Programozási technológia 2.

Verifikáció és validáció
Egységtesztelés (JUnit)

Dr. Szendrei Rudolf
ELTE Informatikai Kar
2020.

Verifikáció és validáció

Minőségbiztosítás

- A szoftver verifikációja és validációja, vagy *minőségbiztosítása* (*quality control*) azon folyamatok összessége, amelyek során ellenőrizzük, hogy a szoftver teljesíti-e az elvárt követelményeket, és megfelel a felhasználói elvárásoknak
  - a *verifikáció* (*verification*) ellenőrzi, hogy a szoftvert a megadott funkcionális és nem funkcionális követelményeknek megfelelően valósították meg
    - történhet formális, vagy szintaktikus módszerekkel
  - a *validáció* (*validation*) ellenőrzi, hogy a szoftver megfelel-e a felhasználók elvárásainak, azaz jól specifikáltuk-e eredetileg a követelményeket
    - alapvető módszere a tesztelés

Verifikáció és validáció

Módszerei

- Az ellenőrzés végezhető
  - *statikusan*, a modellek és a programkód áttekintésével
    - elvégezhető a teljes program elkészülte nélkül is
    - elkerüli, hogy hibák elfedjék egymást
    - tágabb körben is felfedhet hibákat, pl. szabványoknak történő megfelelés
  - *dinamikusan*, a program futtatásával
    - felfedheti a statikus ellenőrzés során észre nem vett hibákat, illetve a programegységek együttműködéséből származó hibákat
    - lehetőséget ad a teljesítmény mérésére

Verifikáció és validáció

Tesztelés

- A tesztelés célja a szoftverhibák felfedezése és szoftverrel szemben támasztott minőségi elvárások ellenőrzése
  - futási idejű hibákat (*failure*), működési rendellenességeket (*malfunction*) keresésünk, kompatibilitást ellenőrzünk
  - általában a program (egy részének) futtatásával, szimulált adatok alapján történik
  - nem garantálja, hogy a program hibamentes, és minden körülmény között helytáll, de felfedheti a hibákat adott körülmények között
- A teszteléshez *tesztelési terv*et (*test plan*) készítünk, amely ismerteti a tesztelés felelőseit, folyamatát, technikáit és céljait

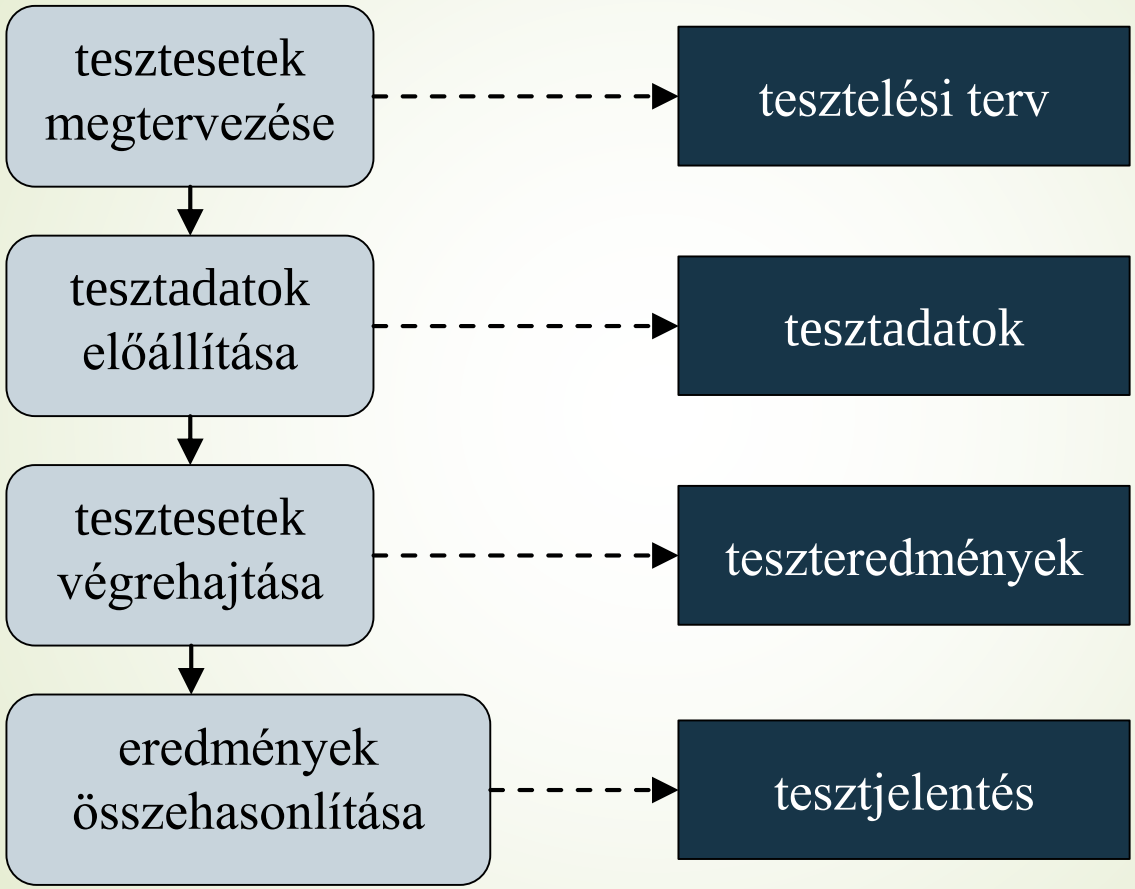
Verifikáció és validáció

Tesztesetek

- A tesztelés során különböző *teszteset*eket (*test case*) különböztetünk meg, amelyek az egyes funkciókat, illetve elvárásokat tudják ellenőrizni
  - megadjuk, adott bemenő adatokra mi a várt eredmény (*expected result*), amelyet a teszt lefutása után összehasonlítunk a kapott eredménnyel (*actual result*)
  - a teszteseteket összekapcsolhatjuk a követelményekkel, azaz megadhatjuk melyik teszteset milyen követelményt ellenőriz (*traceability matrix*)
  - a teszteseteket gyűjteményekbe helyezzük (*test suit*)
- A tesztesetek eredményeiből készül a *tesztjelentés* (*test report*)

Verifikáció és validáció

A tesztelési folyamat



Verifikáció és validáció

A tesztelés lépései

- A tesztelés nem a teljes program elkészülte után, egyben történik, hanem általában 3 szakaszból áll:

1. *fejlesztői teszt* (*development testing*): a szoftver fejlesztői ellenőrzik a program működését
   - jellemzően *fehér doboz* (*white box*) tesztek, azaz a fejlesztő ismeri, és követi a programkódot
2. *kiadásteszt* (*release testing*): egy külön tesztcsapat ellenőrzi a szoftver használatát
3. *felhasználói teszt* (*acceptance testing*): a felhasználók tesztelik a programot a felhasználás környezetében
   - jellemzően *fekete doboz* (*black box*) tesztek, azaz a forráskód nem ismert

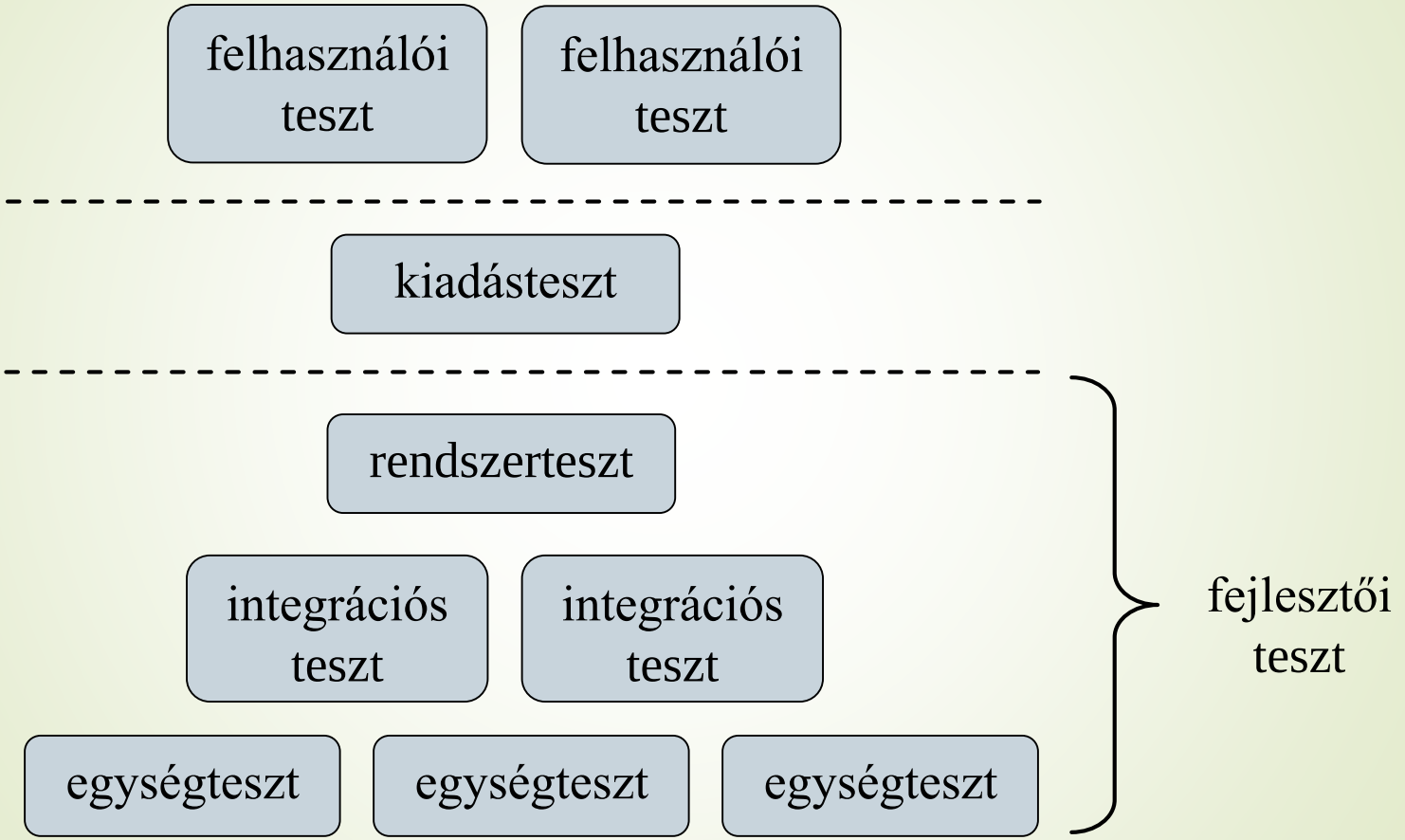
Verifikáció és validáció

A tesztelés lépései

- A fejlesztői tesztnek további négy szakasza van:
  - *egységteszt* (*unit test*): a programegységeket (osztályok, metódusok) külön-külön, egymástól függetlenül teszteljük
  - *integrációs teszt* (*integration test*): a programegységek együttműködésének tesztje, a rendszer egy komponensének vizsgálata
  - *rendszerteszt* (*system test*): az egész rendszer együttes tesztje, a rendszert alkotó komponensek közötti kommunikáció vizsgálata
- A tesztelés egy része automatizálható, bizonyos részét azonban mindenképpen manuálisan kell végrehajtanunk

Verifikáció és validáció

A tesztelés lépései



Verifikáció és validáció

Nyomkövetés

- A tesztelést elősegíti a *nyomkövetés* (*debugging*), amely során a programot futás közben elemezzük, követjük a változók állapotait, a hívás helyét, felfedjük a lehetséges hibaforrásokat
- A jellemző nyomkövetési lehetőségek:
  - *megállási pontok* (*breakpoint*) elhelyezése
  - *változókövetés* (*watch*), amely automatikus a lokális változókra, szabható rá feltétel
  - *hívási lánc* (*call stack*) kezelése, a felsőbb szintek változóinak nyilvántartásával
- A fejlesztőkörnyezetbe épített eszközök mellett külső programokat is használhatunk (pl. *gdb*)

Verifikáció és validáció

Egységtesztek

- Az egységteszt során törekednünk kell arra, hogy a programegység összes funkcióját ellenőrizzük, egy osztály esetén
  - ellenőrizzük valamennyi (publikus) metódust
  - állítsuk be, és ellenőrizzük az összes mezőt
  - az összes lehetséges állapotba helyezzük az osztályt, vagyis szimuláljuk az összes eseményt, amely az osztályt érheti
- A teszteseteket célszerű leszorítani a programegység által
  - megengedett bemenetre, így ellenőrizve a várt viselkedését (korrektség)
  - nem megengedett bemenetre, így ellenőrizve a hibakezelést (robosztusság)

Verifikáció és validáció

Egységtesztek

- A bemenő adatokat részhalmazokra bonthatjuk a különböző hibalehetőségek függvényében, és minden részhalmazból egy bemenetet ellenőrizhetünk
- Pl. egy téglalap méretei egész számok, amelyek lehetnek
  - negatívak, amely nem megengedett tartomány
  - nulla, amely megengedhető (üres téglalap)
  - pozitívak, amelyek megengedettek, ugyanakkor speciális esetet jelenthetnek a nagy számok
- Az egységtesztet az ismétlések és a számos kombináció miatt célszerű automatizálni (pl. a teszt implementációjával)

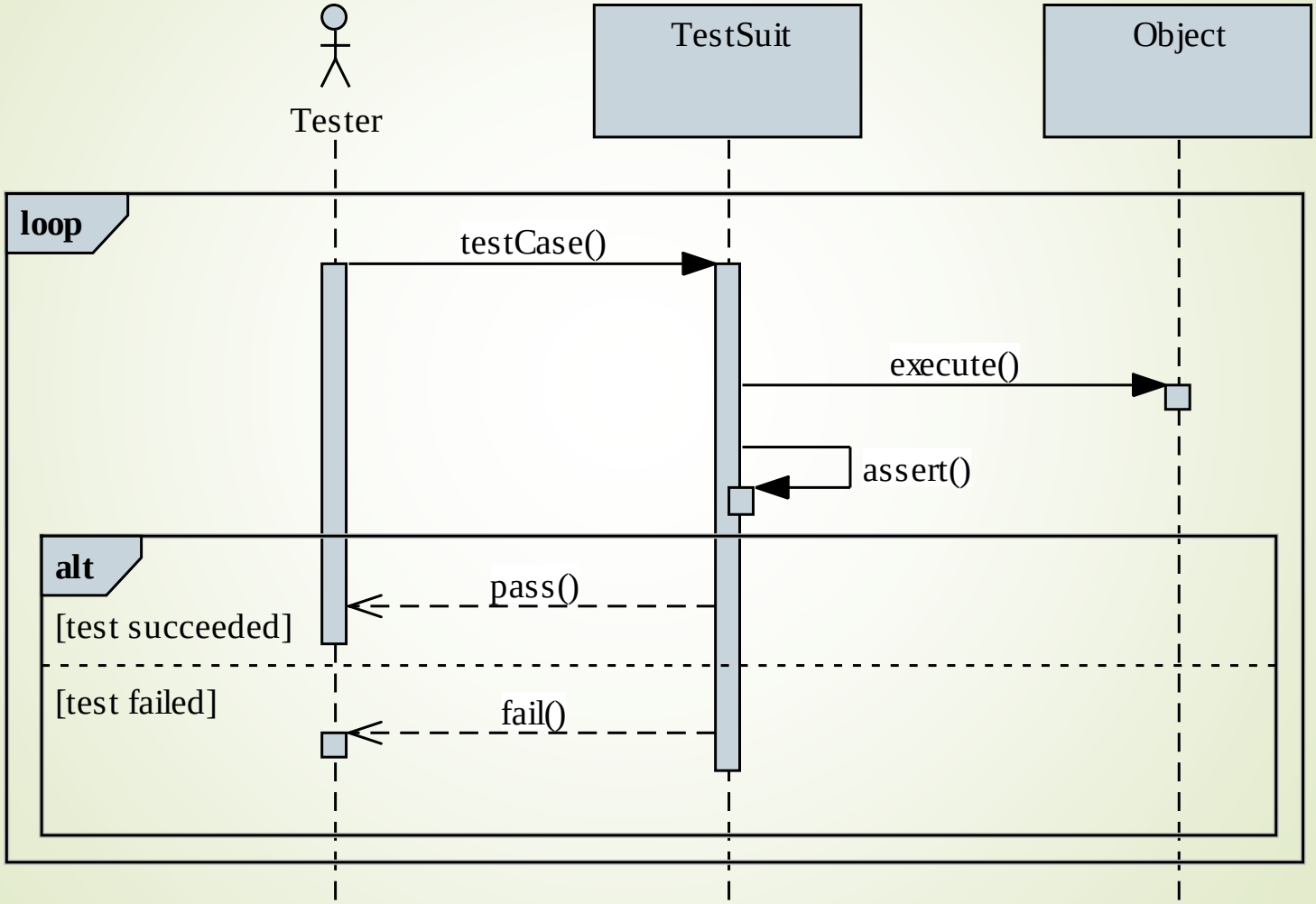
Verifikáció és validáció

Tesztelési keretrendszerek

- Az egységtesztek automatizálását, és az eredmények kiértékelését hatékonyabbá tehetjük tesztelési keretrendszerek (*unit testing frameworks*) használatával
  - általában a tényleges főprogramoktól függetlenül építhetünk teszteseteket, amelyeket futtathatunk, és megkapjuk a futás pontos eredményét
  - a tesztesetekben egy, vagy több ellenőrzés (*assert*) kap helyet, amelyek jelezhetnek hibákat
  - amennyiben egy hibajelzést sem kaptunk egy tesztesettől, akkor az eset sikeres (*pass*), egyébként sikertelen (*fail*)
  - pl. JUnit, *CppTest*, *QTestLib*

Verifikáció és validáció

Tesztelési keretrendszerek



Verifikáció és validáció

Kód lefedettség

- A tesztgyűjtemények által letesztelt programkód mértékét nevezzük *kód lefedettség*nek (*code coverage*)
  - megadja, hogy a tényleges programkód mely részei kerültek végrehajtásra a teszt során
  - számos szempont szerint mérhető, pl.
    - *alprogram* (*function*): mely alprogramok lettek végrehajtva
    - *utasítás* (*statement*): mely utasítások lettek végrehajtva
    - *elágazás* (*branch*): az elágazások mely ágai futottak le
    - *feltételek* (*condition*): a logikai kifejezések mely részei lettek kiértékelve (mindkét esetre)

Verifikáció és validáció

További tesztek

- Az integrációs és rendszertesztek során elsősorban azt vizsgáljuk, hogy a rendszer megfelel-e a követelménybeli elvárásoknak
  - funkcionális és nem funkcionális alapon (pl. teljesítmény, biztonság) is ellenőrizhetjük a rendszert
  - ezeket a teszteseteket már a specifikáció során megadhatjuk
  - a tesztelés első lépése a *füst teszt* (*smoke test*), amely során a legalapvetőbb funkciók működését ellenőrzik
- A kiadásteszt és a felhasználói teszt során a szoftvernek már általában a célkörnyezetben, tényleges adatokkal kell dolgoznia
  - a teszt magába foglalja a kihelyezést (pl. telepítés) is

Verifikáció és validáció

Programváltozatok

- Az implementáció és tesztelés során a szoftver különböző változatait tartjuk nyilván:
  - *pre-alfa*: funkcionálisan nem teljes, de rendszertesztre alkalmas
  - *alfa*: funkcionálisan teljes, de a minőségi mutatókat nem teljesíti
  - *béta*: funkcionálisan teljes, és a minőségi mutatók javarészt megfelelnek a követelményeknek
    - a további tesztelés során nagyrészt a rendellenességek kiküszöbölése folyik, a tesztelés lehet publikus
    - esetlegesen kiegészítő funkciók kerülhetnek implementálásra

Verifikáció és validáció

Programváltozatok

- *kiadásra jelölt* (*release candidate*, *RC*), vagy *gamma*: funkcionálisan teljes, minőségi mutatóknak megfelelő
  - kódteljes (nem kerül hozzá újabb programkód, legfeljebb hibajavítás)
  - csak dinamikus tesztelés folyik, és csak kritikus hiba esetén nem kerül gyártásra
- *végleges* (*final, release to manufacturing, RTM*): a kiadott, legyártott változat
  - nyílt forráskód esetén általában már korábban publikussá válik a (félkész) szoftver
  - a kiadást követően a program további változásokon eshet át (javítások, programfunkció bővítés)

Verifikáció és validáció

Teljesítménytesztek

- A *teljesítménytesztek* (*performance test*) során a rendszer teljesítményét mérjük
  - ezáltal a rendszer megbízhatóságát és teljesítőképességének (válaszidők, átviteli sebességek, erőforrások felhasználása) ellenőrizzük különböző mértékű feladatvégzés esetén
  - végezhetünk teszteket a várható feladatmennyiség függvényében (*load test*), vagy azon túl ellenőrizhetjük a rendszer tűrőképességét (*stress test*)
  - a teljesítmény tesztet sokszor a hardver erőforrások függvényében végezzük, amellyel megállapítható a rendszer skálázhatósága (*capacity test*)

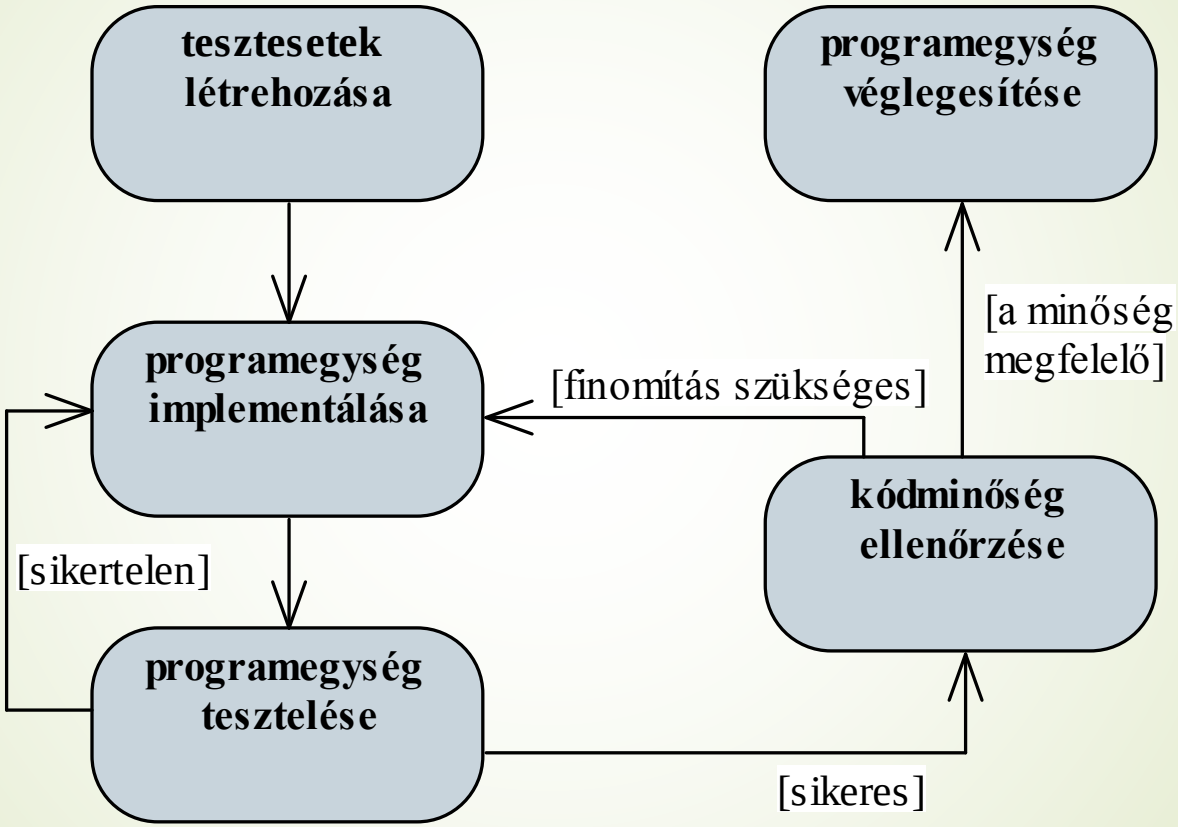
Verifikáció és validáció

Tesztvezérelt fejlesztés

- A *tesztvezérelt fejlesztés* (*test-driven development*, *TDD*) egy olyan fejlesztési módszertan, amely a teszteknek ad elsőbbséget a fejlesztés során
  - a fejlesztés lépései:
    1. tesztesetek elkészítése, amely ellenőrzi az elkészítendő kód működését
    2. az implementáció megvalósítása, amely eleget tesz a teszteset ellenőrzéseinek
    3. az implementáció finomítása a minőségi elvárásoknak (tervezési és fejlesztési elvek) megfelelően
  - előnye, hogy magas fokú a kód lefedettsége, mivel a teszteknek minden funkcióra ki kell térniük

Verifikáció és validáció

Tesztvezérelt fejlesztés



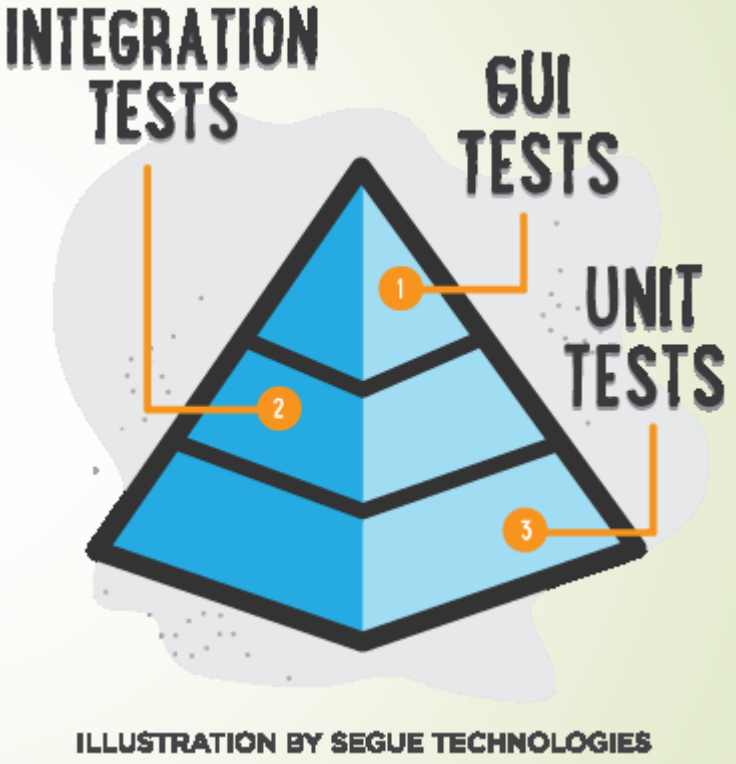
Verifikáció és validáció

A hibajavítás költségei

A hibajavítás költsége a hiba felfedezésének helye (ideje) függvényében		Hiba felfedezésének helye				
		Követelmények	*Tervezés*	*Implementáció*	*Tesztelés*	*Üzemeltetés*
Hiba helye	*Követelmények*	1x	3x	5-10x	10x	10-100x
	Tervezés		1x	10x	15x	25-100x
	Implementáció			1x	10x	10-25x

Egységtesztelés

- A legalacsonyabb szintű, a programot felépítő egységek tesztelése
- Egység: egy rendszer legkisebb önálló egységként tesztelhető része.
- Egység tesztekkel ellenőrizhető, hogy egy egység az elvárásoknak megfelelően működik.
- Egy egység függvényeiről ellenőrizzük, hogy különböző bemenetek esetén megfelelő eredményt, vagy hibát produkálnak.
- Az egységeket egymástól függetlenítve kell tesztelni.



Egységtesztelés

Előnyök

- A hibák sokkal korábban észlelhetőek
- Minden komponens legalább egyszer tesztelt
- Az egységek elkülönítése miatt a hibák helyének meghatározása könnyű
- A funkciók könnyen módosíthatóak, átalakíthatók
- Dokumentációs szerep: példákat biztosít egyes funkciók használatára

Egységtesztelés

Elvek

- **Gyors:** A teszteknek gyorsan kell futnia, lassú teszteket senki nem futtatja gyakran, így a hibák nem derülnek ki idejében.
- **Független:** A teszteknek egymástól függetlennek és bármilyen sorrendben végrehajthatónak kell lennie.
- **Megismételhető:** A teszteknek bármilyen környezetben, hálózat nélkül is végrehajthatónak kell lennie. A különböző futtatások eredményének meg kell egyeznie.
- **Önellenőrző:** A tesztek eredménye egy logikai érték (futásuk vagy sikeres, vagy sikertelen)
- **Automatikus**: Automatikusan, interakció nélkül futó tesztek

Egységtesztelés

Mit kell tesztelni?

- Egy osztály minden publikus metódusát tesztelni kell
- „Triviális” eseteket
- Speciális eseteket
  - Pl.: számok esetén: negatív, 0, pozitív,
    a megengedettnél kisebb, nagyobb,…
- Pozitív / Negatív teszteseteket
  - A negatív teszteset szándékosan hibás paraméterekkel hívja a tesztelt metódust, célja a hibakezelés ellenőrzése
- Végrehajtási lefedettség: a tesztekből indított hívásoknak a tesztelt metódus lehető legtöbb során végig kell haladnia.
- Egy teszt egyetlen „dolog”

Egységtesztelés

Mit NEM kell tesztelni?

- Ami nyilvánvalóan működik
  - külső lib-ek, JDK, JRE, …
- Adatbázis
  - feltételezhető, hogy ha elérhető, akkor helyesen működik
- Triviális metódusok
  - getter / setter
- GUI
  - a felhasználó felület nem tartalmazhat üzleti logikát

Egységtesztelés

JUnit 4.x

- A JUnit egy egységtesztelő keretrendszer a Java nyelvhez
- Annotációkkal konfigurálható.
- Elvárt eredmények ellenőrzése Assert-ekkel.

```
public class MyTests {
  @Test
  public void multiplicationOfZeroIntegersShouldReturnZero() {
    // MyClass is tested
    MyClass tester = new MyClass();
    // assert statements
    assertEquals("10 x 0 must be 0", 0, tester.multiply(10,  0));
    assertEquals("0 x 10 must be 0", 0, tester.multiply( 0, 10));
    assertEquals("0 x 0 must be 0",  0, tester.multiply( 0,  0));
  }
}
```

Egységtesztelés

Teszt osztály felépítése

Annotáció	Eredmény
`@Test` `public void method()`	Teszt metódus jelölése
`@Test(expected = Exception.class)`	A teszt sikertelen a megadott típusú kivétel hiányában
`@Test(timeout = 100)`	A teszt sikertelen, ha a metódus nem fejeződik be adott idő alatt
`@Before / @After` `public void method()`	Metódus amely minden teszt előtt/után lefut
`@BeforeClass / @AfterClass` `public static void method()`	Egyszer fut le az osztályban lévő első teszt metódus indulása előtt/után
`@Ignore / @Ignore("Why disabled")`	A teszt metódus kihagyása

Egységtesztelés

Ellenőrzések, Assert-ek

Utasítás	Leírás
fail(message)	A hívó teszt sikertelen futását eredményezi
assertTrue([message,] boolean condition**)** vagy **assertFalse**(…)	Ellenőrzi, hogy a megadott feltétel igaz (hamis)–e.
assertEquals([message,] expected, actual**)**	Ellenőrzi, hogy két objektum egyenlő-e.
assertNull([message,] obj) vagy **assertNotNull(**…**)**	Az objektum null (nem null).
assertSame([message,] expected, actual) vagy **assertNotSame**(…)	Ellenőrzi, hogy két változó ugyanarra az objektumra mutat (== / !=)

Egységtesztelés

Kivételkezelés ellenőrzése

- A `@Test(expected = Exception.class)` annotáció limitált, csak egyetlen kivételt tud tesztelni.

- Alternatíva

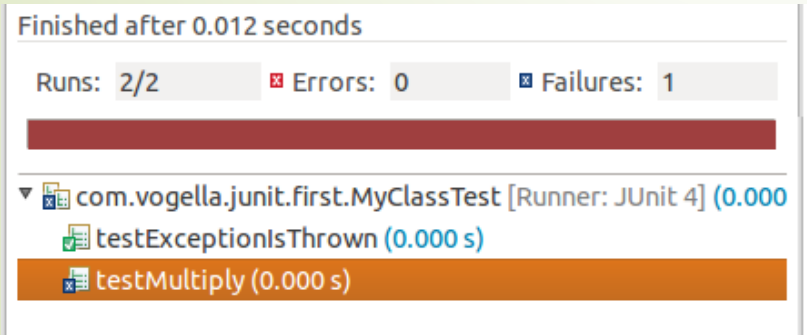
```
try {
    mustThrowException();
    fail();
} catch (Exception e) {
    // expected
}
```

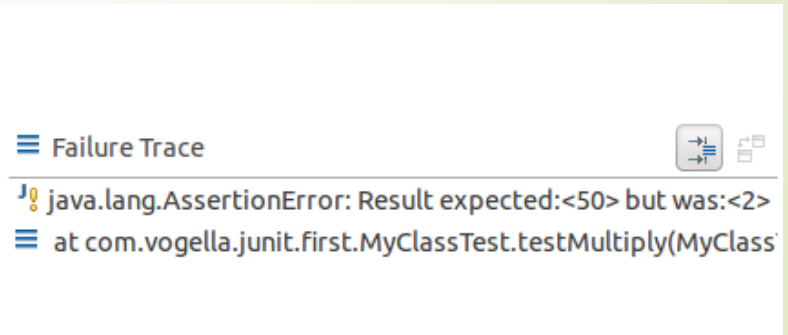
Egységtesztelés

Példa

```
public class MyClass{
  public int multiply(int x, int y) {
    if (x > 999) {
      throw new IllegalArgumentException();
    }
    return x / y;
  }
}
```

```
public class MyClassTest {
  @Test(expected =
     IllegalArgumentException.class)
  public void testExceptionIsThrown(){
    MyClass m = new MyClass();
    m.multiply(1000, 5);
  }
  @Test
  public void testMultiply() {
    MyClass m = new MyClass();
    assertEquals(50,m.multiply(10,5));
  }
}
```





Egységtesztelés

Mock

- A tesztelt metódusok általában más komponens nyújtotta szolgáltatásokat is használnak
- Az egyes funkciókat más komponensektől izoláltan kell tesztelnünk, a tesztek sikeressége nem függhet a függőségek helyes implementációjától
- A Mock egy olyan objektum, amely egy másik objektum működését szimulálja
- Egy osztály függőségeinek szimulálására használandó
- http://easymock.org/

Egységtesztelés

Mock példa

```
public interface ValidatorService {
    boolean allowedToRent(Member m, int numberOfBooksToRent);
}

public class RentalController {
    private ValidatorService validatorService;

    public RentalController(ValidatorService validatorService) {
        this.validatorService = validatorService;
    }

    public void rent(Member m, List<Book> books) {
        if(!validatorService.allowedToRent(m, books.size()) {
            throw new ValidationException();
        }
        ...
    }
}
```

Egységtesztelés

Mock példa – folytatás

```
public class RentalControllerTest {

  private ValidatorService validatorService;
  private RentalController controller;

  @Before public void setUp() throws Exception {

    // NiceMocks return default values for unimplemented methods
      validatorService = createNiceMock(ValidatorService.class);
      controller = new RentalController(validatorService);
  }


  @Test(expected = ValidationException.class)
  public void testRentMoreThanAllowed();

      Member m = new Member();
      List<Books> books = new ArrayList<>();
      expect(validatorService.allowedToRent(m, books))
                                        .andReturn(false).times(1);

      replay(validatorService);
      controller.rent(m, books);
  }
}
```