

**Eötvös Loránd Tudományegyetem
Informatikai Kar**

Szoftverttechnológia

10. előadás

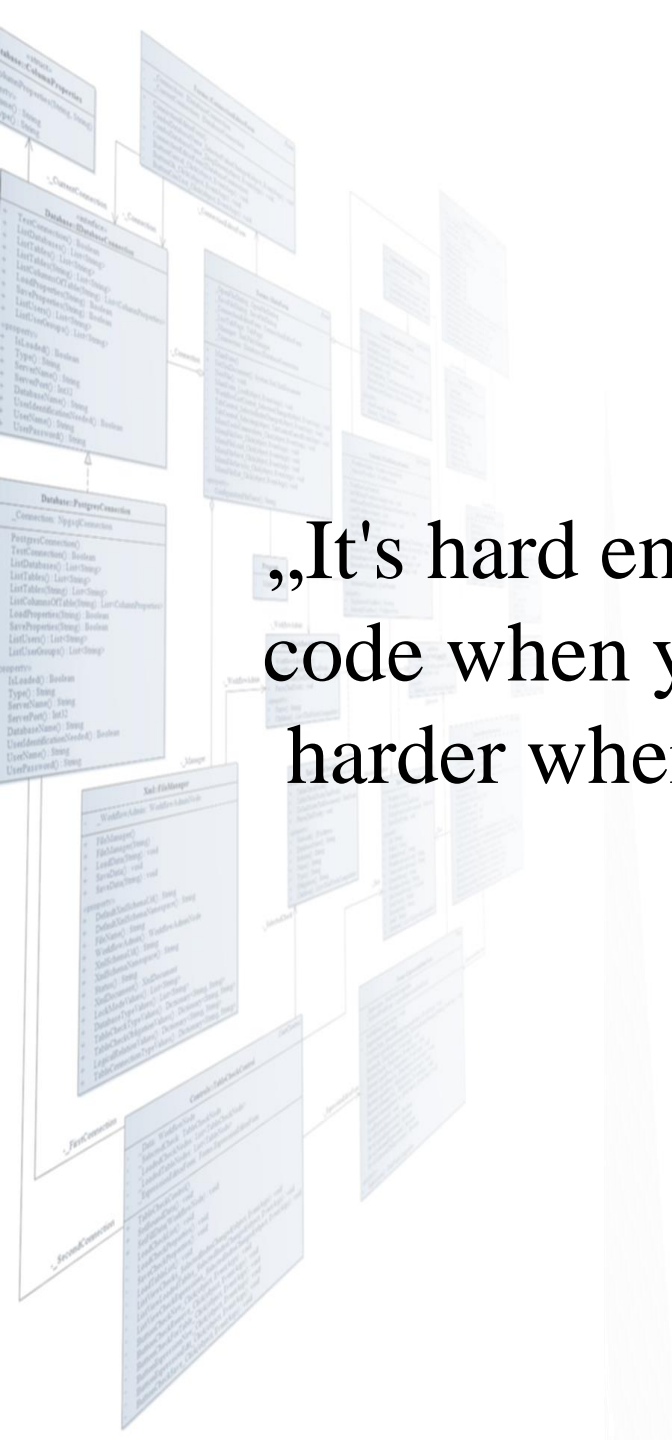
Verifikáció és validáció

Giachetta Roberto

groberto@inf.elte.hu

<http://people.inf.elte.hu/groberto>





„It's hard enough to find an error in your code when you're looking for it; it's even harder when you've assumed your code is error-free.”

(Steve McConnell)



Verifikáció és validáció

Minősegbiztosítás

- A szoftver verifikációja és validációja, vagy *minősegbiztosítása* (*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 helyáll, de felfedheti a hibákat adott körülmények között
- A teszteléshez *tesztelési tervet (test plan)* készítünk, amely ismerteti a tesztelés felelőseit, folyamatát, technikáit és céljait

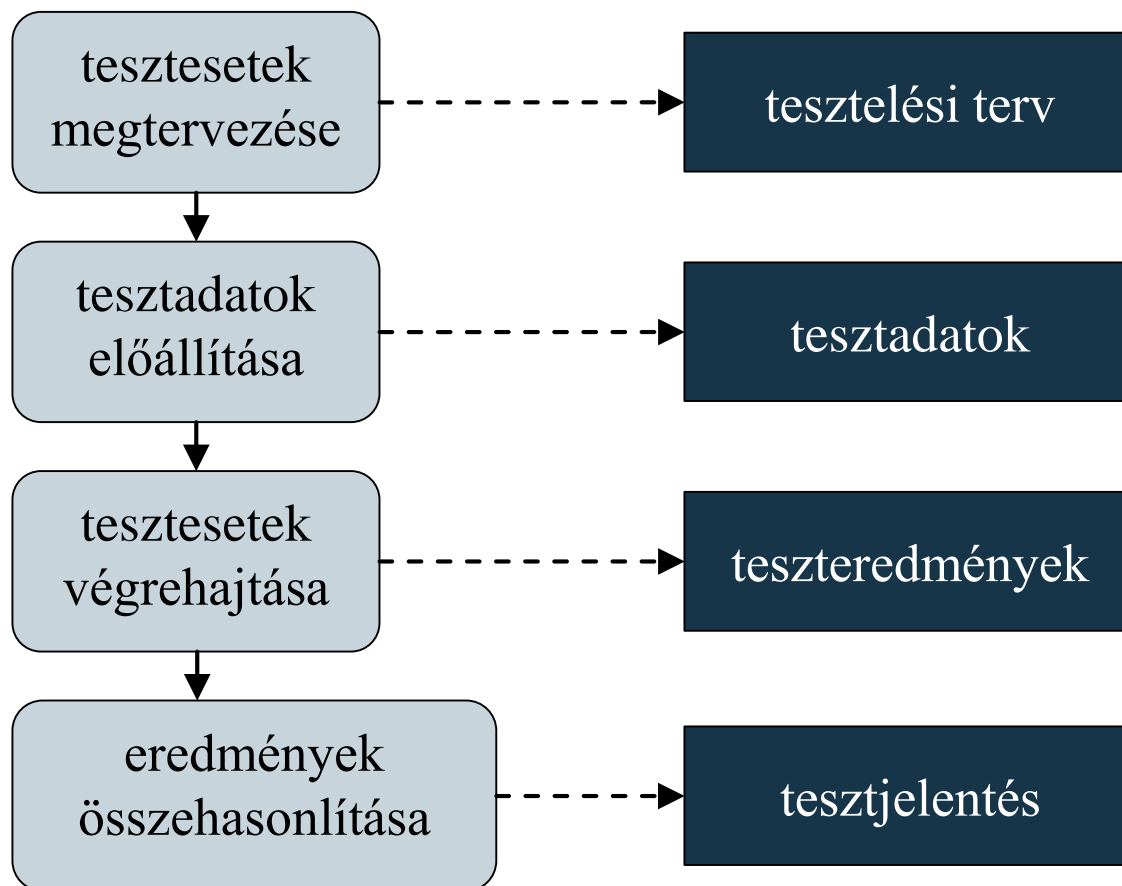
Verifikáció és validáció

Tesztesetek

- A tesztelés során különböző *teszteseteket* (*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 tesztesetek 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 tesztcsoport 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

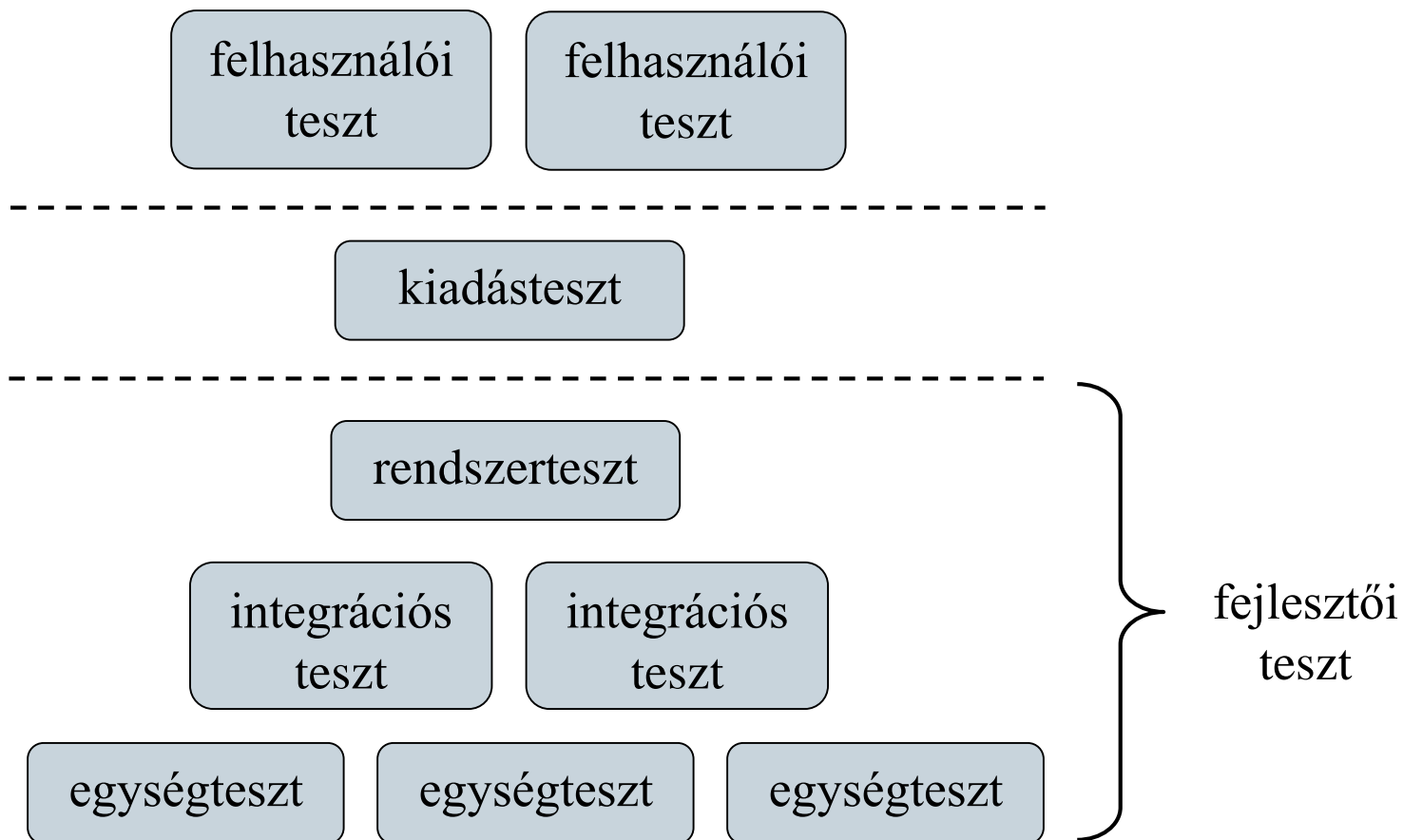
Verifikáció és validáció

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
 - *rendszer teszt (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 teszteseket célszerű lezárítani a programegység által
 - megengedett bementre, 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

Tesztesetek

- Pl.:

```
class Rectangle { // tesztelendő osztály
private:
    int _width;
    int _height;
public:
    Rectangle(int width, int height)
        : _width(width), _height(height)
    { }
    int getWidth() { return _width; }
    int getHeight() { return _width; }
    int getArea() { return _width * _height; }
}
```

Verifikáció és validáció

Tesztesetek

- Pl.:

```
void testRectangle4x6() // teszt eset
{
    Rectangle rec(4,6); // szimulált bemenő adatok

    if (rec.getWidth() != 4)
        cout << "getWidth failed";
    // ha a végrehajtás nem a várt eredményt
    // adja, hibát jelzünk
    if (rec.getHeight() != 6)
        cout << "getHeight failed";
    if (rec.getArea() != 24)
        cout << "getArea failed";
}
```

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 függetlenül kell tesztelni.

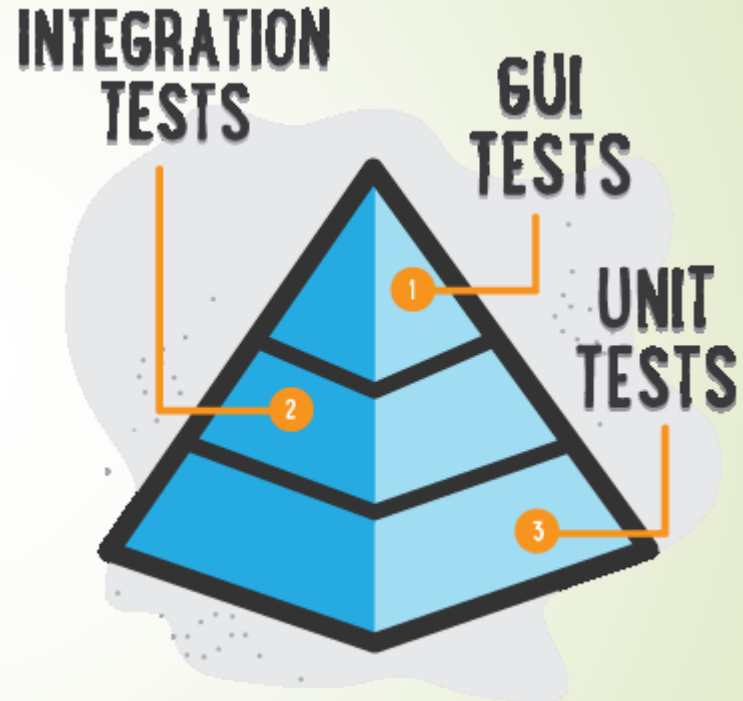


ILLUSTRATION BY SEGUE TECHNOLOGIES



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 tesztekől 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

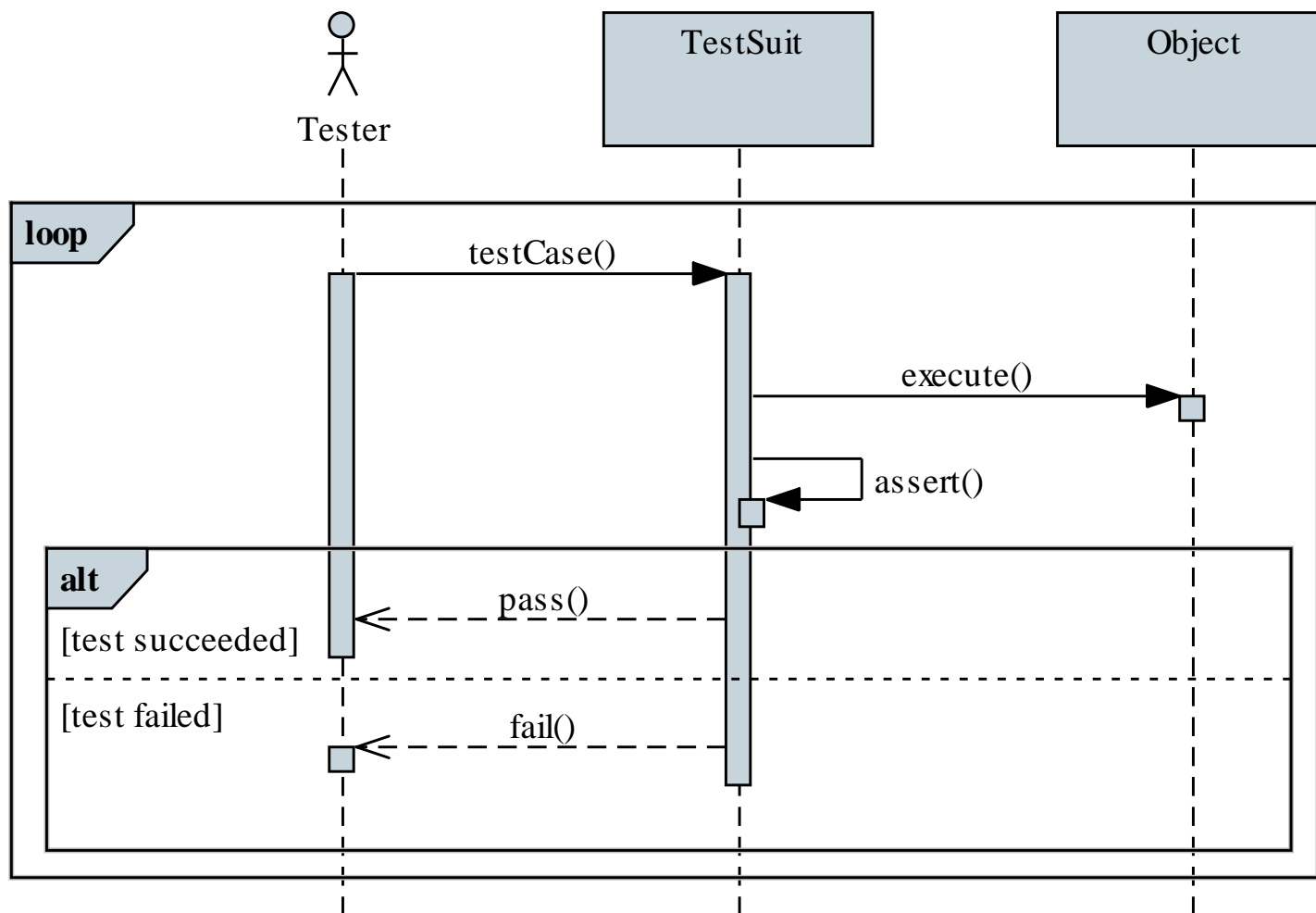
Verifikáció és validáció

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 tesztestekben 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. *CppTest*, *QTestLib*

Verifikáció és validáció

Tesztelési keretrendszerek



Verifikáció és validáció

Egységtesztek Qt keretrendszerben

- A Qt keretrendszer tartalmaz egy beágyazott tesztelő modul (*QTestLib*), amely lehetőségeket ad egységtesztek és teljesítménytesztek könnyű megfogalmazására, és végrehajtására
 - a teszteseteket **QObject** leszármazott osztályokban valósítjuk meg eseménykezelőként
 - az ellenőrzéseket makrók segítségével valósítjuk meg, pl.:
 - logikai kifejezés ellenőrzése: **QVERIFY(<kifejezés>)**
 - összehasonlítás:
QCOMPARE(<aktuális érték>, <várt érték>)
 - figyelmeztetés: **QWARN(<üzenet>)**
 - hiba: **QFAIL(<üzenet>)**

Verifikáció és validáció

Egységtesztek Qt keretrendszerben

- Pl.:

```
class RectangleTest : QObject {  
    // tesztesetek osztálya  
    Q_OBJECT  
private slots:  
    void testRectangle4x6() { // teszteset  
        Rectangle rec(4,6);  
        // szimulált bemenő adatok  
        QCOMPARE(rec.getWidth(), 4);  
        QCOMPARE(rec.getHeight(), 6);  
        QCOMPARE(rec.getArea(), 24);  
        // ellenőrzések  
    }  
}
```


Verifikáció és validáció

Egységtesztek Qt keretrendszerben

- A tesztünk futtatása részletes eredményt ad
 - tesztetenként láthatjuk az eredményt (sikeres, vagy sikertelen)
 - hiba esetén láthatjuk annak okát (az ellenőrző makró típusának megfelelően) és helyét (a makró sorát a fájlban)

pl.:

PASS : RectangleTest::testRectangle4x6()

PASS : RectangleTest::testRectangle0x0()

FAIL! : RectangleTest::...()

Compared values are not the same

Loc : [../rectangletest.cpp(106)]!

Totals: 2 passed, 1 failed, 0 skipped

Verifikáció és validáció

Kód lefedettség

- A tesztgyűjtemények által letesztelt programkód mértékét nevezzük *kód lefedettségnek (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 ellenőrzi
- 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 tesztek a várható feladatmennyiség függvényében (*load test*), vagy azon túl ellenőrizhetjük a rendszer tűrőképességét (*stress test*)
 - a teljesítményt sokszor a hardver erőforrások függvényében végezzük, amellyel megállapítható a rendszer skálázhatósága (*capacity test*)

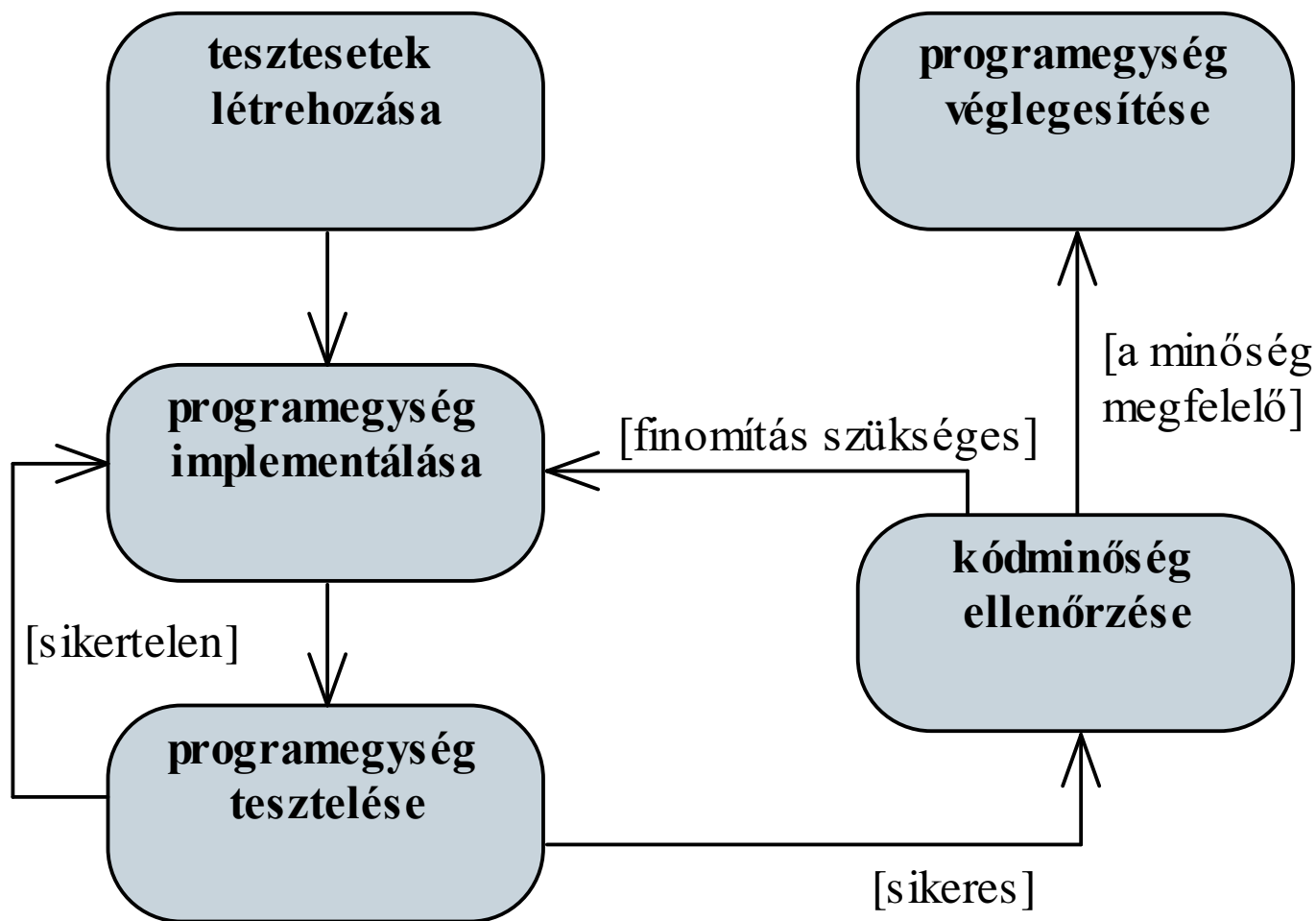
Verifikáció és validáció

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				
		<i>Követelmények</i>	<i>Tervezés</i>	<i>Implementáció</i>	<i>Tesztelés</i>	<i>Üzemeltetés</i>
Hiba helye	<i>Követelmények</i>	1x	3x	5-10x	10x	10-100x
	<i>Tervezés</i>		1x	10x	15x	25-100x
	<i>Implementáció</i>			1x	10x	10-25x

**Eötvös Loránd Tudományegyetem
Informatikai Kar**

Szoftverttechnológia

11. előadás

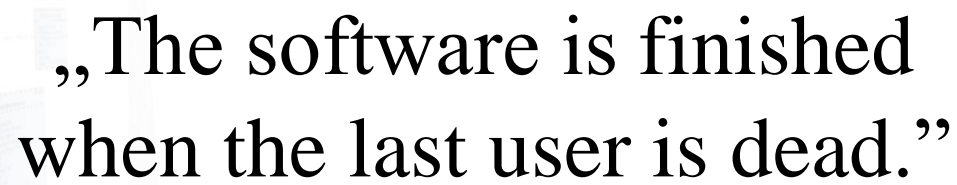
Szoftver evolúció

Giachetta Roberto

groberto@inf.elte.hu

<http://people.inf.elte.hu/groberto>





(ismeretlen)



Szoftver evolúció

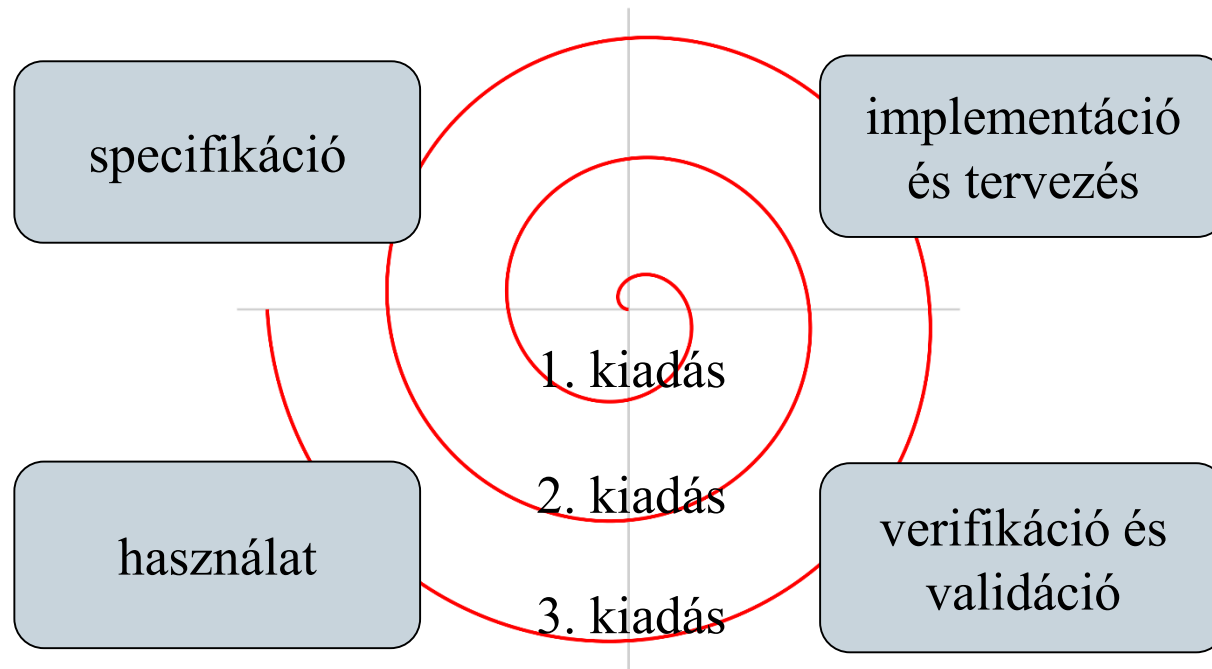
Változások kezelése

- A program használata során számos változás következhet be, amely hatására tovább kell fejlesztenünk a szoftver, ez az *evolúció* (*software evolution*)
 - átlagosan a szoftver élettartamának 80%-a
 - elsősorban külső követelmények befolyásolják
- A változás hatására a program egyes részeit újra kell tervezni, illetve megvalósítani, a változásra a projekt felkészülhet
 - a követelmények előrevetítésével (*change avoidance*), pl. prototípus létrehozásával
 - a módosítási lehetőségek nyitottan hagyásával (*change tolerance*), pl. inkrementális fejlesztéssel

Szoftver evolúció

Az evolúciós életciklus

- A szoftver evolúció lépésekben történik, amelyek a szoftver újabb kiadását (*release*) eredményezik
 - minden lépésben végig kell vinnünk a szoftvertechnológiai folyamatot, ami egy spirált ad



Szoftver evolúció

Lépések és befolyásoló tényezők

- A lépések 3 kategóriába sorolhatóak:
 - *bővítés (perfective maintenance)*: új funkciók bevezetése
 - *adaptálás (adaptive maintenance)*: a szoftver új (hardver/szoftver) környezetbe történő bevezetése
 - *hibajavítás (corrective maintenance)*: a kiadás után felmerült hibák javítása
- Az evolúció idejét, költségeit több tényező is befolyásolja:
 - a fejlesztőcsapat stabilitása
 - a dokumentáció és a programkód minősége
 - a program kora és struktúrája

Szoftver evolúció

Programverziók

- A szoftver az evolúció során több kiadott változattal (*release*) rendelkezik, amelyeket verziószámmal látunk el
 - a javításokat, funkció bővítéseket általában kisebb (*minor*) verziók jelölik
 - a jelentős módosításokat új programváltozatok (*major*) keretében kerülnek publikálásra
 - a kisebb verziók között hangsúlyos a kompatibilitás megőrzése
- A verziószám további információk egészíthetik ki, pl.
`major.minor[.maintenance[.build]]`, vagy
`major.minor[.build[.revision]]` formában

Szoftver evolúció

Lehman törvények

- A szoftver változási folyamatokra érvényes általános elveket a *Lehman törvények* fogalmazzák meg
 - egy szoftvernek változnia kell, vagy folyamatosan csökken a használhatósága és minősége
 - a változó szoftver egyre összetettebb lesz és egyre több funkcióval rendelkezik
 - az evolúciós folyamat önszabályozó, a rendszer tulajdonságai (méret, hibák száma, ...) függetlenek a kiadástól
 - az evolúciós folyamatban szükséges a visszajelzések, amelyeket figyelembe kell venni

Szoftver evolúció

Legacy rendszerek

- A *legacy rendszerek* olyan régi szoftverrendszerek, amely még mindig hasznosak és nélkülözhetetlenek
 - nem cserélhetőek újabb rendszerre, vagy cseréjük túl költséges lenne
 - általában elavult nyelven készültek elavult technológiákkal, dokumentációjuk nincs, vagy hiányos
- A legacy rendszerek esetén sokszor újra meg kell valósítani a szoftvert (*reengineering*), ami magában foglalhatja
 - a program és az adatok elemzését (*reverse engineering*)
 - a forráskód átvitelét új programnyelvre, a programstruktúra és az adatrepresentáció megváltoztatását