



**Eötvös Loránd Tudományegyetem
Informatikai Kar**

Eseményvezérelt alkalmazások fejlesztése I

11. előadás

Szoftverek tesztelése

© 2014 Giachetta Roberto
groberto@inf.elte.hu
<http://people.inf.elte.hu/groberto>

Szoftverek tesztelése

Tesztelés

- A *tesztelés* annak ellenőrzése, hogy a program teljesíti-e az előírt minőségi mutatókat
 - futási idejű hibák, működési rendellenességek keresése, kompatibilitások ellenőrzése
 - a tesztelésnek négy szakasza van:
 - *egységteszt*: a programegységeket külön-külön teszteljük (ami könnyen végezhető az implementáció közben is)
 - *integrációs teszt*: a programegységek felületének, kommunikációjának ellenőrzése
 - *rendszereszt*: az egész rendszer együttes tesztje
 - *rendszerintegrációs teszt*: a rendszer tesztelése a kihelyezett környezetben

Szoftverek tesztelése

Tesztelés

- a tesztelés módja szerint lehet:
 - *fekete doboz tesztelés*: a tesztelendő programrész ismeretlen, csak a hiba voltát fedezzük fel
 - *fehér doboz tesztelés*: a programrész teljes mértékben ismert, tehát így a hiba helyét is megtalálhatjuk
 - *szürke doboz tesztelés*: a belső adatok, illetve egyes algoritmusok elérhetőek, így a hiba helye behatárolható
- a tesztelés módszere lehet:
 - *statikus*: kód kiértékelés és ellenőrzés (eszköze a statikus kódelemzés)
 - *dinamikus*: a program futtatása adott tesztesetekkel (eszköze a nyomkövetés és az egységteszt)

Szoftverek tesztelése

Nyomkövetés

- *Nyomkövetés (debugging)* 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 tesztelése

Egységtesztek

- Az *egységteszt* (*unit test*) egy olyan automatikusan futtatható ellenőrzés, amely lehetőséget osztályok és objektumok viselkedésének ellenőrzésére (a tényleges viselkedés megegyezik-e az elvárttal)
 - lehetőséget adnak tesztesetek gyors, sorozatos futtatására
 - automatikusan végrehajthatóak, paramétereizhetők (amennyiben a környezet támogatja)
 - egységteszteket elsősorban a logikát megvalósító rétegekre (modell, adatkezelésre) írunk
 - alapvető eszköze a *tesztvezérelt fejlesztésnek* (*Test Driven Development, TDD*)

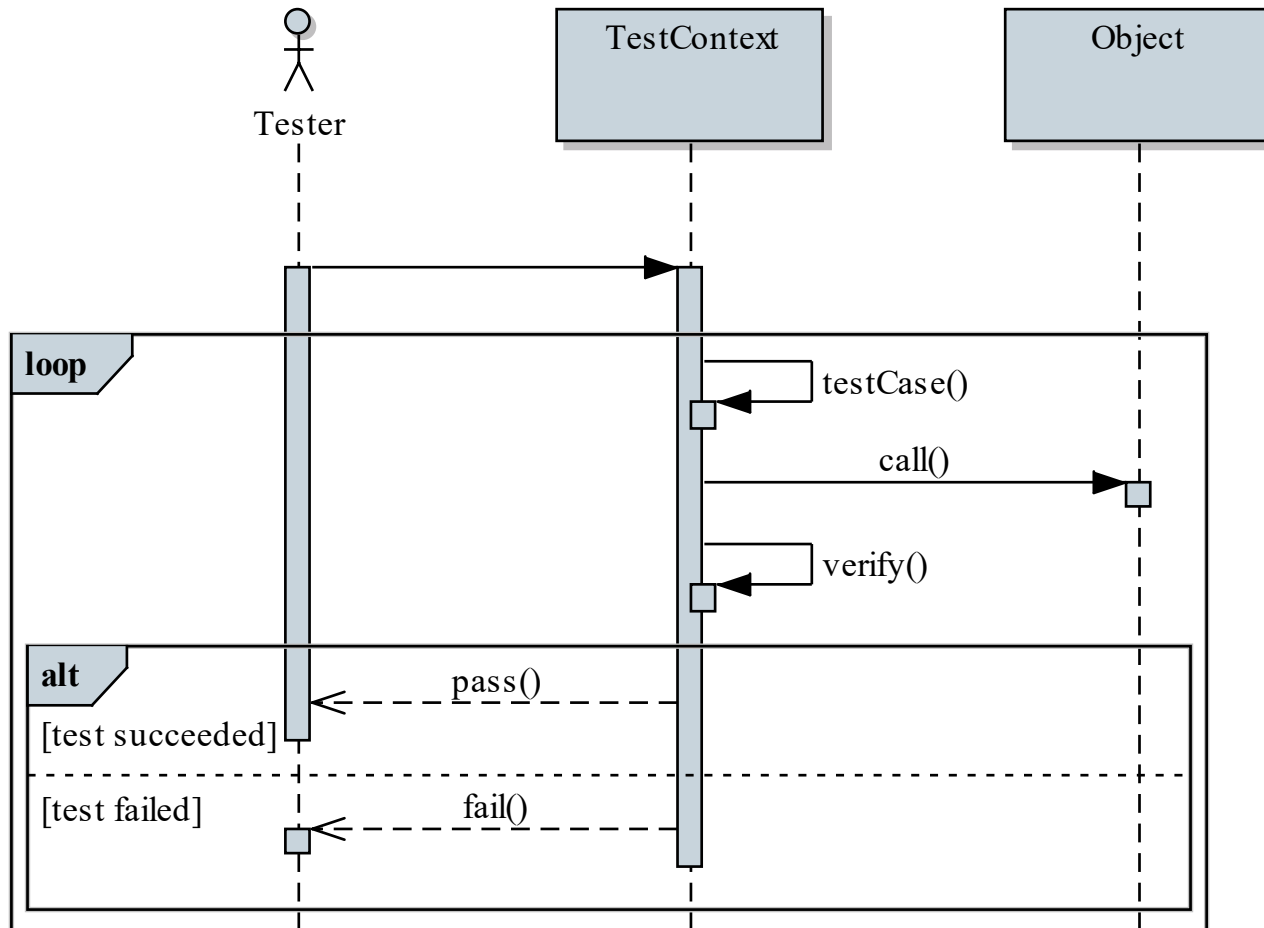
Szoftverek tesztelése

Egységtesztek

- Az egységtesztek az ellenőrzéseket adott *tesztkörnyezetben* (*test context*) végzik, amelyeket egy osztállyal reprezentálunk
 - általában egy tesztelendő osztályhoz egy tesztkörnyezet tartozik, annak interfészét (publikus metódusait) ellenőrzi
 - a tesztkörnyezet több *tesztesetet* (*test case*) ad meg, általában egy teszteset egy működés ellenőrzésére koncentrál
 - a tesztestekben egy, vagy több ellenőrzés kap helyet, amelyek jelezhetnek hibákat
 - amennyiben egy hibajelzést sem kaptunk egy tesztesetből, akkor az eset sikeres (*pass*), egyébként sikertelen (*fail*)

Szoftverek tesztelése

Egységtesztek



Szoftverek tesztelése

Tesztelés Qt keretrendszerben

- A Qt keretrendszer tartalmaz egy beágyazott tesztelő modul (*QTestLib*), amely lehetőségeket ad egységtesztek és teljesítménytesztek könnyű megfogalmazására, és végrehajtására
 - a tesztekhez szükséges funkciókat a `QtTest` fájlban találjuk
 - a tesztkörnyezetet `QObject` leszármazott osztályokban valósítjuk meg (amelyeket ellátunk `Q_OBJECT` makróval)
 - a tesztesetek eseménykezelők lesznek, amelyekben ellenőrzéseket végzünk
 - a projektben megjelöljük a modul használatát (`QT += testlib`)

Szoftverek tesztelése

Tesztelés Qt keretrendszerben

- Az ellenőrzéseket makrók segítségével valósítjuk meg, pl.:
 - logikai kifejezés ellenőrzése: `QVERIFY (<kifejezés>)`
 - összehasonlítás:
`QCOMPARE (<aktuális érték>, <várt érték>)`
 - hiba: `QFAIL (<üzenet>)`
 - figyelmeztetés: `QWARN (<üzenet>)`
- A teszt futtatását a `QTEST_MAIN (<osztálynév>)` (vagy `QTEST_MAIN (<osztálynév>)`) makróval végezhetjük, amely automatikusan legenerál egy főprogramot, és végrehajtja a teszteseteket, így a tesztek egyszerű konzolos alkalmazásként futtathatóak

Szoftverek tesztelése

Tesztelés Qt keretrendszerben

- Pl.:

```
class MyClass {  
    // tesztelendő osztály  
private:  
    int _value;  
  
public:  
    // a publikus műveleteket teszteljük  
    MyClass (int v) { _value = v; }  
    void add(int v) { _value += v; }  
    int getValue() const { return _value; }  
}
```

Szoftverek tesztelése

Tesztelés Qt keretrendszerben

```
class MyClassTest : QObject {
    // tesztkörnyezet
    Q_OBJECT
private slots:
    // tesztesetek, mint eseménykezelők

    void testGetValue()
    {
        MyClass mc(10);
        // végrehajtunk egy ellenőrzést:
        QVERIFY(mc.getValue() == 10);
        // másként:
        // QCOMPARE(mc.getValue(), 10);
    }
}
```

Szoftverek tesztelése

Tesztelés Qt keretrendszerben

```
void testAdd()
{
    MyClass mc(10);
    mc.add(5);
    QCOMPARE(mc.getValue(), 15);

    mc.add(15);
    QCOMPARE(mc.getValue(), 30);
    // tetszőleges sok ellenőrzést
    // végezhetünk
}
...
}
```

Szoftverek tesztelése

Tesztelés Qt keretrendszerben

- A Qt Creator biztosít egy teszt projekt típust (*Qt Unit Test*)
 - létrehozza a megadott tesztkörnyezetet, valamint a főprogram generátort (egy forrásfájlban)
- A tesztünk futtatása részletes eredményt ad, tesztesetenként láthatjuk az eredményt, az esetleges hibajelenséget, valamint a hiba helyét, pl.:

```
PASS      : MyClassTest::testGetValue ()
```

```
PASS      : MyClassTest::testAddValue ()
```

```
FAIL!     : MyClassTest:... ()
```

```
    Compared values are not the same
```

```
    Loc : [../MyTest/myclasstest.cpp(106)]!
```

```
Totals: 2 passed, 1 failed, 0 skipped
```

Szoftverek tesztelése

Tesztelés Qt keretrendszerben

- Lehetőségünk van a tesztkörnyezet konfigurálására
 - a tesztkörnyezetben mezőként bármilyen adatot eltárolhatunk, ezeket speciális eseménykezelőkkel állíthatjuk
 - az első teszteset előbb lefut a tesztkörnyezet inicializálás (**initTestCase**)
 - az utolsó teszteset után lefut a tesztkörnyezet megsemmisítés (**cleanupTestCase**)
 - minden teszt előtt lefut a teszteset inicializálás (**init**)
 - minden teszt után lefut a teszteset megsemmisítés (**cleanup**)

Szoftverek tesztelése

Tesztelés Qt keretrendszerben

- Pl.:

```
class MyClassTest : QObject {
    Q_OBJECT
private: // a tesztkörnyezet értékei
    MyClass* _mc;
private slots:
    void initTestCase() { // inicializálás
        _mc = new MyClass(10);
    }
    void cleanupTestCase() { // megsemmisítés
        delete _mc;
    }
    // _mc az összes tesztesetben elérhető lesz
    ...
}
```

Szoftverek tesztelése

Példa

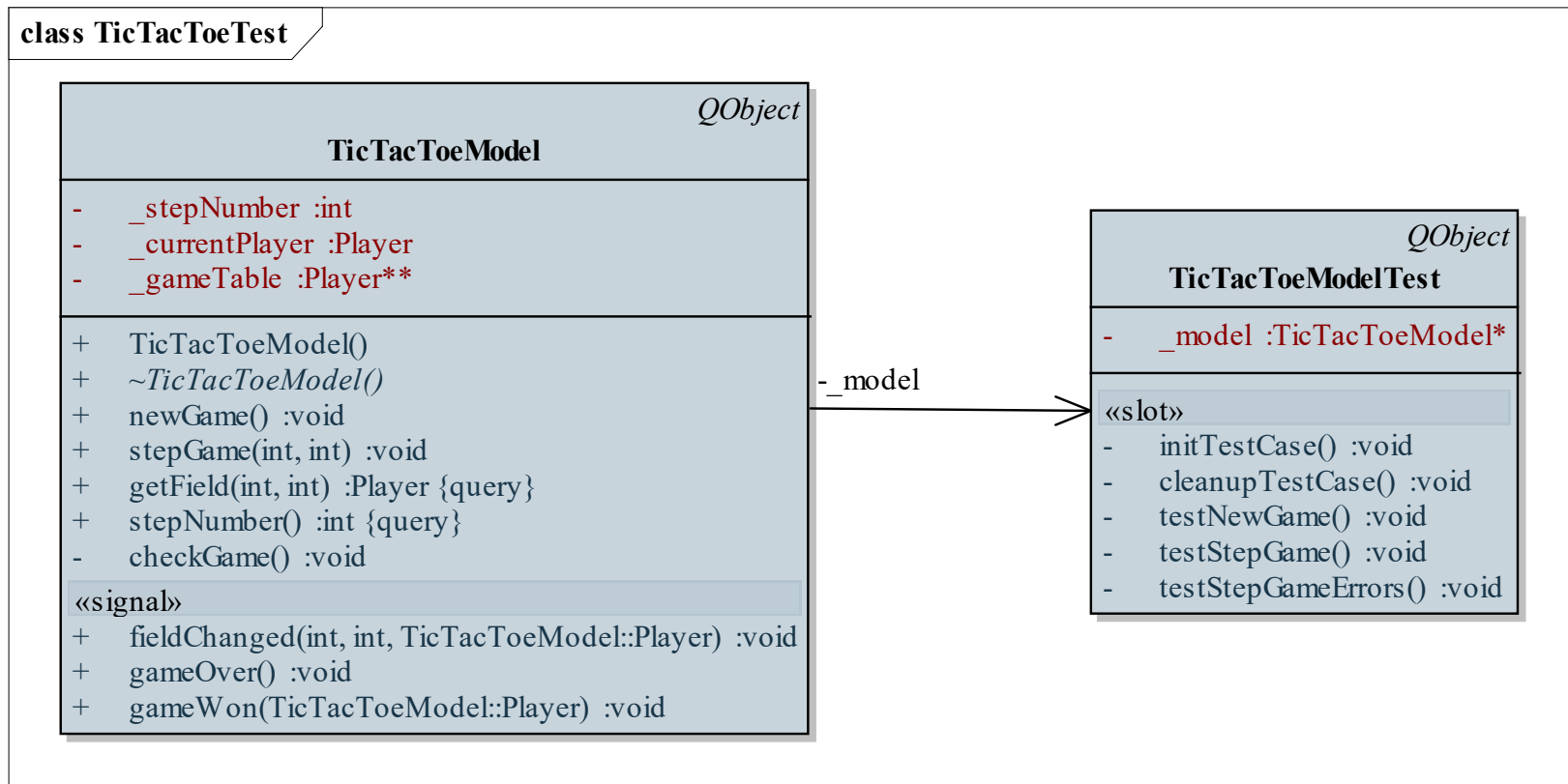
Feladat: Teszteljük le a Tic-Tac-Toe játék kétrétegű megvalósításának modelljét.

- létrehozunk egy tesztprojektet, amelybe bemásoljuk a **TicTacToeModel** osztályt
- létrehozunk egy tesztkörnyezetet (**TicTacToeModelTest**), amelyben teszteljük új játék kezdését (**testNewGame**), lépések végrehajtását (**testStepGame**)
- a tesztkörnyezet tárolja a modell egy példányát, amelyet inicializál (**initTestCase**), majd megsemmisít (**cleanupTestCase**)

Szoftverek tesztelése

Példa

Tervezés (architektúra):



Szoftverek tesztelése

Példa

Megvalósítás (tictactoemodeltest.cpp):

```
void TicTacToeModelTest::testNewGame ()
{
    _model->newGame ();

    // ellenőrizzük, hogy kezdetben minden mező
    // üres és 0 a lépésszám
    QCOMPARE (_model->stepNumber (), 0);

    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            QCOMPARE (_model->getField(i, j),
                TicTacToeModel::NoPlayer);
}
```