Szoftvertesztelés

Jegyzőkönyv

Egységteszt

Készítette: Veres Marcell

JEDU1N

2020.11.25

Tartalom

Bevezetés

A szoftvertesztelés a szoftverfejlesztés részét képezi. A tesztelés a programok felügyelt körülmények melletti futtatását, valamint és az eredmények kiértékelését jelenti. A szoftvertesztelés célja a szoftveren belüli hibák felfedezése, ezzel is növelve a termék minőségét és megbízhatóságát. Minél korábban találunk hibát a fejlesztés során, annál kisebb költséggel jár a javítása. A tesztelés továbbá magába foglalja a kockázatok becslését, valamint menedzselését.

Szokták emlegetni, hogy tökéletes szoftver nem létezik, csak olyan amelyikben még nem találtak hibát. Ez persze érthető, mivel ezeket a termékeket emberek fejlesztik, és az emberek hibázhatnak. A tesztelés, valamint az esetleges hibák javítása nem feltétlenül jelenti azt, hogy sikeresen feltártunk minden hibát, ezért nagy hangsúlyt kell fektetni arra, hogy ezek a folyamatok a lehető legtöbb esetet vizsgáljanak. A tesztelések során továbbá gyakori először a program azon funkcióit tesztelni, amiket a felhasználók legtöbbször fognak használni.

Alapelvek

A tesztelés alapjait a következő alapelvekben foglalhatjuk össze:

1. **A tesztelés hibák jelenlétét jelzi:** A tesztelés képes felfedni a hibákat, de azt nem, hogy nincs hiba. Ugyanakkor a szoftver minőségét és megbízhatóságát növeli.

2. **Nem lehetséges kimerítő teszt:** Minden bemeneti kombinációt nem lehet letesztelni Általában csak a magas kockázatú és magas prioritású részeket teszteljük.

3. **Korai teszt:** Érdemes a tesztelést az életciklus minél korábbi szakaszában elkezdeni, mert minél hamar találunk meg egy hibát, annál olcsóbb javítani. Ez azt is jelenti, hogy nemcsak programot, hanem dokumentumokat is lehet tesztelni.

4. **Hibák csoportosulása:** A tesztelésre csak véges időnk van, ezért a tesztelést azokra a modulokra kell koncentrálni, ahol a hibák a legvalószínűbbek, illetve azokra a bemenetekre kell tesztelnünk, amelyre valószínűleg hibás a szoftver (pl. szélsőértékek).

5. **A féregirtó paradoxon:** Ha az újratesztelés során mindig ugyanazokat a teszteseteket futtatjuk, akkor egy idő után ezek már nem találnak több hibát. Ezért a tesztjeinket néha bővíteni kell.

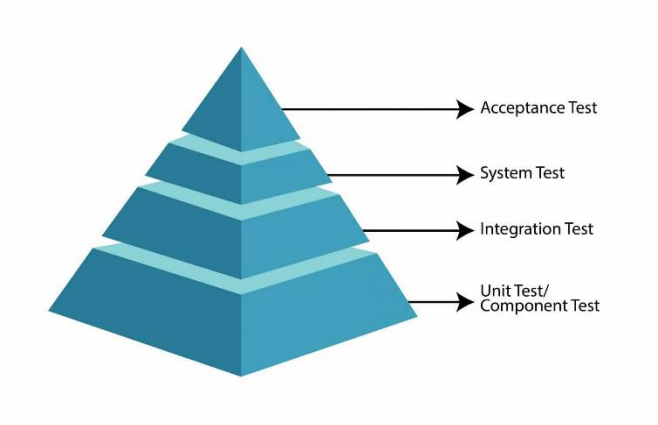
6. **A tesztelés függ a körülményektől:** Másképp tesztelünk egy atomerőműnek szánt programot és egy beadandót. Másképp tesztelünk, ha a tesztre 10 napunk vagy csak egy éjszakánk van.

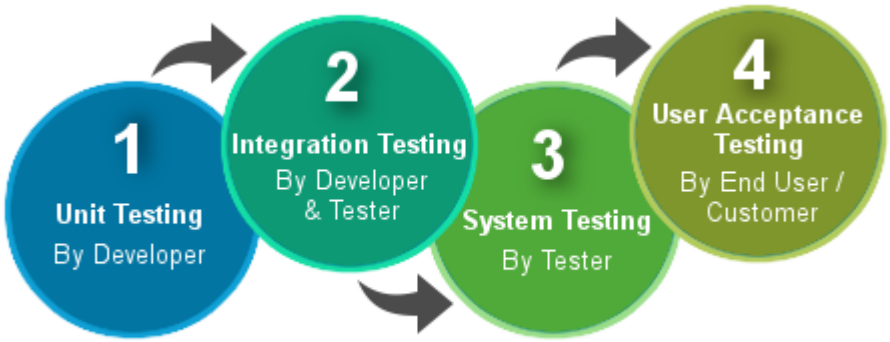
7. **A hibátlan rendszer téveszméje:** Hiába javítjuk ki a hibákat a szoftverben, azzal nem lesz elégedett a megrendelő, ha nem felel meg az igényeinek. Azaz használhatatlan szoftvert nem érdemes tesztelni.

Tesztelési szintek

A szintek a következők:

1. **komponensteszt**
    a. **unit-teszt**
    b. modulteszt
2. integrációs teszt
3. rendszerteszt
4. átvételi teszt



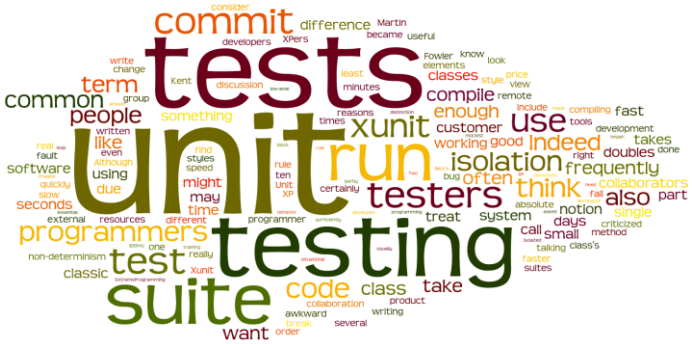


Jelen esetben csak a komponens teszttel, azon belül is az egység tesztekkel fogunk foglalkozni.

A komponensteszt a rendszer önálló részeit teszteli általában a forráskód ismeretében (fehér dobozos tesztelés).

Egységteszt

A unit-teszt, vagy más néven **egységteszt, a metódusokat teszteli**. Adott paraméterekre ismerjük a metódus visszatérési értékét (vagy mellékhatását). Az ilyen típusú tesztelés lényege az, hogy megbizonyosodjon arról, hogy a **visszatérési érték megegyezik-e az elvárttal**. Abban az esetben, ha ez teljesül, akkor a teszt sikeresnek mondható, egyébként sikertelen. Elvárás továbbá, hogy magának **a tesztnek ne legyen mellékhatása**.



Az egység tesztelésére létrehozott tesztesetek darabszáma önmagában nem minőségi kritérium: nem állíthatjuk bizonyossággal, hogy attól, mert több tesztesetünk van, nagyobb eséllyel találjuk meg az esetleges hibákat. Ennek oka, hogy a teszteseteinket gondosan meg kell tervezni. Pusztán véletlenszerű tesztadatok alapján nem biztos, hogy jobb eséllyel fedezzük fel a rejtett hibákat, azonban egy olyan tesztesettervezési módszerrel, amely például a bemenetek jellegzetességeit figyelembe véve alakítja ki a teszteseteket, nagyobb eséllyel vezet jobb teszteredményekhez. Egy fontos mérőszám a **tesztlefedettség**, amely azon **kód százalékos aránya**, amelyet az egységteszt tesztel.

Előnyei:

- 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

JUnit

A JUnit egy olyan automatizált egységtesztelő keretrendszer, amely lehetővé teszi, hogy a teszteseteket programonként definiáljuk.

A keretrendszer magába foglal egy annotációfeldolgozót, amely segítségével képes felismerni a metódusokat. Ezek **a metódusok** egy úgynevezett **tesztosztályban kapnak helyet, elkülönítve a tesztelendő programrészektől**.

Annotációk:

Annotáció	**Leírás**
@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 or @After public void method()	Metódus, amely minden teszt előtt/ (után) lefut.
@BeforeClass or AfterClass public static void method()	Metódus egyszer fut le, az osztályban lévő első teszt metódus indulása előtt
@Ignore or @Ignore("Why disabled")	Adott teszt metódus kihagyása

Mockolás

Definíció: *az objektum-orientált programozásban a mock objektumok szimulált objektumok, amik leutánozzák a valós objektum viselkedését.*

A mockolás lényege az, hogy valamilyen működést, vagy esetleg adatokat szimulálunk. Ezt általában azért tesszük, hogy csak egy adott problémára koncentráljunk. Olyan esetekben lehet ez hasznos, ha az adatok nem állnak rendelkezésre (pl.: adatbázis nem áll készen). A mockolásnak viszont sok más oka is lehet, például nem férünk még hozzá egy másik rendszerhez, nincs még kész egy kódrészlet, viszont ezekre az adatokra is fel szeretnénk készíteni a kódot. **Általában a mock objektumokat tesztesetek írásához szokták használni**.

Kód

Employee

```java
package com.meiit.szoftteszt.employee;

public class Employee
{
    private int id;
    private String name;
    private int salary;
    private String division;

    public Employee()
    {
        super();
    }

    public Employee(int id, String name, int salary, String division)
    {
        super();
        this.id = id;
        this.name = name;
        this.salary = salary;
        this.division = division;
    }


    public int getId()
    {
        return id;
    }
    public void setId(int id)
    {
        this.id = id;
    }

    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
```

```
    public int getSalary()
    {
        return salary;
    }
    public void setSalary(int salary)
    {
        this.salary = salary;
    }

    public String getDivision()
    {
        return division;
    }
    public void setDividion(String division)
    {
        this.division = division;
    }

    @Override
    public boolean equals(Object obj)
    {
        if (obj == this)
        {
            return true;
        }

        if (!(obj instanceof Employee))
        {
            return false;
        }

        Employee emp = (Employee) obj;

        return
            this.getId() == emp.getId() &&
            this.getName() == emp.getName() &&
            this.getSalary() == emp.getSalary() &&
            this.getDivision() == emp.getDivision();
    }
}
```

EmployeeDao

```
package com.meiit.szoftteszt.employee;

import java.sql.SQLException;

public interface EmployeeDao
{
    public void hireEmployee(Employee employee) throws ClassNotFoundException,
    SQLException;

    public Employee getEmployee(int id) throws ClassNotFoundException,
    SQLException;

    public void fireEmployee(int id) throws ClassNotFoundException,
    SQLException;
}
```

EmployeeDaoImpl

```
package com.meiit.szoftteszt.employee;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class EmployeeDaoImpl implements EmployeeDao
{
    Connection connection;

    public EmployeeDaoImpl(Connection connection)
    {
        this.connection = connection;
    }

    public void hireEmployee(Employee employee) throws ClassNotFoundException,
    SQLException
    {
        int id = employee.getId();
        String name = employee.getName();
        int salary = employee.getSalary();
        String division = employee.getDivision();

        PreparedStatement statement = connection.
        prepareStatement("INSERT INTO Employee VALUES (?, ?, ?, ?)");
```

```
        statement.setInt(1, id);
        statement.setString(2, name);
        statement.setInt(3, salary);
        statement.setString(4, division);

        statement.executeUpdate();
    }

    public Employee getEmployee(int id) throws ClassNotFoundException,
    SQLException
    {
        PreparedStatement statement = connection.
        prepareStatement("SELECT * FROM Employee WHERE id = ?");
        statement.setInt(1, id);

        ResultSet resultSet = statement.executeQuery();

        String name = resultSet.getString(2);
        int salary = resultSet.getInt(3);
        String division = resultSet.getString(4);

        resultSet.close();

        Employee employee = new Employee(id, name, salary, division);

        return employee;
    }

    public void fireEmployee(int id) throws ClassNotFoundException,
    SQLException
    {
        PreparedStatement statement = connection.
        prepareStatement("DELETE FROM Employee WHERE id = ?");
        statement.setInt(1, id);

        statement.executeUpdate();
    }
}
```

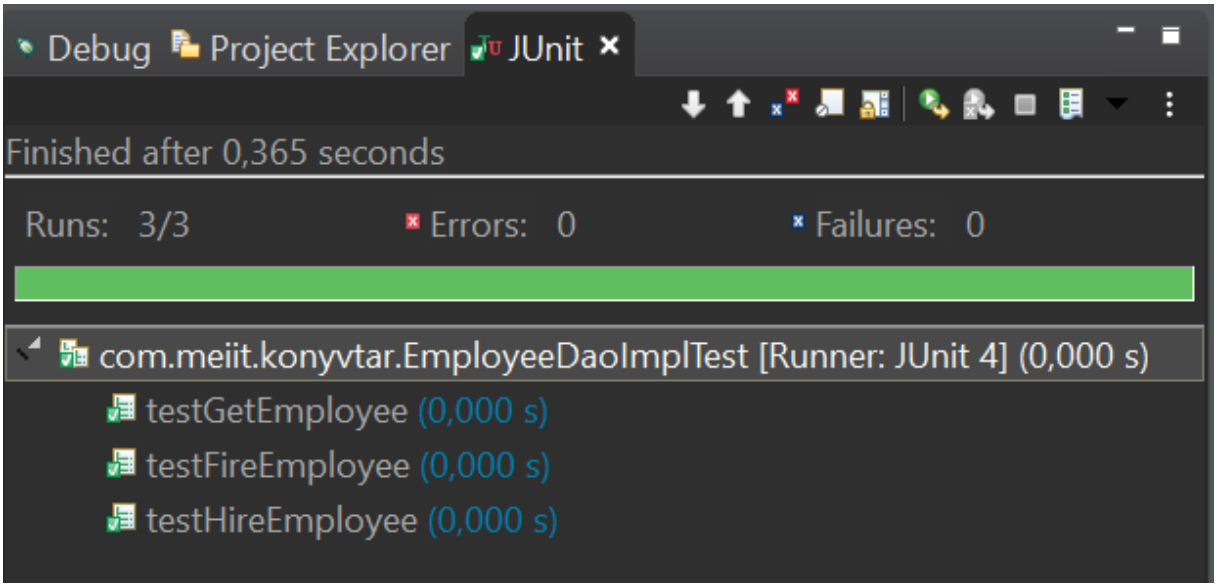
Teszt

```java
package com.meiit.konyvtar;

import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.ArgumentCaptor;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;

import com.meiit.szoftteszt.employee.Employee;
import com.meiit.szoftteszt.employee.EmployeeDao;
import com.meiit.szoftteszt.employee.EmployeeDaoImpl;

@RunWith(MockitoJUnitRunner.class)
public class EmployeeDaoImplTest
{
    @Mock
    Connection mockConnection;

    @Mock
    PreparedStatement mockPreparedStatement;

    @Mock
    ResultSet mockResultSet;

    EmployeeDao dao;

    Employee expectedEmployee;

    @Before
    public void setup() throws ClassNotFoundException, SQLException
    {
        dao = new EmployeeDaoImpl(mockConnection);
        expectedEmployee = new Employee(1, "Jakab Gipsz", 100000, "A");

        when(mockConnection.prepareStatement(anyString())).
        thenReturn(mockPreparedStatement);
        when(mockPreparedStatement.executeQuery()).thenReturn(mockResultSet);
    }
```

```
@Test
public void testHireEmployee() throws ClassNotFoundException, SQLException
{
    dao.hireEmployee(expectedEmployee);

    verify(mockPreparedStatement, times(1)).setInt(eq(1), anyInt());
    verify(mockPreparedStatement, times(1)).setInt(eq(3), anyInt());
    verify(mockPreparedStatement, times(1)).setString(eq(2), anyString());
    verify(mockPreparedStatement, times(1)).setString(eq(4), anyString());

    ArgumentCaptor<Integer> setArgument1 = ArgumentCaptor.
    forClass(Integer.class);
    ArgumentCaptor<Integer> setArgument3 = ArgumentCaptor.
    forClass(Integer.class);
    ArgumentCaptor<String> setArgument2 = ArgumentCaptor.
    forClass(String.class);
    ArgumentCaptor<String> setArgument4 = ArgumentCaptor.
    forClass(String.class);

    verify(mockPreparedStatement).setInt(eq(1), setArgument1.
    capture());
    verify(mockPreparedStatement).setInt(eq(3), setArgument3.
    capture());
    verify(mockPreparedStatement).setString(eq(2), setArgument2.
    capture());
    verify(mockPreparedStatement).setString(eq(4), setArgument4.
    capture());

    assertEquals(expectedEmployee.getId(), setArgument1.getValue().
    intValue());
    assertEquals(expectedEmployee.getSalary(), setArgument3.getValue().
    intValue());
    assertEquals(expectedEmployee.getName(), setArgument2.getValue());
    assertEquals(expectedEmployee.getDivision(), setArgument4.getValue());
}
```

```java
    @Test
    public void testGetEmployee() throws ClassNotFoundException, SQLException
    {
        when(mockResultSet.getString(2)).
        thenReturn(expectedEmployee.getName());
        when(mockResultSet.getInt(3)).
        thenReturn(expectedEmployee.getSalary());
        when(mockResultSet.getString(4)).
        thenReturn(expectedEmployee.getDivision());

        Employee actualEmployee = dao.getEmployee(1);

        assertTrue(actualEmployee.equals(expectedEmployee));
    }

    @Test
    public void testFireEmployee() throws ClassNotFoundException, SQLException
    {
        dao.fireEmployee(1);

        verify(mockPreparedStatement, times(1)).executeUpdate();
    }

}
```

Eredmények



Források

- https://hu.wikipedia.org/wiki/Szoftvertesztel%C3%A9s
- https://regi.tankonyvtar.hu/hu/tartalom/tamop425/0046_szoftverteszteles/index.html
- https://gyires.inf.unideb.hu/GyBITT/21/ch03s02.html
- https://www.professionalqa.com/levels-of-testing
- https://medium.com/swlh/software-testing-and-methodologies-1fc519c98fdf
- https://www.toolsqa.com/software-testing/istqb/component-testing/
- https://github.com/junit-team/junit4
- https://sungsoo.github.io/2014/05/23/unit-test.html
- https://swap.web.elte.hu/2018191_pt2e/ea08.pdf
- https://infoframe.wordpress.com/2010/11/23/mockolas/
- https://www.north-47.com/knowledge-base/unit-testing-with-mockito/