

Szoftverfejlesztés

II. Zárthelyi Dolgozat

Tartalom

- Unified Modeling Language
 - UML modellelemek
 - Osztálydiagramok
 - Osztályok
 - Számosság
 - Tulajdonságok
 - Műveletek
 - Statikus attribútumok és műveletek
 - Absztrakt osztályok
 - Asszociációk
 - Egész-rész kapcsolat
 - Általánosítás
 - Interfészek
 - Szoftvertesztelés
 - Objektumorientált tervezési alapelvek
 - Minták a szoftverfejlesztésben
 - Tiszta kód
-

1. Unified Modeling Language (UML)

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

osztályozók

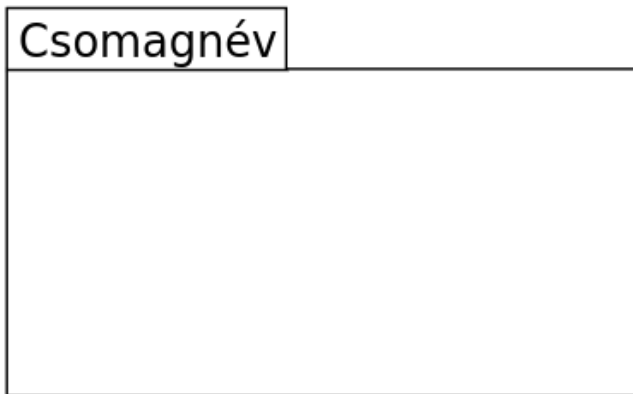
Az osztályozó egy modellelem, mely közös jellemzőkkel műveletekkel rendelkező példányok egy halmazát ábrázolja.

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

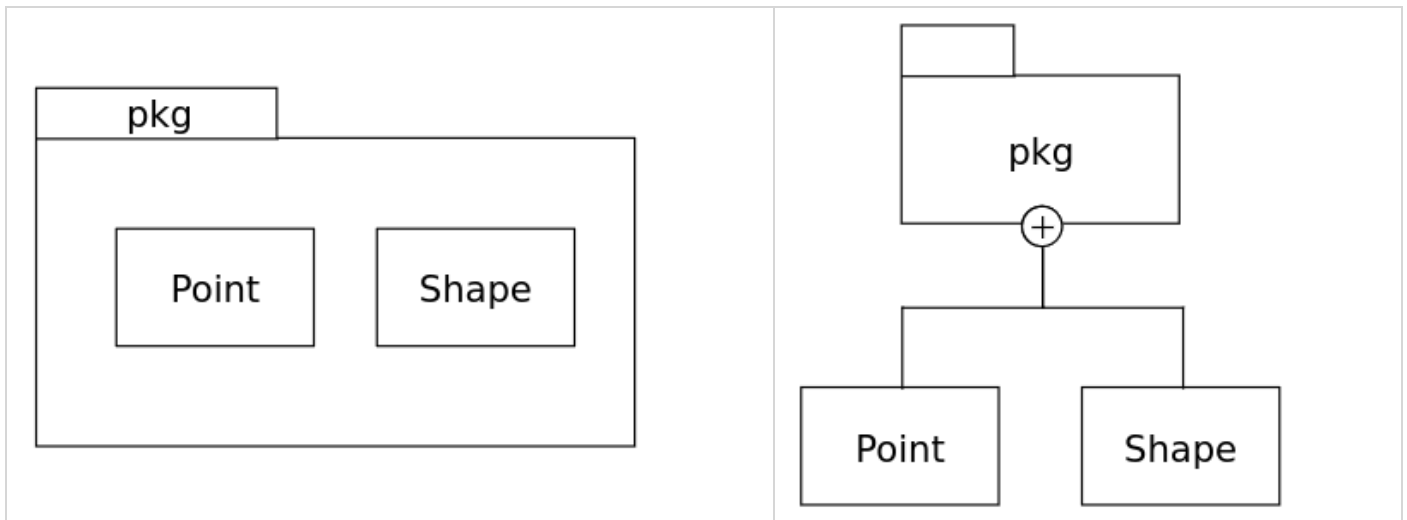
csomagok

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

Jelölésmód:



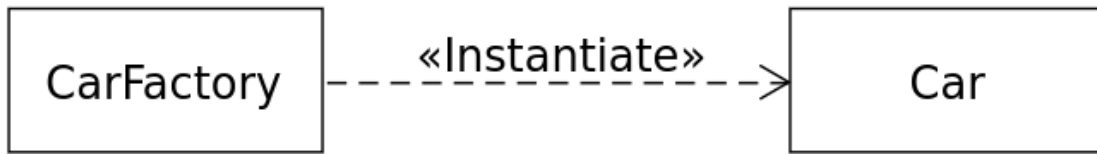
A tartalmazott elemekre csomagnév::elemnév formájú minősített nevekkel lehet hivatkozni (például `pkg::Point`, `pkg::Shape`).



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 sztereotípa.

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.

This class was added
right after watching
the movie 'Monty Python
and the Holy Grail'.

Shrubbery

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 [, 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.

Példa:

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

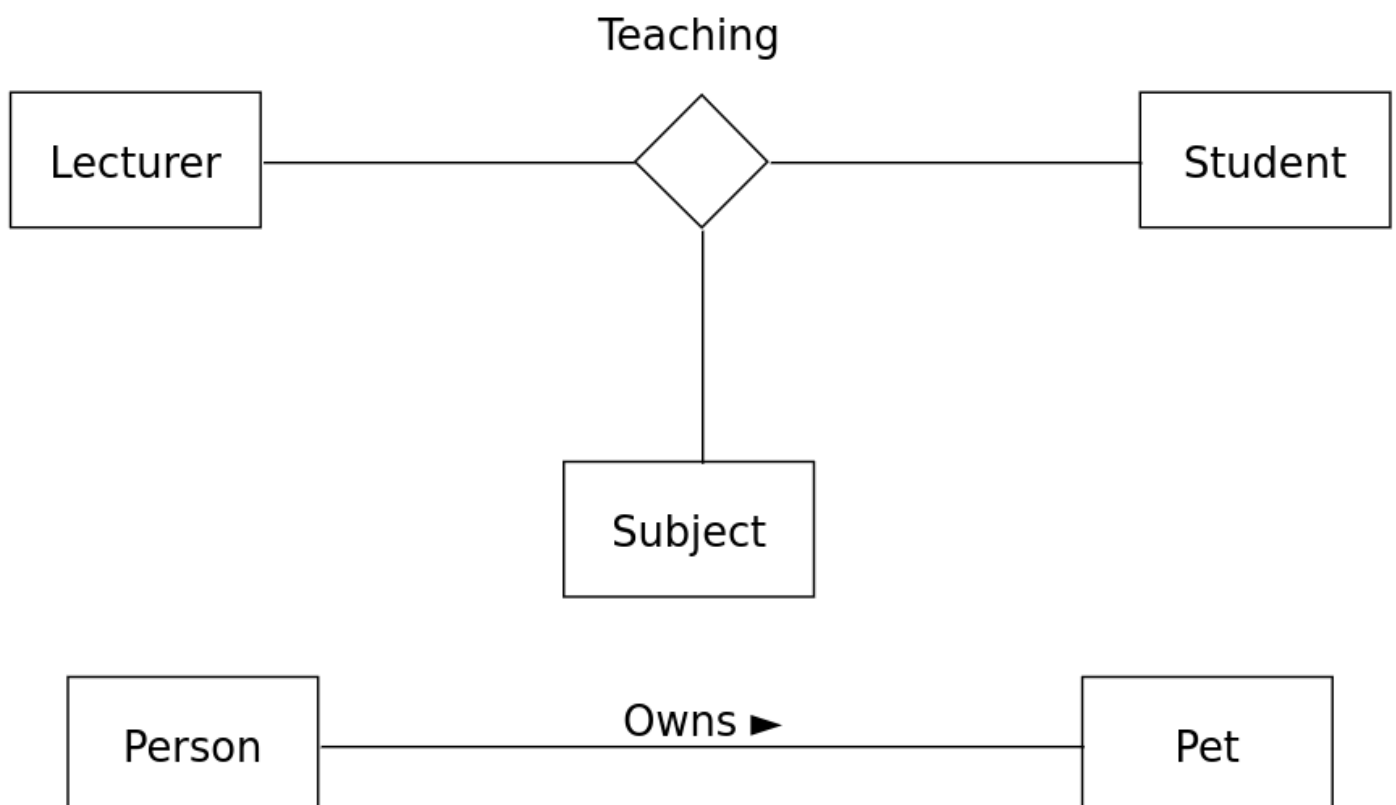
Asszociációk

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 rombuszsal, 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):

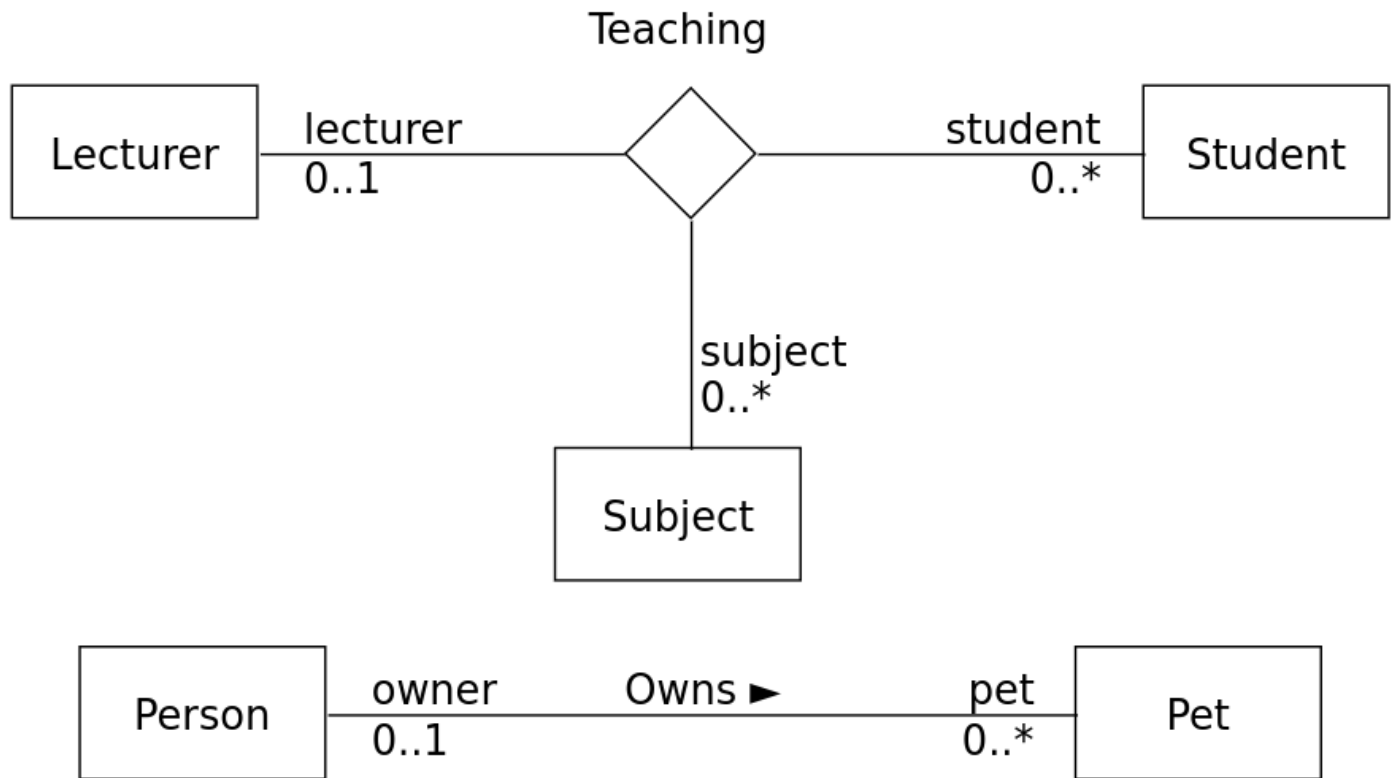
- Név (gyakran szerepkörnek nevezik)
- Számosság (ha nincs megadva, akkor semmilyen feltevessel nem élhetünk a számosságról)

- Módosító (lásd a tulajdonságoknál)
- Láthatóság

A vonal végén egy **nyílt nyílhegy** azt jelzi, hogy a vég navigálható, egy **×** 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.





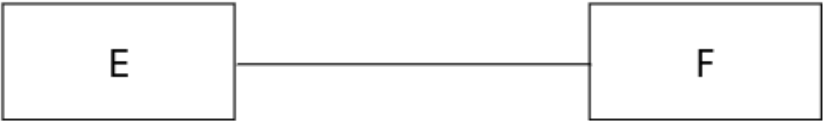
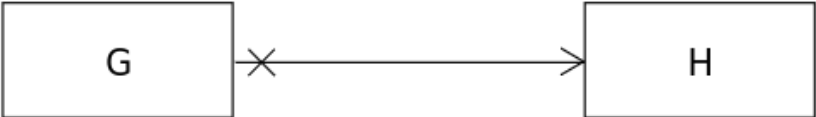

Az osztályozó és a vonal érintkezési pontjában elhelyezhető egy kis tömör kör (a továbbiakban pontnak nevezzük).

- 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.	

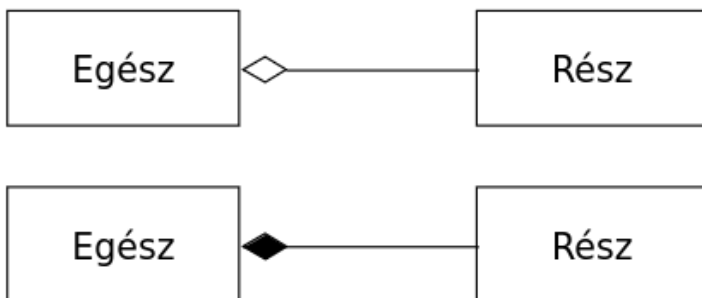
Egész-rész kapcsolat

A bináris asszociációk egész-rész kapcsolatot kifejező fajtái:

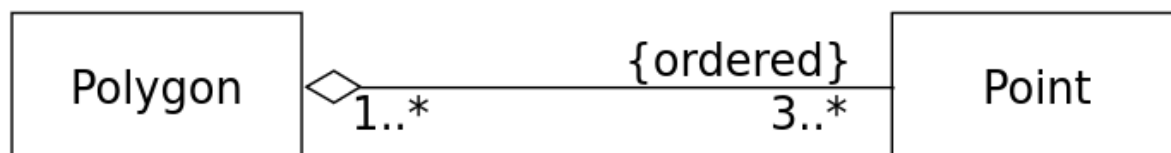
- **Aggregáció** (*shared aggregation, aggregation*): Egy rész objektum egyidejűleg több aggregációs objektumhoz is tartozhat, a részek és az aggregációs objektum egymástól függetlenül is létezhetnek.
- **Kompozíció** (*composite aggregation, composition*): Az aggregáció erősebb formája. Egy rész objektum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

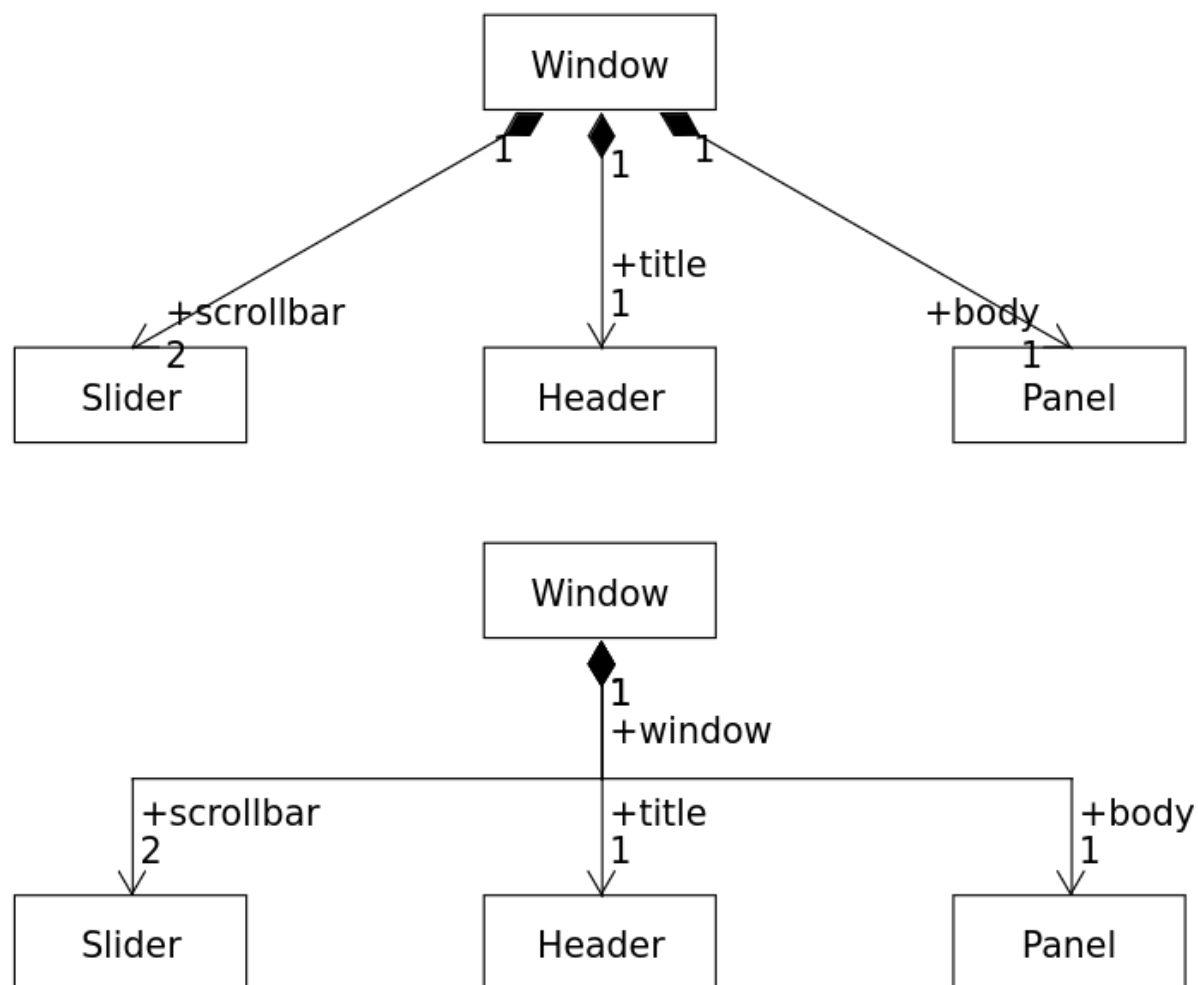
Jelölésmód:



Példa aggregációra:



Példa kompozícióra:



Általánosítás

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

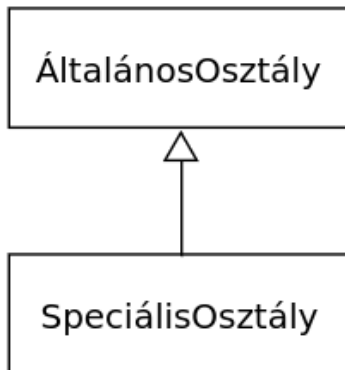
Az általánosítás/specializáció reláció tranzitív lezártja szerint értelmezzük egy osztályozó általánosításait és specializációit.

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

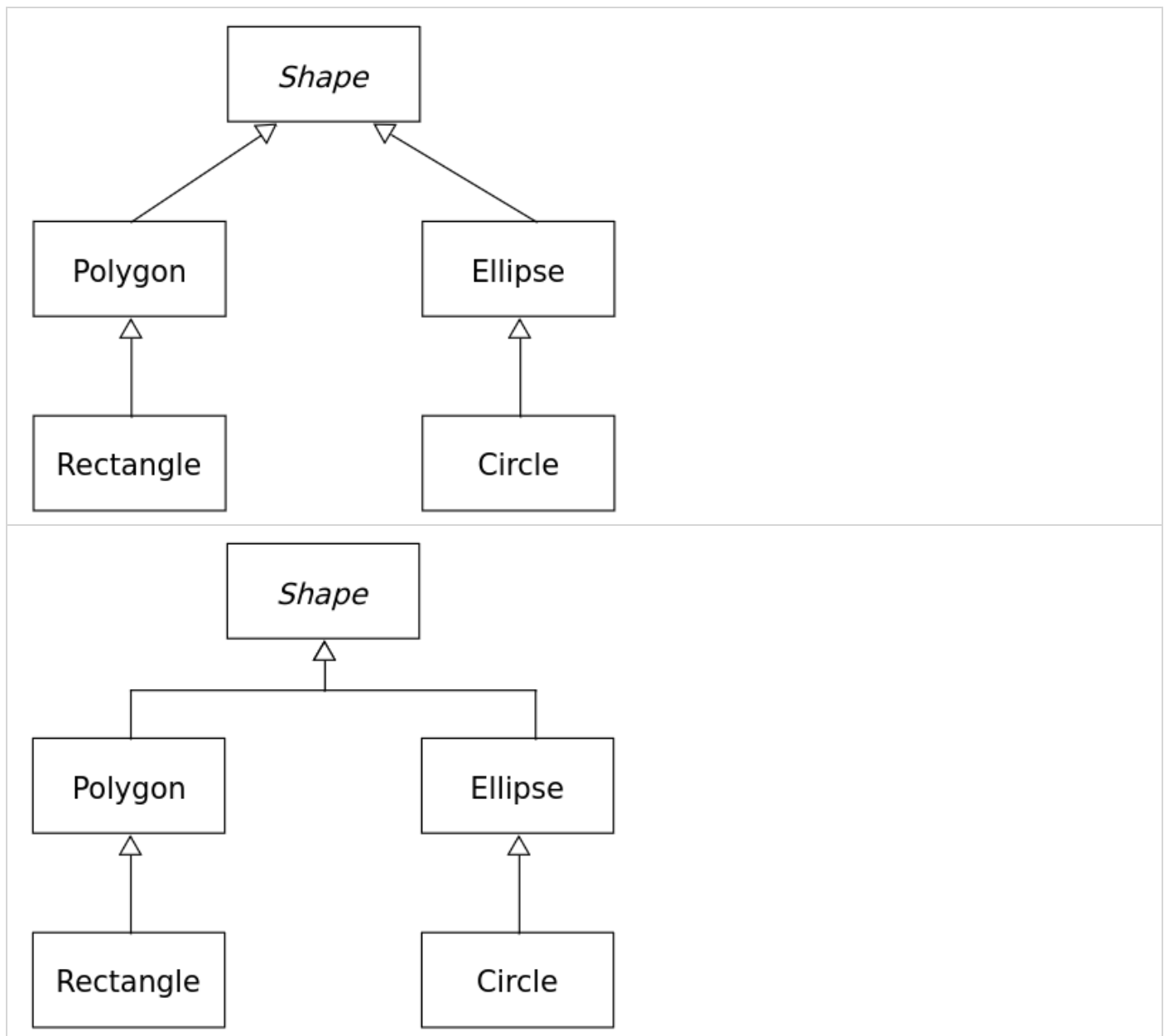
Egy osztályozó egy példánya minden általánosításának példánya.

A speciális osztályozó örökli az általános osztályozó bizonyos tagjait.

Jelölésmód:



Példa:

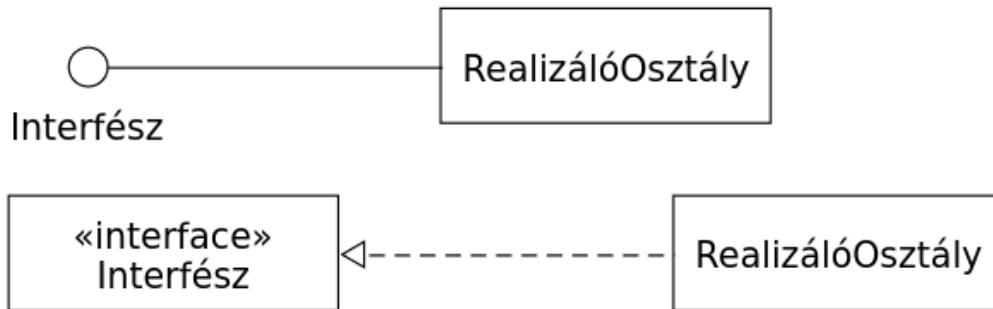


Interfészek

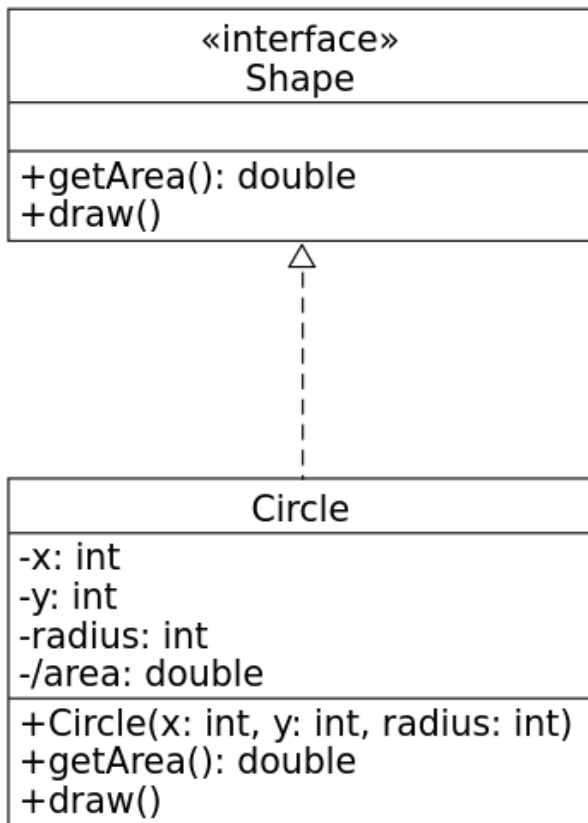
Az interfész egy olyan fajta osztályozó, mely nyilvános jellemzőket és kötelezettségeket deklarál. Az interfész egy szerződést határoz meg.

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 vontakozás, 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 felelősége, 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.

Teszt típusok

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 használhatóság, teljesítmény vagy biztonság.

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

fehér dobozos tesztelés

A rendszer belső felépítésén vagy megvalósításán alapuló tesztek.

A belső szerkezetbe beleérthető kód, architektúra vagy a rendszeren belüli munkafolyamatok.

változással kapcsolatos tesztelés

Teszteket kell végezni, amikor módosítások történnek egy rendszerben egy hiba kijavításához vagy új funkcionalitás hozzáadásához/létező funkcionalitás módosításához.

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 tesztekkel 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, öröközhetők 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:

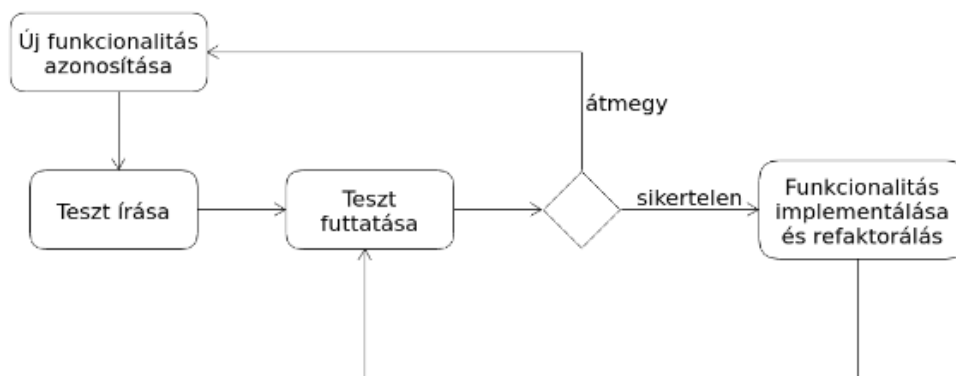
1. Írj egy tesztet (Write a test)
2. Érd el, hogy működjön (Make it run)
3. Javítsd ki (Make it right)

A TDD mantrája:

1. Vörös: írd 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: távolíts 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. Objektumorientált tervezési alapelvek

Statikus kódelemzés fogalma, példák statikus kódelemző eszközökre

A DRY elv

A KISS elv

A YAGNI elv

Csatoltság, laza és szoros csatoltság

GoF alapelvek

SOLID alapelvek: egyszeres felelősség elve, nyitva zárt elv, Liskov-féle helyettesítési

elv, interfész szétválasztási elv, függőség megfordítási elv

Függőség befecskendezés

4. Minták a szoftverfejlesztésben

Mi a minta?

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

Tervezési minták, tervezési minták osztályozása

Létrehozási minták: elvont gyár (abstract factory), egyke, építő, objektumkészlet (object pool)

Szerkezeti minták: díszítő, illesztő

Viselkedési minták: sablonfüggvény, megfigyelő

Programozási idiómák/implementációs minták

Antiminták, a massa és a spagetti kód antiminta

5. Tiszta kód

Milyen a tiszta kód?

értelmes nevek

Függvények

Mi a baj a megjegyzésekkel? Jó és rossz megjegyzések fajtái

Forráskód formázás: vízszintes és függőleges formázás, az újság metafora

Hibakezelés, ellenőrzött és nem ellenőrzött kivételek

null átadása függvényeknek, null visszatérése
