Modell tesztelése

Tesztelés célja és módja

- A tesztelés célja a szoftverhibák felfedezése és a szoftverrel szemben támasztott minőségi elvárások ellenőrzése.
- A tesztelés során különböző teszteseteket (*test case*) vizsgálunk, amelyek az egyes funkciókat, illetve elvárásokat tudják ellenőrizni:
  - megadjuk, hogy adott bemenő adatokra mi a várt eredmény (*expected result*), amelyet a teszt lefutása után összehasonlítunk a számított eredménnyel (*actual result*).
- A tesztelés egy fontos része a szoftverfejlesztésnek, de különleges szerepet tölt be egy speciális szoftverfejlesztési módszerban, a teszt-vezérelt szoftverfejlesztésben (*Test Driven Development*, *TDD*). Ennek során egy programegység elkészítése előtt tesztesetek formájában specifikáljuk a megoldandó részfeladatot, majd a programegység elkészülte után azt ezek alapján azonnal teszteljük.

Tesztelés szakaszai

- A tesztelés általában 3 lépésből áll:
  - fejlesztői teszt (*development testing*),
  - kiadásteszt (*release testing*),
  - felhasználói teszt (*acceptance testing*).
- A fejlesztői tesztnek további három 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

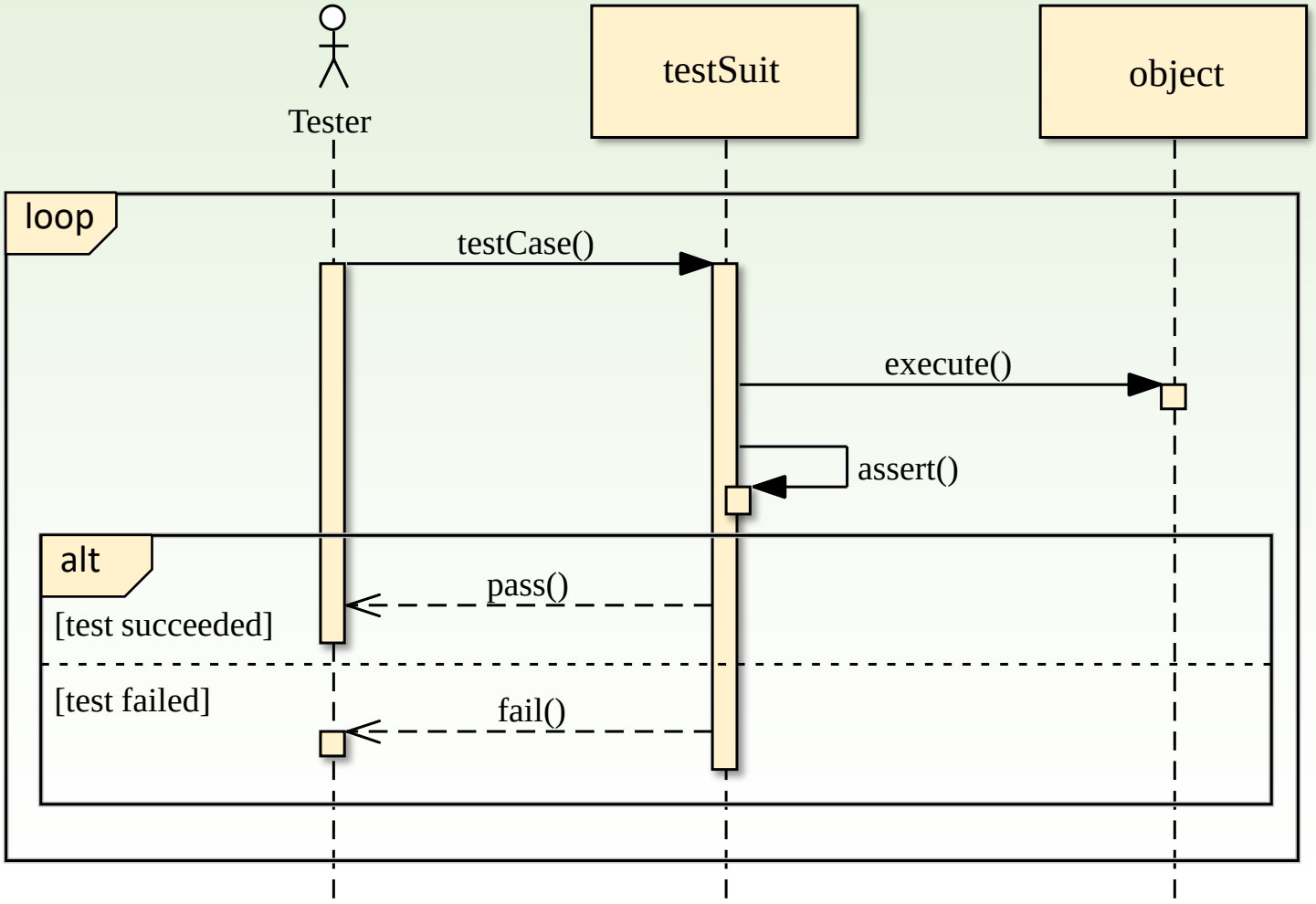
M/V architektúra tesztelése

- A modell-nézet architektúrájú alkalmazásknál külön teszteljük a modellt, és külön a nézetet.
- A modell egységeinek tesztje, ahol a várt kimenet jól definiálható értékekként adható meg, könnyen automatizálható, csak megfelelő keretrendszer kell hozzá.
- A nézet tesztjét (bér léteznek a kereskedelemben a felhasználói felület tesztelésére szolgáló rendszerek) többnyire manuálisan végzik. Ennek alapját a felhasználói esetek részletes leírása (lásd user story vagy given-when-then táblázat) adja. A táblázat „when” rubrikájában leírt akciónak a részletezésével egy felhasználói eset több tesztesetre bontható, és ezek kipróbálásával végezhető el a tesztelés.

Automatikus tesztelés

- Az egységtesztek automatizálását, és az eredmények kiértékelését teszik hatékonyabbá a tesztelési keretrendszerek (*unit testing frameworks*).
  - Ilyenkor a tényleges főprogramtól függetlenül építhetünk teszteseteket, amelyeket futtatva, képet kapunk a tesztelt programegység minőségéről.
  - Egy tesztesetben egy, vagy több ellenőrzés (*assert*) kap helyet, amelyek jelezhetnek hibákat.
  - Amennyiben nem kapunk hibajelzést egy tesztesetből, akkor az a teszteset sikeres (*pass*), egyébként sikertelen (*fail*).

Egységtesztek



Tesztelés Qt keretrendszerben

- A Qt keretrendszer tartalmaz egy beágyazott tesztelő modult (*QTestLib*), amely lehetőségeket ad egységtesztek és teljesítménytesztek könnyű megfogalmazására, és végrehajtására.
  - A tesztekhez szükséges funkciókat a **`QtTest`** könyvtárban találjuk.
  - A tesztkörnyezetet **`QObject`** leszármazott osztályokban valósítjuk meg (amelyeket ellátunk **`Q_OBJECT`** makróval).
  - A teszteseteket tesztkörnyezet szignálok formájában váltja ki, és azok eseménykezelőiben ellenőrzhetjük a teszteseteket.
  - A projektben megjelöljük a modul használatát (**`QT += testlib`**).

Makrók és futtatás

- 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 a **`QTEST_APPLESS_MAIN(<osztálynév>)`** makró végzi, amely automatikusan legenerál egy főprogramot, és végrehajtja a teszteseteket (kiváltja a teszteseteket előidéző szignálokat), így a tesztek egyszerű konzolos alkalmazásként futtathatók.

Példa tesztelendő osztályra

tesztelendő osztály

publikus metódusokat teszteljük

```
class MyClass {
private:
    int _value;
public:
    MyClass (int v) { _value = v; }

    void add(int v) { _value += v; }
    int getValue() const { return _value; }
}
```

Példa tesztkörnyezetre

```
class MyClassTest : QObject {
    Q_OBJECT
private slots:
    void testGetValue() {
        MyClass mc(10);
        QVERIFY(mc.getValue() == 10);
        // másképp: QCOMPARE(mc.getValue(), 10);
    }
    void testAdd() {
        MyClass mc(10);
        mc.add(5);
        QCOMPARE(mc.getValue(), 15);
        mc.add(15);
        QCOMPARE(mc.getValue(), 30);
    }
    …
}
```

tesztkörnyezet

tesztesetek, mint eseménykezelők

tetszőleges sok ellenőrzést végezhetünk

Tesztprojekt

- A QtCreator biztosít egy teszt projekt sablont (*Qt Unit Test*).
  - Létrehoz a megadott tesztkörnyezetet, valamint generál annak futtatására szolgáló főprogramot.
- A tesztünk futtatása részletes eredményt ad, tesztesetenként láthatjuk az eredményt, az esetleges hibajelenséget, valamint a hiba helyét:

```
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
```

Tesztkörnyezet beállítása

- Lehetőségünk van a tesztkörnyezet konfigurálására:
  - A tesztkörnyezetet adó osztály adattagjaként bármilyen adatot eltárolhatunk.
  - Az adattagok értékét speciális eseménykezelőkkel állíthatjuk:
    - az első teszteset előtt 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`**)

Példa tesztkörnyezet beállítására

```
class MyClassTest : QObject {
    Q_OBJECT
private:
    MyClass* _mc;
private slots:
    void initTestCase() {
        _mc = new MyClass(10);
    }
    void cleanupTestCase() {
        delete _mc;
    }
    …
```

tesztkörnyezet értékei

inicializálás

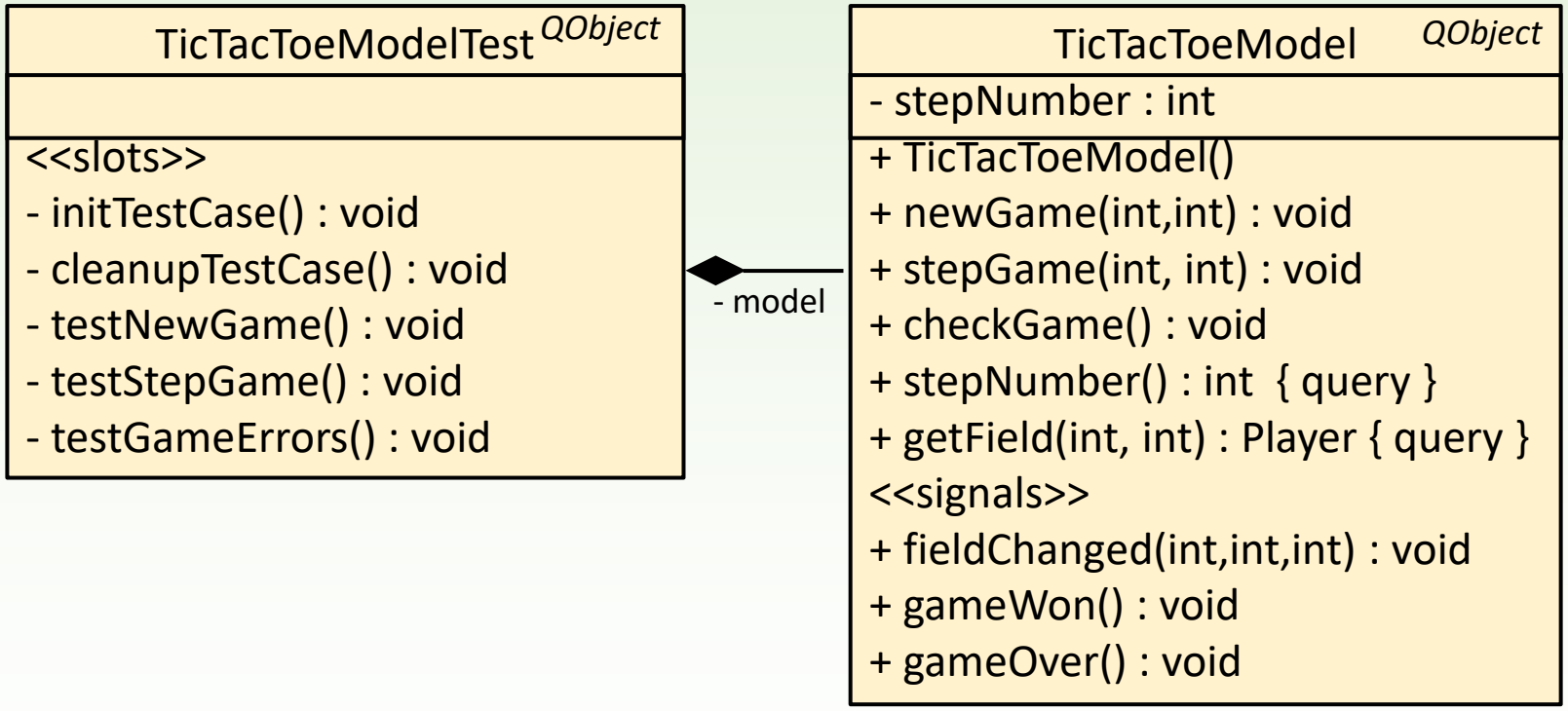
megsemmisítés

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 az új játék kezdését (**`testNewGame`**), és a lépések végrehajtását (**`testStepGame`**).
- a tesztkörnyezet tárolja a modell egy példányát, amelyet inicializál (**`initTestCase`**), majd megsemmisít (**`cleanupTestCase`**)

Feladat: tervezés



Teszt osztály

```
#include <QtTest>
#include "tictactoemodel.h"
class TicTacToeModelTest : public QObject{
    Q_OBJECT
private:
    TicTacToeModel* _model;
private slots:
    void initTestCase();
    void cleanupTestCase();
    void testNewGame();
    void testStepGame();
    void testStepGameErrors();
};

…

QTEST_APPLESS_MAIN(TicTacToeModelTest)
#include "tictactoemodeltest.moc"
```

Teszt osztály metódusai

```
void TicTacToeModelTest::initTestCase(){
    _model = new TicTacToeModel();
}

void TicTacToeModelTest::cleanupTestCase(){
    delete _model;
}

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

tesztkörnyezet létrehozása

tesztkörnyezet megsemmisítése

tesztesetek

ellenőrizzük, hogy kezdetben minden mező üres, és a lépésszám 0

Teszt osztály metódusai

```
void TicTacToeModelTest::testStepGame(){
    _model->newGame();
    _model->stepGame(0, 0);
    QCOMPARE(_model->stepNumber(), 1);
    QCOMPARE(_model->getField(0, 0), TicTacToeModel::PlayerX);
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            QVERIFY((i == 0 && j == 0) ||
              (_model->getField(i, j) == TicTacToeModel::NoPlayer));
    _model->stepGame(0, 1);
    QCOMPARE(_model->stepNumber(), 2);
    QCOMPARE(_model->getField(0, 1), TicTacToeModel::PlayerO);
    _model->stepGame(0, 2);
    QCOMPARE(_model->stepNumber(), 3);
    QCOMPARE(_model->getField(0, 2), TicTacToeModel::PlayerX);
}
```

ellenőrizzük, hogy kezdetben minden mező üres, és a lépésszám 0

ellenőrizzük, hogy közben más mező nem változott

ellenőrizzük, hogy ezután O következik

majd ismét az X

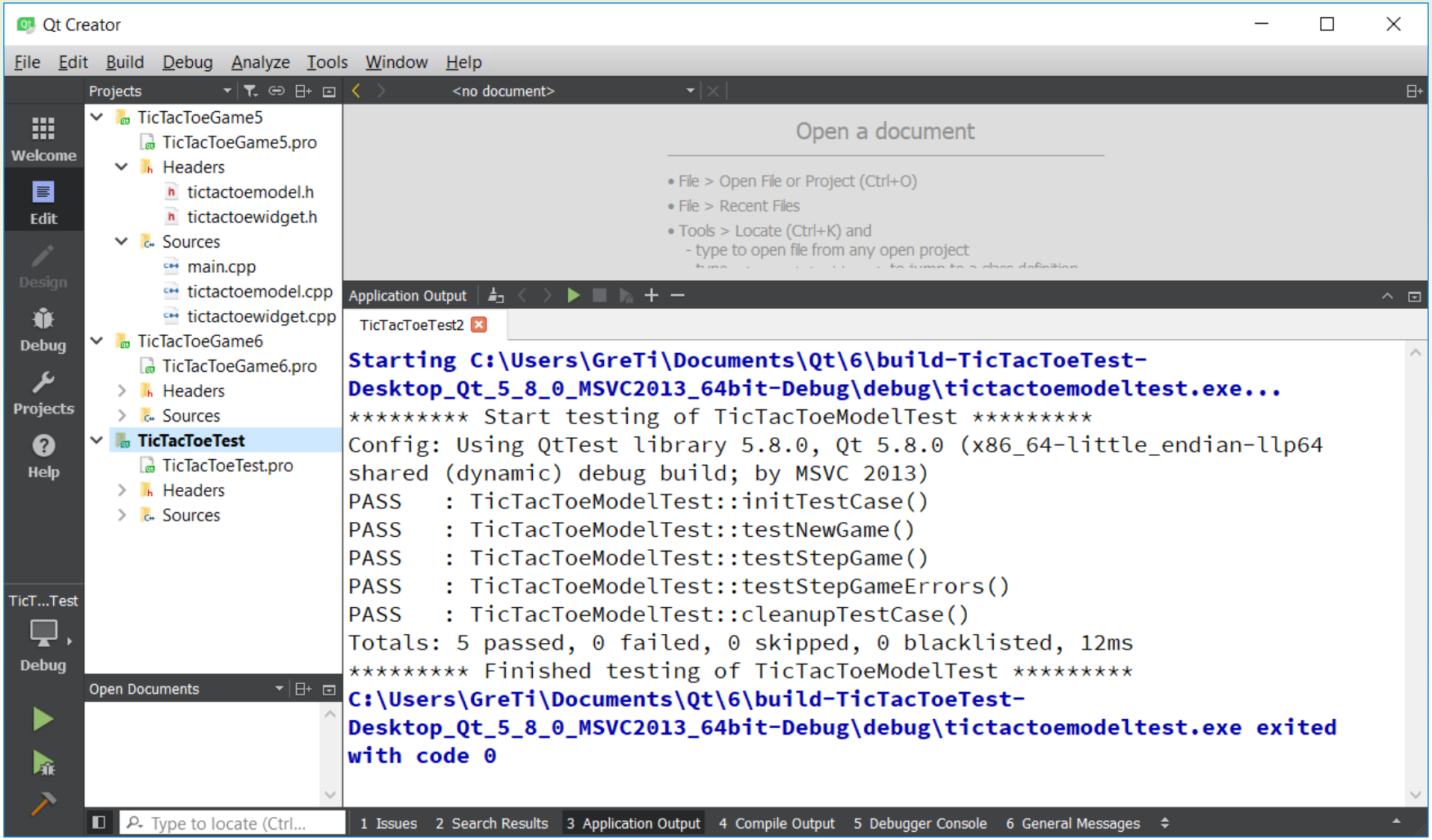
Teszt osztály metódusai

```
void TicTacToeModelTest::testStepGameErrors(){
    _model->newGame();
    _model->stepGame(-1, 0);
    _model->stepGame(0, -1);
    _model->stepGame(3, 0);
    _model->stepGame(0, 3);
    QCOMPARE(_model->stepNumber(), 0);
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            QVERIFY(_model->getField(i, j)==TicTacToeModel::NoPlayer);
    _model->stepGame(0, 0);
    _model->stepGame(0, 0);
    QCOMPARE(_model->stepNumber(), 1);
    QCOMPARE(_model->getField(0, 0), TicTacToeModel::PlayerX);
}
```

ellenőrizzük, hogy nem tudunk rossz mezőre lépni

ellenőrizzük, hogy kétszer nem tudunk lépni ugyanarra a mezőre

Teszt eredmények



Teszt eredmények