Eötvös Loránd Tudományegyetem
Informatikai Kar

Térinformatikai és távérzékelési alkalmazások fejlesztése

Szoftverek minőségbiztosítása

groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

Szoftverek minőségbiztosítása

Verifikáció és validáció

- 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

Szoftverek minőségbiztosítása

Verifikáció és validáció

- 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

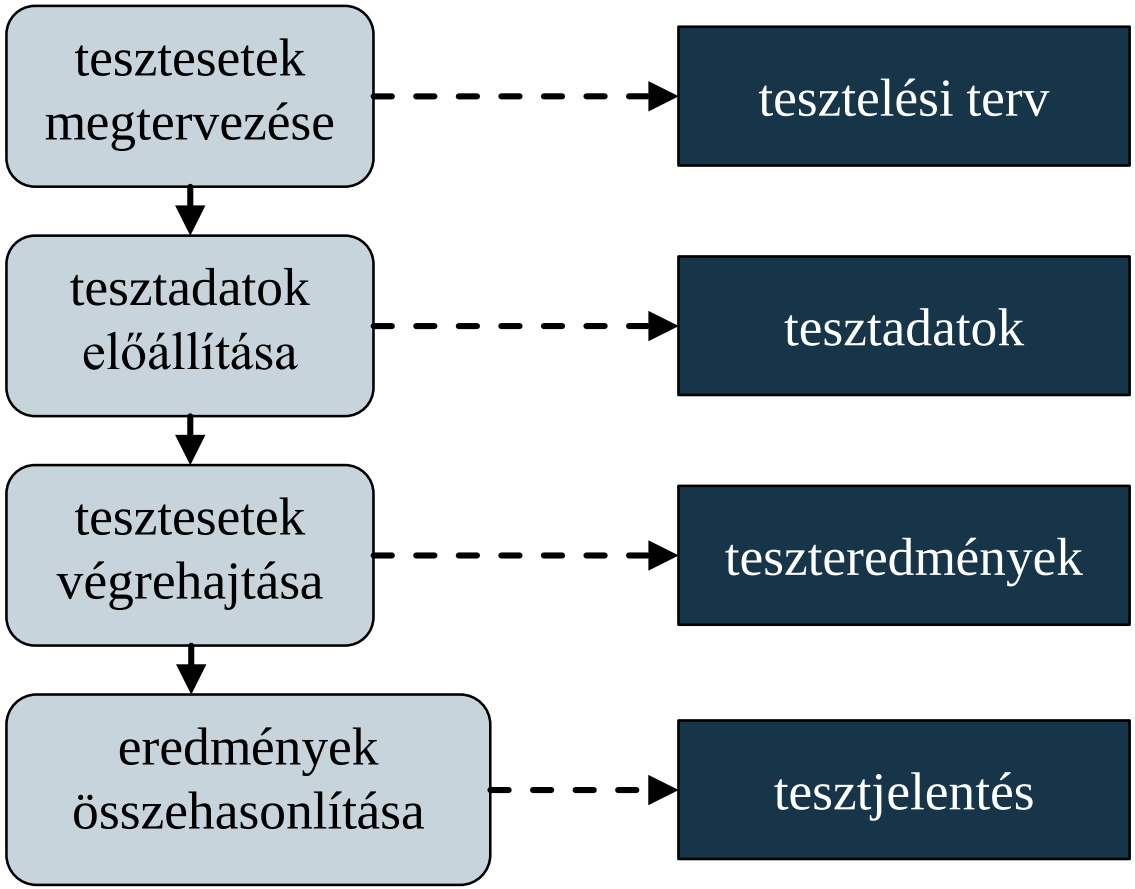
Szoftverek minőségbiztosítása

Egységtesztek

- 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 tesztesetek gyűjteményekbe helyezzük (*test suit*)
- A tesztesetek eredményeiből készül a *tesztjelentés* (*test report*)

Szoftverek minőségbiztosítása

A tesztelési folyamat



Szoftverek minőségbiztosítása

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

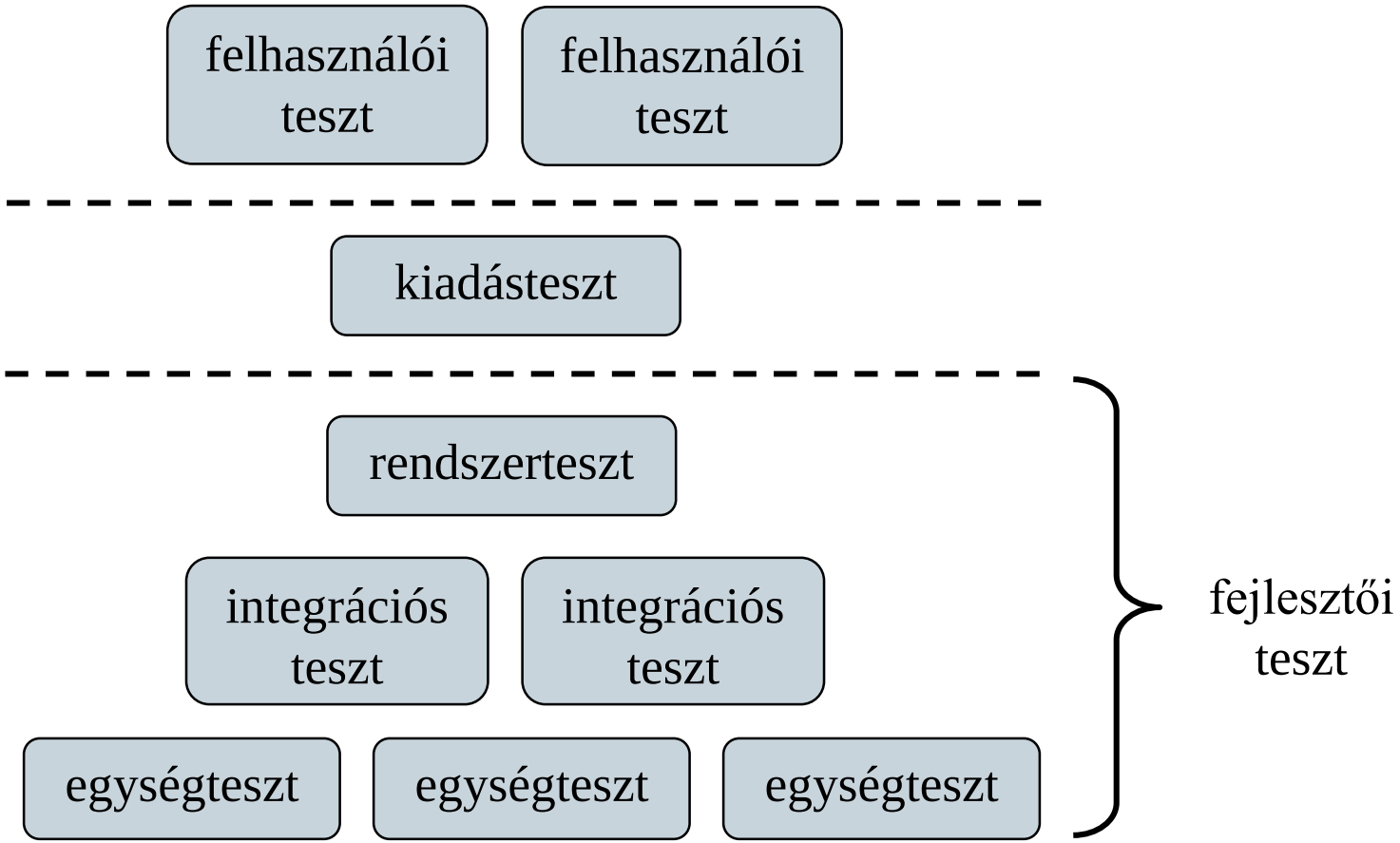
Szoftverek minőségbiztosítása

A tesztelés lépései

- A fejlesztési 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

Szoftverek minőségbiztosítása

A tesztelés lépései



Szoftverek minőségbiztosítása

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*)

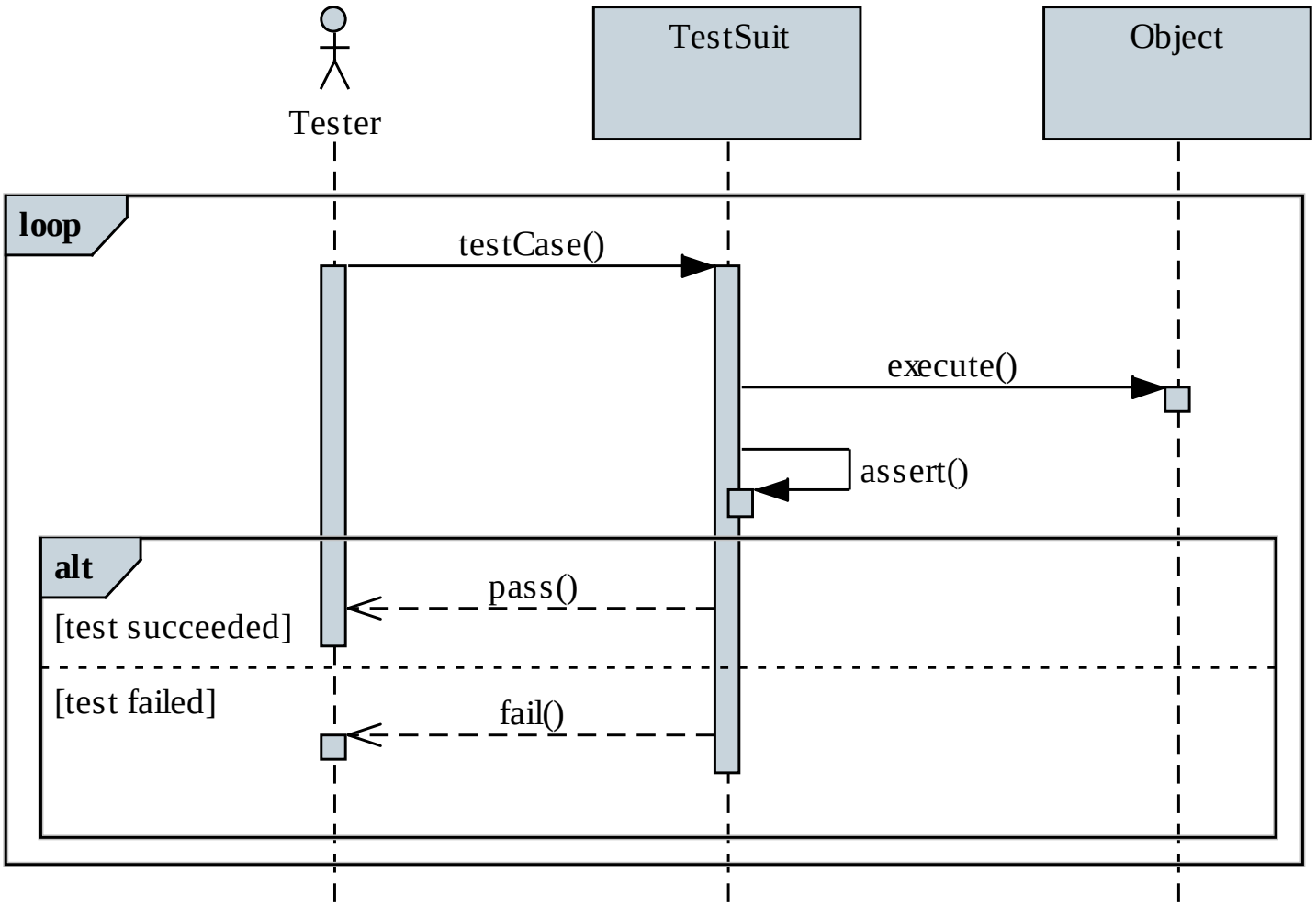
Szoftverek minőségbiztosítása

Egységtesztek

- 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 tesztesetből, akkor az eset sikeres (*pass*), egyébként sikertelen (*fail*)

Szoftverek minőségbiztosítása

Egységtesztek



Szoftverek minőségbiztosítása

Egységtesztek

- A .NET keretrendszerben az egységteszteket külön projekt keretében valósítjuk meg
  - a tesztelést végző osztályt a **`TestClass`** attribútummal, a teszteseteket a **`TestMethod`** attribútummal jelöljük
  - a tesztek az **`Assert`** osztály segítségével végeznek ellenőrzéseket (**`AreEqual`**, **`IsNotNull`**, **`IsFalse`**, **`IsInstanceOfType`**, …), és különböző eredményei lehetnek (**`Fail`**, **`Inconclusive`**)
  - lehetőségünk van a teszteket inicializálni (**`TestInitialize`**, **`TestCleanup`**)
  - a teszt környezetének (**`TestContext`**) paraméterei külön kezelhetőek

Szoftverek minőségbiztosítása

Egységtesztek

- Pl.:

```
[TestClass] // tesztosztály
public class RationalTest {
    …
    [TestMethod] // tesztművelet a konstruktorra
    public void RationalConstructorTest(){
        Rational actual = new Rational(10, 5);
        Rational target = new Rational(2, 1);
            // az egyszerűsítést teszteljük
        Assert.AreEqual(actual, target);
            // ha a kettő egyezik, akkor eredményes a
            // teszteset
    }
}
```

Szoftverek minőségbiztosítása

Egységtesztek függőségekkel

- Amennyiben függőséggel rendelkező programegységet tesztelünk, a függőséget helyettesítjük annak szimulációjával, amit *mock objektum*nak nevezünk
  - megvalósítja a függőség interfészét, egyszerű, hibamentes funkcionalitással
  - használatukkal a teszt valóban a megadott programegység funkcionalitását ellenőrzi, nem befolyásolja a függőségben felmerülő esetleges hiba
- Mock objektumokat manuálisan is létrehozhatunk, vagy használhatunk erre alkalmas programcsomagot
  - pl. .NET keretrendszerben *NSubstitute*, *Moq*

Szoftverek minőségbiztosítása

Egységtesztek függőségekkel

- Pl. :

```
class DependencyMock : IDependency
   // mock objektum
{
   // egy egyszerű viselkedést adunk meg
   public Double Compute() { return 1; }
   public Boolean Check(Double value) {
      return value >= 1 && value <= 10;
   }
}
…
Dependant d = new Dependant(new DependencyMock());
   // a mock objektumot fecskendezzük be a függő
   // osztálynak
```

Szoftverek minőségbiztosítása

Egységtesztek függőségekkel

- *Moq* segítségével könnyen tudunk interfészekből mock objektumokat előállítani
  - a **`Mock`** generikus osztály segítségével példányosíthatjuk a szimulációt, amely az **`Object`** tulajdonsággal érhető el, és alapértelmezett viselkedést produkál, pl.:

```
Mock<IDependancy> mock =
    new Mock<IDependancy>();
    // a függőség mock objektuma
Dependant d = new Dependant(mock.Object);
    // azonnal felhasználható
```

  - a **`Setup`** művelettel beállíthatjuk bármely tagjának viselkedését (**`Returns(…)`**, **`Throws(…)`**, **`Callback(…)`**), a paraméterek szabályozhatóak (**`It`**)

Szoftverek minőségbiztosítása

Egységtesztek függőségekkel

- pl. :

```
mock.Setup(obj => obj.Compute()).Returns(1);
   // megadjuk a viselkedést, mindig 1-t ad
   // vissza
mock.Setup(obj =>
            obj.Check(It.IsInRange<Double>(0, 10,
                        Range.Inclusive)))
    .Returns(true);
mock.Setup(obj => obj.Check(It.IsAny<Double>())
    .Returns(false);
    // több eset a paraméter függvényében
…
```

- lehetőségünk van a hívások nyomkövetésére (**`Verify(…)`**)

Szoftverek minőségbiztosítása

Integrációs tesztek

- *Integrációs tesztek* (*I&T*) során a függőségeket már nem szimuláljuk, hanem közvetlenül használjuk, így tesztelhető az egyes komponensek együttműködése
  - automatizálható az egységtesztelés eszközeivel, csupán a függőségeket is fel kell használnunk
  - lehetőség van a felhasználói interakció szimulálására is (pl. *Coded UI Test*, *SpecsFor.Mvc*)
  - megközelítésben lehet:
    - *alulról felfelé* (*bottom-up*), az alsó rétegekből építi fel az alkalmazást, így nincs szükség függőség szimulációra
    - *felülről lefelé* (*top-down*): a felső rétegekből indul, és lépésekben csökkenti a szimulációt

Szoftverek minőségbiztosítása

Viselkedés alapú tesztelés

- Viselkedés alapú fejlesztés (BDD) esetén a szoftvernek adott viselkedési mintáknak kell megfelelniük, amelynek elemei
  - a történet (*story*), amely megadja a viselkedés célját (*in order to*), a felhasználói szerepet (*as a*) és tevékenységet (*I want to*)
  - a történeten belüli forgatókönyvek (*scenario*), ahol adott kiindulási állapotból (*given*) egy tevékenység végrehajtása (*when*) hatására szeretnénk eljutni egy állapotba (*then*)
- A történetek megfeleltethetőek tesztkörnyezetnek, forgatókönyvek pedig teszteseteknek, így jól illeszthetőek a tesztek eredményei a követelményekhez

- Pl.:

```
Story: Tic-Tac-Toe game
In order to play the game
As a player
I want to make steps on the game table

Scenario 1: First step
Given a newly created game table
And me being the first player
When I step on the first field
Then an X should appear on first field
```

- Pl.:

```
[TestClass]
public class TicTacToeTest {
    [TestMethod]
    public void FirstStepTest()
    {
        // given
        GameTable table = new GameTable();
        // when
        table.Step(0, 0);
        // then
        Assert.AreEqual("X", table.GetField(0, 0));
    }
}
```

Szoftverek minőségbiztosítása

Statikus kódelemezés

- A *statikus kódelemzés* (*static code analysis*) lehetővé teszi, hogy a forráskódot még a fordítás előtt előfeldolgozzuk, és a lehetséges hibákat és problémákat előre feltérképezzük
  - a fejlesztőkörnyezet beépített *kódelemző*vel rendelkezhet, amely megadott szabályhalmaz alapján a lehetséges hibaeseteket felfedi
  - a statikus kód elemzés egy része kimondottan a kódolási konvenciók (pl. elnevezések, tagolás, dokumentáltság) ellenőrzését biztosítja
  - a kódra számíthatók *metrikák* (code metric), amelyek megadják karbantarthatóságának, összetettségének mértékét (pl. cyclomatic complexity, class coupling)

Szoftverek minőségbiztosítása

Kódolási stílus

- A *kódolási stílus* (*coding style*) egy szabályhalmaz, amely a forráskód megjelenésére ad iránymutatást
  - a kódolási stílus követése javítja a kód értelmezhetőségét, a későbbi karbantartást (a program életciklusának 80%-t is kiteheti)
  - a kódolási stílus lehet nyelvi szinten, vállalati szinten, vagy szoftverszinten rögzített
    - így a fejlesztők közötti kommunikáció zökkenőmentes
    - pl. CamelCase, magyar jelölés, K&R, 1TBS
  - a jó programozási stílus általában szubjektív, nem túl szigorú, de alapvető elemeket definiál

Szoftverek minőségbiztosítása

Kódolási stílus

- Pl.:

```
class Point { // camel case (upper, Pascal case)
private:
   int xCoordinate; // camel case (lower)
   int yCoordinate;
   …
   double DistanceTo(Point p) const
      // camel case (upper)
   {
      return …
   } // K&R

   …
};
```

Szoftverek minőségbiztosítása

Kódolási stílus

- Pl.:

```
class Point {
private:
   int x_coordinate; // snake case
   int y_coordinate;
   …
   double Distance_To(Point p) { // Oxford case
      return …
   } // 1TBS
   …
};
```

Szoftverek minőségbiztosítása

Kódolási stílus

- Általános érvényű javaslatok:
  - kódrészletek megfelelő elválasztása (szóköz, sortörés, behúzás, függőleges igazítás)
  - beszédes és konzisztens elnevezések használata (kevesebb kommentezést igényelnek)
  - beégetett tartalmi elemek (számok, szövegek) megnehezítik a karbantartást (*hard coding*), ezért célszerű a kerülése, kiemelése fejlécbe, vagy konfigurációs fájlba (*soft coding*)
- A kódolási konvenció rákényszeríthető a programozóra kódolási stílus ellenőrző eszköz segítségével
  - pl. *C++Test*, *StyleCop*

Szoftverek minőségbiztosítása

Kommentezés

- A kódot a stílusnak megfelelő kommenttel kell ellátni
  - alapvető fontosságú a felület kommentezése (osztályok, függvények, paraméterek)
  - a megvalósítás kommentezése összetett funkcionalitás esetén hasznos, de megfelelő kódolási stílus esetén nem szükséges
  - tartalmazhat speciális jelöléseket (pl. **`TODO`**, **`FIXME`**)
  - a túl kevés, vagy túl sok komment is ártalmas lehet
- A kommentek felhasználhatóak dokumentáció előállítására is (pl. *Doxygen*), amennyiben azokat megfelelő séma szerint hozzuk létre

Szoftverek minőségbiztosítása

Kommentezés

- Pl.:

```
// a type representing a 2D point
class Point {
   …
   // computes the Euclidean distance to another
   // point
   double DistanceTo(Point p) const
   {
      return sqrt(pow(…) + pow(…));
   }
   …
};
```

Szoftverek minőségbiztosítása

Kommentezés

- Pl.:

```
// Name: Point
//
// Purpose: This type represents a point in a 2D
// coordinate system.
// Remarks: Is based on double coordinates.
//
// License: LGPL v2.
//
// Author: Roberto Giachetta
// Date: 27/11/2014
// Contact: groberto@inf.elte.hu
…
class Point {
```

Szoftverek minőségbiztosítása

Kommentezés

- Pl.:

```
// Name: distance
// Purpose: This method computes the Euclidean
// distance to another Point instance.
// Remarks: This a query method.
// Parameters: p : another point
// Return value: The distance to the other.
double distance(Point p) const
{
   // uses sqrt and pow functions from math.h
   // formula: …
   return sqrt(pow(…) + pow(…));
}
…
```

Szoftverek minőségbiztosítása

Kódolási stílus

- Általában nyílt és zárt programkódokra más szabályok vonatkoznak
  - *nyílt forráskód* esetén törekedni kell, hogy a kód minél gyorsabban értelmezhető legyen bárki számára
    - követni kell a programozási nyelv tördelési és elnevezési konvencióit
    - a kód megfelelő mennyiségű megjegyzéssel kell ellátni
  - *zárt forráskód* esetén a cél a fejlesztőcsapat minél nagyobb rálátása a kódra
    - törekedni kell, hogy minél nagyobb kódmennyiség legyen egyszerre áttekinthető (kevesebb helyköz és komment)

Szoftverek minőségbiztosítása

Kód-újratervezés

- A *kód-újratervezés* (*refactoring*) célja, hogy a kód szerkezete úgy módosuljon, hogy külső viselkedését ezzel nem befolyásoljuk
  - pl. átnevezések, ismétlődő kódok kiemelése, típuscsere, interfész kiemelése, tervezési minta bevezetése
  - ezzel a kód nem funkcionális követelményeit javíthatjuk
  - általában két a karbantarthatóság, illetve a bővíthetőség növelése a cél
  - manuálisan is elvégezhető, de ez hibákhoz vezethet, így erre a célra *kód-újratervező* (*refactoring*) eszközöket alkalmazzunk