egy adott attribútum értékét, sikeres beállítás esetén `true`-t ad vissza, nem létező attribútum esetén pedig `false`-t:

- ```
public boolean set(String attribute, String value);
```

Az olvasó számára nem világos, hogy mi a kódban például az alábbi utasítás jelentése:

- ```
if (set("username", "unclebob")) { ... }
```

Hibakódok visszaadása helyett részesítsük előnyben a kivételeket.

Így egyszerű további kivételek bevezetése, az új kivételek egy kivételosztály leszármazottai 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. Egy hibakezelő függvény kizárólag egy try/catch blokkot kell, hogy tartalmazzon.

Példa try/catch blokkok kiemelésére:

```
public void delete(Page page) {  
    try {  
        deletePageAndAllReferences(page);  
    } catch (Exception e) {  
        logError(e);  
    }  
}  
  
private void deletePageAndAllReferences(Page page) throws Exception {  
    deletePage(page);  
    registry.deleteReference(page.name);  
    configKeys.deleteKey(page.name.makeKey());  
}  
  
private void logError(Exception e) {  
    logger.log(e.getMessage());  
}
```

Strukturált programozás:

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

Mi a baj a megjegyzésekkel? Jó és rossz megjegyzések fajtái

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.
- 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.

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:** hanyagul odavetett megjegyzés, mely legfeljebb a szerzőnek jelent valamit
- **Fölösleges megjegyzés**
- **Félrevezető megjegyzés:** nem eléggé pontos megjegyzés.
- **Kötelező megjegyzés**
- **Napló megjegyzés**
- **Zaj-megjegyzés:** új információval nem szolgáló magától értetődő megjegyzés
- **Pozíciójelző/szalagcím megjegyzés**
- **Záró kapcsos zárójel megjegyzés**
- **Szerző neve megjegyzésben**
- **Megjegyzésbe tett kód**
- **HTML megjegyzés:** rontja a forráskód olvashatóságát.
- **Nem lokális megjegyzés:** olyan megjegyzés, mely nem a közvetlen környezetében lévő kódra vonatkozik.
- **Túl sok információt tartalmazó megjegyzés**
- **A kódhoz nem nyilvánvalóan kapcsolódó megjegyzés**
- **Javadoc megjegyzés nem nyilvános kódban**

Redundáns megjegyzés: nem szolgál semmiféle plusz információval a kódról, nem könnyebb elolvasni sem, mint magát a kódot.

Kötelező megjegyzés: a nyilvánvalót magyarázó megjegyzés, ami csak azért van ott, mert megkövetelik, hogy minden függvényhez, változóhoz tartozzon dokumentációs megjegyzés.

Napló megjegyzés: egy modul elején a benne végzett minden egyes módosítást naplószerűen dokumentáló megjegyzés.

Szerző neve megjegyzésben: a verziókezelő rendszerek feleslegessé tették.

Megjegyzésbe tett kód: a rossz megjegyzések legutálatosabbja, kerülendő. Mások azt fogják gondolni, hogy okkal van ott, fontos, és ezért nem lesz bátorságuk törölni.

Forráskód formázás: vízszintes és függőleges formázás, az újság metafora

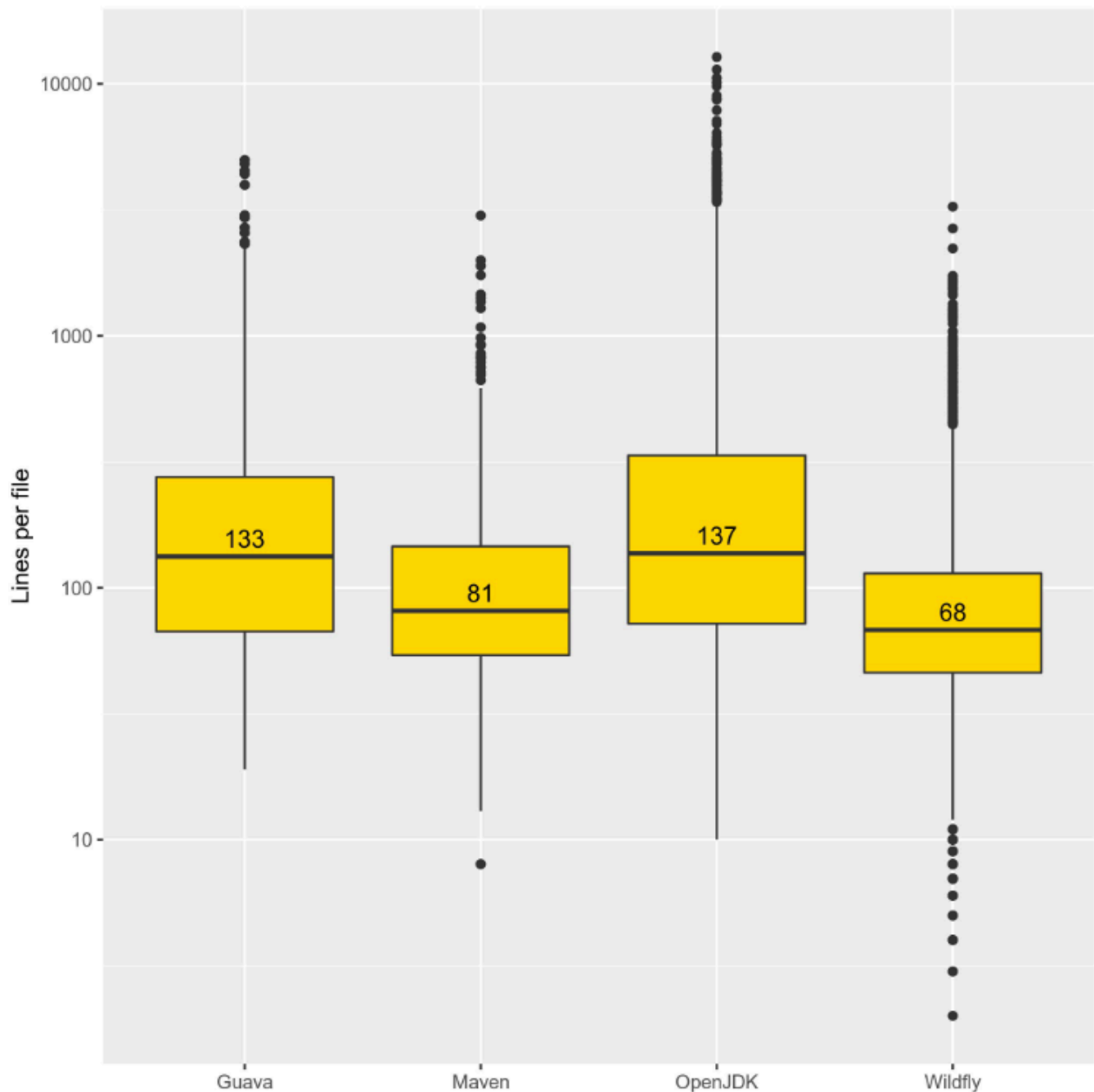
A forráskódot úgy kell formázni, hogy az jól olvasható legyen.

- 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

Empirikus vizsgálatként tekintünk a forrásállományok méretét az alábbi szoftverekben:

- Apache Maven 3.8.4
- Guava 31.0.1
- OpenJDK 17.0.1
- Wildfly 26.0.0.Final



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.

Újság metafora:

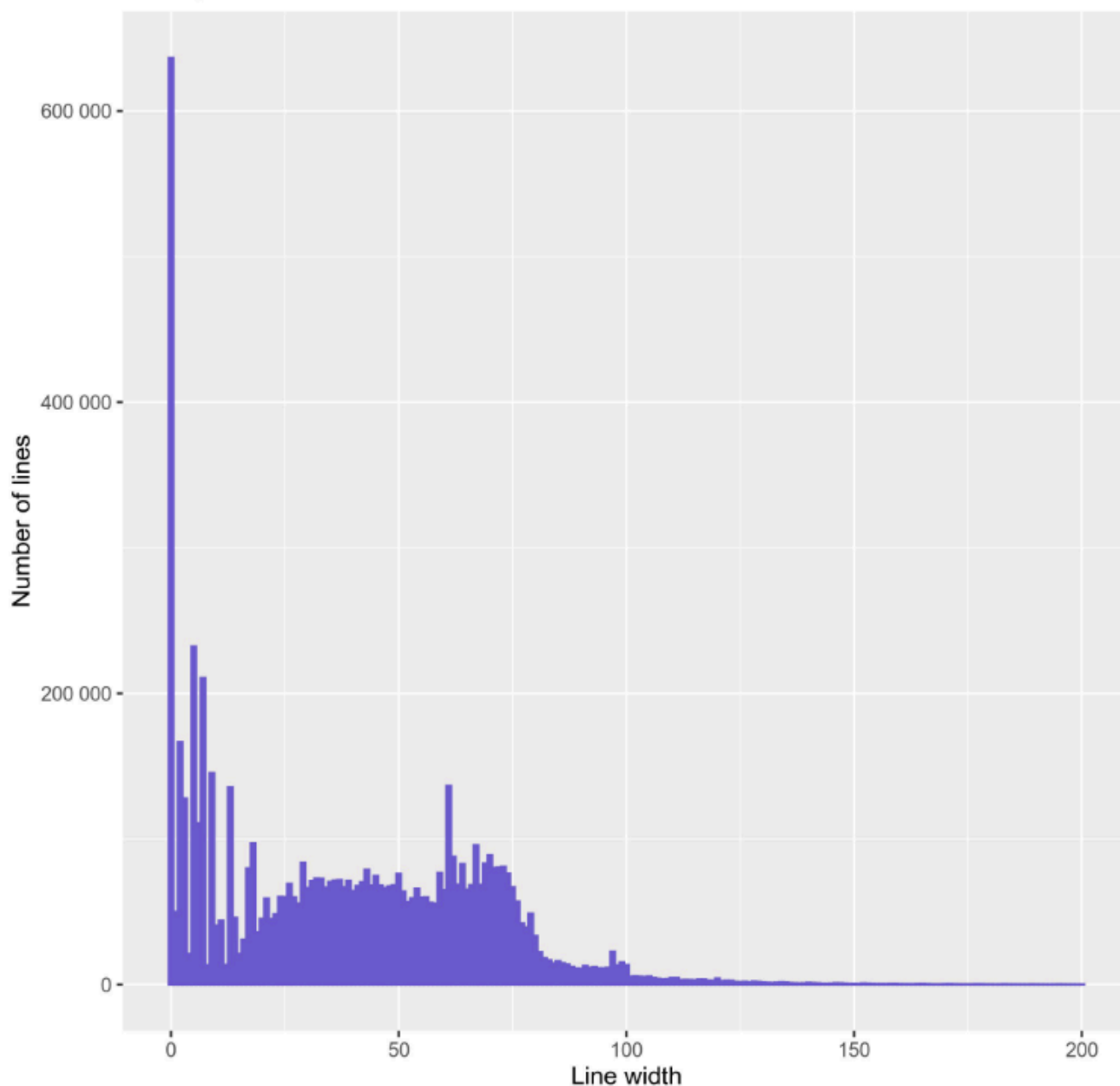
- Egy jól megírt újságcikket felülről lefelé haladva olvasunk.
- A tetején egy olyan cím van, mely alapján az olvasó eldöntheti, hogy érdekli-e egyáltalán. Az első bekezdés a teljes történet egy összefoglalását adja. Lefelé haladva a bekezdések egyre több és több részletet tartalmaznak.
- Egy forrásállomány 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.

Vízszintes formázás

Empirikus vizsgálatként tekintsük a kódsorok hosszának eloszlását:

- Apache Maven 3.8.4
- Guava 31.0.1
- OpenJDK 17.0.1
- Wildfly 26.0.0.Final
- A vizsgálat eredményét a következő oldalon látható hisztogram mutatja

Histogram of line widths



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.

Kerülendő például:

- `public int size() { return size; }`

Ha `for` vagy `while` ciklus törzseként üres utasítást kell használni, ezt célszerű egy új sorba elhelyezni.

Hibakezelés, ellenőrzött és nem ellenőrzött kivételek

Hibakódok visszaadása helyett részesítsük előnyben a kivételeket.

Használjunk nem ellenőrzött kivételeket.

Ne adjunk át/vissza *null*-t.

Java-ban a kivételek ellenőrzöttek (*checked*) vagy nem ellenőrzöttek (*unchecked*).

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

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

Az ellenőrzött kivételek használata megsértheti a nyitva zárt eleveket.

Ha egy alacsony szintű metódust úgy módosítunk, hogy benne egy ellenőrzött kivételen kerüljön dobásra, akkor a metódus szignatúráját is megfelelően kell módosítani.

A *throws* záradéknak tartalmaznia kell a kivételosztályt.

Ez azt jelenti, hogy a módosított metódust meghívó minden metódust úgy kell módosítani, hogy kapja el a kivételt vagy hogy a *throws* záradéka tartalmazza a kivételt.

***null* átadása függvényeknek, null visszaszolgáltatása**

A *null* referenciák azért rosszak, mert `NullPointerException` kivételeket okoznak

Amikor *null* referenciát adunk vissza, akkor lényegében magunknak csinálunk munkát, mivel a *null*-t speciális esetként kell kezelni a `NullPointerException` elkerüléséhez.

Ha kísértést érzünk egy metódusból *null* visszaadására, vegyük helyette fontolóra inkább egy kivétel dobását vagy egy speciális eset objektum (például egy üres lista) visszaadását.

null visszaadása metódusokból rossz, de *null* átadása metódusoknak még rosszabb.

Amikor csak lehetséges, el kell kerülni.

A legtöbb programozási nyelven nincs jó megoldás a hívó által véletlenül átadott *null* referencia kezelésére.

Mivel ez a helyzet, az észszerű megközelítés alapértelmezésben megtiltani *null* átadását

Null-értékűséget jelző annotációk (például `@NotNull`, `@NonNull`, `@Nullable`)

Felhasználhatják őket:

- Statikus kódelemző eszközök (például *Checker Framework*, *IntelliJ IDEA*)
 - Kódgenerátor eszközök (például *IntelliJ IDEA*, *Lombok*)
-