JUnit – Tutorial

Tento tutoriál vysvětluje, jednotkové testování s JUnit 4.x.

Úvod:

- *Testování softwaru:*

  Proč testovat? Vývoj SW se stále zrychluje, požadavky na kvalitu jsou minimálně stejné, ne-li vyšší než tomu bylo před lety. Testování SW tak nabývá stále na větším významu, protože čím dříve se chyby odhalí, tím nižší jsou náklady na jejich odstranění. Testování SW se stává nedílnou součástí životního cyklu vývoje software (SDLC – Software Development Life Cycle).

  Co je cílem testování? Cílem testování obvykle bývá ověřit, že SW dělá přesně to, co je uvedeno ve specifikaci a dále jak je schopen se vyrovnat s nestandardními stavy, jak reaguje na chybu uživatele nebo chybu v datech, selhání jiné SW nebo HW komponenty, jak se vypořádá se zátěží a nedostatkem systémových zdrojů, zda se dokáže zotavit po havárii, zda je odolný vůči útokům, jak funguje na různých HW a SW konfiguracích, atd.

  Je třeba mít neustále na paměti, že to, že se během testování neobjevila žádná chyba a všechny testy byly úspěšné, neznamená, že SW žádné chyby neobsahuje. S velkou pravděpodobností nějaké obsahuje, ale jen se na ně nepřišlo. To je dáno tím, že SW stejně jako testy dělají lidé a lidé jak známo chybují. Odhalené množství chyb je tak značně závislé na kvalitě vývoje a kvalitě testování. V praxi tak může dojít k tomu, že:

  - kvalita SW i testů je vysoká, lze předpokládat, že bude odhaleno jen malé množství chyb, ale pár jich produkt přesto může obsahovat;

  - kvalita SW je vysoká, naproti tomu kvalita testů je nízká, lze předpokládat, že bude odhaleno méně chyb než v předchozím případě;

  - kvalita SW i testů je nízká, lze předpokládat, že bude odhaleno jen malé množství chyb, přestože produkt jich bude obsahovat velké množství;

  - kvalita SW je nízká, naproti tomu kvalita testů je vysoká, lze předpokládat, že bude odhaleno velké množství chyb.

  To, že je SW nebo testování kvalitní, je často založeno jen na naší víře a přesvědčení. Můžeme se tak mylně domnívat, že nízký počet odhalených chyb je dán vysokou kvalitou SW a testování. Opak však může být pravdou, SW obsahuje plno chyb, ale z důvodu nízké kvality testování jich bylo odhaleno jen pár.

- *Unit testing:*

  Slovní spojení unit testing se při programování používá jako pojem a v českém jazyce pravděpodobně dosud nemá ustálený překlad. Pojem by se dal přeložit jako testování jednotek, testování aplikačních jednotek, doslovně jako jednotkové testování. Unit testing je činností související s vývojem aplikačních programů. Koncoví uživatelé programů se s ním nesetkávají.

  Pod pojem unit testing se zahrnují nástroje, metodika a činnost, jejímž cílem je ověřování správné funkčnosti dílčích částí neboli jednotek zdrojového kódu.

  Technicky řečeno, za jednotku by se měla považovat samostatně testovatelná část aplikačního programu. Z pohledu procedurálního programování může být jednotkou program, funkce, procedura, atd. Z pohledu objektově orientovaného programování je jednotkou obvykle třída. V rámci ní se ale obvykle individuálně testují její metody.

  V ideálním případě by měl být každý testovaný případ nezávislý na ostatních. Při testování se snažíme testovanou část izolovat od ostatních částí programu. Za tím účelem se někdy vytvářejí pomocné objekty, které simulují předpokládaný kontext, ve kterém testovaná část pracuje.

- *Co je JUnit:*

  „JUnit'' je asi nejznámější framework pro testování a je nejen pro Javu, ale taky například pro C, C#, Delphi, Free Pascal, Python, PHP nebo JavaScript. Testuje jednotlivé funkce na správnost výsledků různých typů vstupních proměn, nejedná se však o ``black-box testing``, kde máme zkompilovaný celek a nevídime dovnitř.

  Patří do rodiny tzv. xUnit testovacích frameworků. Na jeho vývoji se podíleli vývojáři Kate Beck a Erich Gamma. Rychle se stal základem Java testů a v současnosti se může těšit už ze čtvrté verze.Pro testování je nutné vytvořit jen minimální strukturu kódu k volání testované funkce a k porovnání očekávaných výstupů testované funkce. Junit test umožňuje spojit několik testů do jednoho celku, který umožňuje spustit všechny testy spojené s projektem nebo jen s částí projektu, aniž by bylo nutné spouštět celou vyvíjenou aplikaci.

  Pro rozpoznání JUnit testů v kódu se používají tzv. *@-notace*. Například jako @-notace *@Test, @Before, @After, @BeforeClass* a další. Díky těmto notacím je možné vykonávat různé operace (inicializace polí, přihlášení, resetování proměnných, atd.) před začátkem celého testu, před a po každém jednotlivém testu. Či vytvářet různé pomocné metody pro testování aplikace.

- *JUnit testy:*

  Pro psaní testů v Javě je připraven právě JUnit framework. Umožňuje nám vytvářet javovské třídy, které mají ve jméně klíčové slovo Test. Tetovací metody takové třídy jsou vždy typu void Vždy začínají „*test*“

  Testy využívají speciálního slovíčka „*assert“.* Tyto tzv. “assert” metody jako například *assertEquals(boolean, boolean)* předpokládají na prvním vstupním parametru hodnotu, kterou očekávám, druhý parametr je pak hodnota, kterou metoda (třída) skutečně vrací.

  Pokud jsou si hodnoty rovny (odpovídají myšlence testu), pak JUnit vrací kladnou odpověď. Pokud jsou hodnoty různé, test se dostane do chybového stavu – fail.

  - Požíváme typicky: *assertNull, assertTrue, assertEqals, assertSame(),* a další.

  Rozdíl mezi assertSame(...) a assertEquals(...) spočívá v tom, že assertSame(...) používá pro porovnání == (porovnání referencí), kdežto assertEquals(...) používá metodu equals(). Nemáme-li metodu equals() v našem objektu implementovanou, pak je každá instance rovna pouze sama sobě.

- *Typy a rady:*
  - Testovací třídy je vhodné psát před implementací samotné třídy testované. Při psaní testů se mnohdy odhalí chyby, na které se nemyslelo při návrhu.

  - Název testovací třídy bývá zvykem vytvořit spojením názvu třídy testované a slova Test.

  - K inicializaci a vyčistění prostředí JUnit poskytuje metody *setUp()* a *tearDown()*, které jsou volány před každým a po každém vykonání testu. Testy ve třídě jsou tak od sebe izolovány. *setUp()* je vlastně metoda, která se provede před každým jedním testem. V *setUp()* si připravíme data, nad kterými budeme provádět testy a v *tearDown*() budeme po době uklízet.

  - Existují i metody, které se vyvolají právě jednou před testováním a právě jednou po testování - přišlo s nimi JUnit 4.x a takové metody mají @-notace @*BeforeClass a @AfterClass*.

  - JUnit umožňuje také testovat výjimky, tzv. exceptions. A to tím způsobem, že napíšete anotaci @Test(expected = a typ dané výjimky) příklad: „@Test(expected = ArithmeticException.class)“

- JUnit podporuje také časové omezení testů, které se zejména hodí k testování různých algoritmů dělajících stejnou činnost. Pokud dojde během testování k překročení časového limitu (v milisekundách) skončí test chybou. Opět se využívá @-notace: @Test(timeout = „požadovaný čas v ms“)

- Je vhodné testovat minimálně všechny public metody a třídy

- Většinou se netestují primitivní gettery a settery

Základy tvorby JUnit testů v NetBeans:

Úvod:

Následující text obsahuje základní seznámení s použitím testovacího frameworku JUnit ve vývojovém prostředí Netbeans.

Celý tento tutoriál, jak je uvedeno na začátku, se vztahuje k JUnit frameworku verze 4. Ta na rozdíl od předcházející verze 3 obsahuje syntaktické změny činící ji jednodušeji použitelnější. Stále je však zpětně kompatibilní a proto i v ní je možno spouštět i psát testy odpovídající syntaxi předchozí verze. Některá z nových vylepšení jsou obsažena v následujícím seznamu:

- Určení testovací jednoty pomocí anotace @Test namísto striktního dodržování jmenné konvence (ve verzi 3 musely všechny názvy testových metod končit slovem Test).
- Určení metod setUp a tearDown pomocí anotace @Before a @After.
- Určení metod setUp a tearDown, které se aplikují na celou testovací třídu. Metody označené @BeforeClass jsou spuštěny pouze jednou, a to před tím než je spuštěna jakákoliv testovací metoda dané třídy. Metody označené pomocí @AfterClass jsou taktéž spuštěny pouze jednou, a to poté co jsou dokončeny všechny testovací metody.
- identifikace testování předpokládaných vyjímek.
- Označení testů, které by měly být vynechány pomocí anotoce @Ignore.
- Určení maximální doby běhu testu.

Kódy dvou ukázkových tříd:

Pro názorné předvedení zmíněných základních dovedností testovací frameworku JUnit budu v tomto tutoriálu použit kód následujících dvou tříd: Utils.java a Vectors.java

Utils.java

```
package sample;

import java.lang.reflect.Method;
import java.math.BigInteger;

public class Utils {

    private Utils() { }

    public static String concatWords(String... words) {
        StringBuilder buf = new StringBuilder();
        for (String word : words) {
            buf.append(word);
        }
        return buf.toString();
    }

    public static String computeFactorial(int number)
                                            throws IllegalArgumentException {
        if (number < 1) {
            throw new IllegalArgumentException("zero or negative parameter (" + number + ')');
        }

        BigInteger factorial = new BigInteger("1");
        for (int i = 2; i <= number; i++) {
            factorial = factorial.multiply(new BigInteger(String.valueOf(i)));
        }
        return factorial.toString();
    }

    public static String normalizeWord(String word) {
        try {
            int i;
            Class<?> normalizerClass = Class.forName("java.text.Normalizer");
            Class<?> normalizerFormClass = null;
            Class<?>[] nestedClasses = normalizerClass.getDeclaredClasses();
            for (i = 0; i < nestedClasses.length; i++) {
                Class<?> nestedClass = nestedClasses[i];
                if (nestedClass.getName().equals("java.text.Normalizer$Form")) {
                    normalizerFormClass = nestedClass;
                }
            }
            assert normalizerFormClass.isEnum();
            Method methodNormalize = normalizerClass.getDeclaredMethod(
                                                            "normalize",
                                                            CharSequence.class,
                                                            normalizerFormClass);
            Object nfcNormalization = null;
            Object[] constants = normalizerFormClass.getEnumConstants();
            for (i = 0; i < constants.length; i++) {
                Object constant = constants[i];
                if (constant.toString().equals("NFC")) {
                    nfcNormalization = constant;
                }
            }
            return (String) methodNormalize.invoke(null, word, nfcNormalization);
        } catch (Exception ex) {
            return null;
        }
    }

}
```

Vectors.java

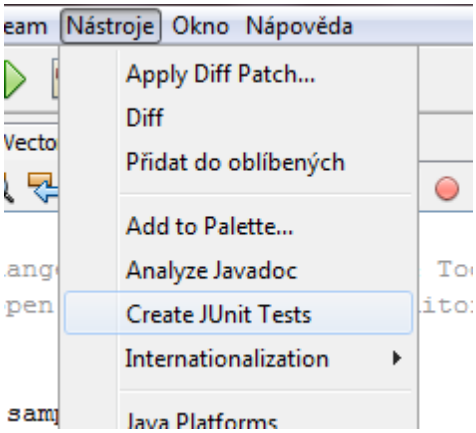
```
package sample;

public final class Vectors {

    private Vectors() {}
    public static boolean equal(int[] a, int[] b) {
        if ((a == null) || (b == null)) {
            throw new IllegalArgumentException("null argument");
        }

        if (a.length != b.length) {
            return false;
        }

        for (int i = 0; i < a.length; i++) {
            if (a[i] != b[i]) {
                return false;
            }
        }

        return true;
    }

    public static int scalarMultiplication(int[] a, int[] b) {
        if ((a == null) || (b == null)) {
            throw new IllegalArgumentException("null argument");
        }

        if (a.length != b.length) {
            throw new IllegalArgumentException(
                    "different tuple of the vectors ("
                    + a.length + ", " + b.length + ')');
        }

        int sum = 0;
        for (int i = 0; i < a.length; i++) {
            sum += a[i] * b[i];
        }
        return sum;
    }

}
```

Framework JUnit při testování vychází z tzv. **testovacích** tříd, které popisují průběh jednotlivých testů na třídách **testovaných**. Proto je jejich správné vytvoření klíčovou dovedností prezentovanou tímto tutoriálem.

- *Vytvoření testovací třídy pro třídu Vectors:*

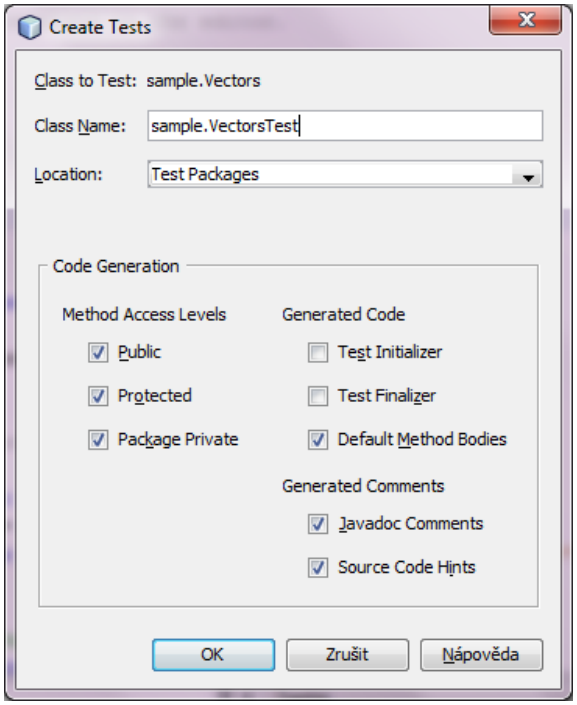
  Nejprve si následujícím způsobem vytvoříme kostru testovací třídy pomocí prostředí NetBeans:

1. Klikneme pravým tlačítkem na třídu Vectors.java a z hlavní nábídky *Nástroje* vybereme možnost *Create JUnit Tests*.



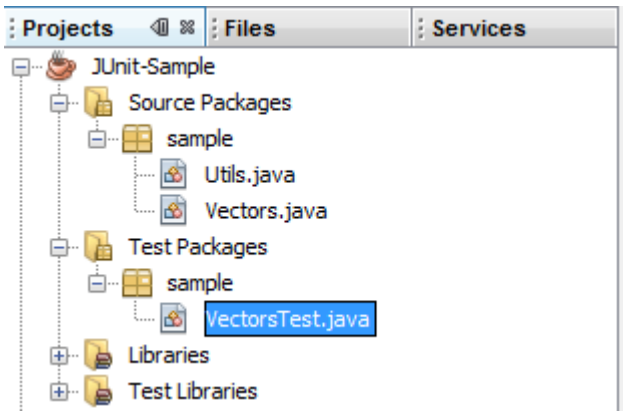
obr. 1: nabídka Nástroje

2. V nově otevřeném dialogu zvolíme možnost JUnit 4.x.
3. V dalším dialogu můžeme upravit jméno testovací třídy (novinka ve verzi 4), ačkoliv se doporučuje ponechat automaticky vygenerované z důvodů možnosti navigace mezi testovanou a testovací třídou.
4. Zrušíme volbu *Test Initializer* a *Test Finalizer* a potvrdíme klepnutím na tlačítko OK.



obr. 2: nabídka Create Tests

Po potvrzení volby vytvoří IDE soubor VectorsTest.java s testovací třídou v balíčku pojmenovaném stejně jako balíček testované třídy (tedy sample) zařazeném pod položkou Test Packages



obr. 3: umístění testovací třídy ve struktuře projektu

Třída VectorsTest nyní obsahuje zmíněnou kostru testů. Jednotlivé testovací metody (uvozené anotací @Test) byly pojmenovány podle testovaných metod třídy Vectors. Toto pojmenovaní však ve verzi 4 není díky nově zavedeným anotacím nutné nadále dodržovat.

Pokud si nepřejete, aby byla vygenerována i těla testovacích metod, je možno tuto volbu příště v dialogu na obr.2 také odškrtnout.

Kromě metod se zmíněnou anotací @Test se v souboru také nacházejí 2 další metody s anotacemi @BeforeClass a @AfterClass pojmenované setUpClass a tearDownClass.

Obecně tyto metody slouží k vytvoření počátečních podmínek pro běh testu a k jejich následnému „uklizení“. Jejich anotace zajišťují, že budou spuštěny každá pouze jednou a to buď před provedením jakékoliv jiné metody z dané testovací třídy (@BeforeClass) nebo až zcela nakonec (@AfterClass).

Například namísto vytváření databázového spojení v inicializátoru testu (anotace @Before, viz „Vytvoření testovací třídy pro Utils“) a vytváření nového spojení před každou testovací metodou bychom mohli použít inicializér @BeforeClass k otevření onoho spojení před rozběhnutím samotných testů.

Metoda s anotací @AfterClass by nám pak spojení měla naopak uzavřít. V našem konkrétním příkladu je však nebudeme potřebovat a tudíž je můžete ze souboru odstranit.

- *Psaní testovacích medod pro Vectors.java:*

  Abychom demonstrovali možnost vlastního pojmenování testovacích tříd, upravte třídu VectorsTest do následující podoby (přidána je i vlastní implementace jednotlivých testovacích metod):

```
package sample;

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class VectorsTest {

    public VectorsTest() {
    }

    @Test
    public void equalsCheck() {
        System.out.println("* VectorsJUnit4Test: equalsCheck()");
        assertTrue(Vectors.equal(new int[] {}, new int[] {}));
        assertTrue(Vectors.equal(new int[] {0}, new int[] {0}));
        assertTrue(Vectors.equal(new int[] {0, 0}, new int[] {0, 0}));
        assertTrue(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0, 0}));
        assertTrue(Vectors.equal(new int[] {5, 6, 7}, new int[] {5, 6, 7}));

        assertFalse(Vectors.equal(new int[] {}, new int[] {0}));
        assertFalse(Vectors.equal(new int[] {0}, new int[] {0, 0}));
        assertFalse(Vectors.equal(new int[] {0, 0}, new int[] {0, 0, 0}));
        assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0}));
        assertFalse(Vectors.equal(new int[] {0, 0}, new int[] {0}));
        assertFalse(Vectors.equal(new int[] {0}, new int[] {}));

        assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0, 1}));
        assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 1, 0}));
        assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {1, 0, 0}));
        assertFalse(Vectors.equal(new int[] {0, 0, 1}, new int[] {0, 0, 3}));
    }

    @Test
    public void ScalarMultiplicationCheck() {
        System.out.println("* VectorsJUnit4Test: ScalarMultiplicationCheck()");
        assertEquals(  0, Vectors.scalarMultiplication(new int[] { 0, 0}, new int[] { 0, 0}));
        assertEquals( 39, Vectors.scalarMultiplication(new int[] { 3, 4}, new int[] { 5, 6}));
        assertEquals(-39, Vectors.scalarMultiplication(new int[] {-3, 4}, new int[] { 5,-6}));
        assertEquals(  0, Vectors.scalarMultiplication(new int[] { 5, 9}, new int[] {-9, 5}));
        assertEquals(100, Vectors.scalarMultiplication(new int[] { 6, 8}, new int[] { 6, 8}));
    }
}
```

Jistě jste si všimli, že jsou v novém kódu velmi často využívány metody, jejichž pojmenování začíná na slovo „assert“.

Úspěšný průchod testu je přímo závislý právě na jejich výsledku. U metody assertEquals, která přejímá dva vstupní parametry, se tyto parametry musejí přímo rovnat. Obvykle na její vstup tedy přivádíme výsledek daný testovanou funkcí a jeho předpokládanou hodnotu.

Metody assertTrue a assertFalse pak pro úspěšné dokončení testu musejí jako argument ve všech případech dostat hodnotu true resp. false. Používáme je tedy zejména k testování metod s návratovou hodnotou boolean.

- *Vytvoření testovací třídy pro třídu Utils:*

  Nejprve opět vytvoříme kostru testovací třídy – postupujeme tedy obdobně jako v případě třídy Vectors. Vytvořený soubor UtilsTest upravíme na následující tvar:

```
package sample;

import org.junit.Ignore;
import org.junit.After;
import org.junit.Before;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class UtilsTest {

    public UtilsTest() {
    }

    @BeforeClass
    public static void setUpClass() throws Exception {
        System.out.println("* UtilsJUnit4Test: @BeforeClass method");
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
        System.out.println("* UtilsJUnit4Test: @AfterClass method");
    }

    @Before
    public void setUp() {
        System.out.println("* UtilsJUnit4Test: @Before method");
    }

    @After
    public void tearDown() {
        System.out.println("* UtilsJUnit4Test: @After method");
    }

    @Test
    public void helloWorldCheck() {
        System.out.println("* UtilsJUnit4Test: test method 1 - helloWorldCheck()");
        assertEquals("Hello, world!", Utils.concatWords("Hello", ", ", "world", "!"));
    }

    @Test(timeout=1000)
    public void testWithTimeout() {
        System.out.println("* UtilsJUnit4Test: test method 2 - testWithTimeout()");
        final int factorialOf = 1 + (int) (30000 * Math.random());
        System.out.println("computing " + factorialOf + '!');
        System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf));
    }

    @Test (expected=IllegalArgumentException.class)
    public void checkExpectedException() {
        System.out.println("* UtilsJUnit4Test: test method 3 - checkExpectedException()");
        final int factorialOf = -5;
        System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf));
    }

    @Ignore
    @Test
    public void temporarilyDisabledTest() throws Exception {
        System.out.println("* UtilsJUnit4Test: test method 4 - checkExpectedException()");
        assertEquals("Malm\u00f6", Utils.normalizeWord("Malmo\u0308"));
    }
}
```

Nyní si postupně popíši jednotlivé změny. S anotacemi @BeforeClass a @AfterClass jsme se již setkali v minulém příkladu. Zde uvedenou novinkou jsou velmi podobně znějící anotace @Before a @After. Takto označené metody se budou spouštět před během kteréhokoliv testovacího případu (metody s anotací @Test) resp. po jeho ukončení a slouží např. k inicializaci vnitřních proměnných a k jejich opětovnému vynulování na konci testu. V našem příkladu jsou však uvedeny pouze pro ukázku a mohou být s klidem vymazány.

Rozebrání jednotlivých testů:

- *helloWorldCheck()* – v této metodě využíváme k testování již zmíněného *assertEquals*, za zmínku stojí snad jen neopomenutí vložení příkazu *println* pro snazší identifikaci testu ve výstupu testovací konzole.

- *testWithTimeout()* – zde můžeme předpokládat, že je metoda implementována dobře a zajímá nás pouze její časová náročnost. Zadáním hodnoty v milisekundách určujeme, po jaké době se běh testovacího vlákna přeruší a metoda vyvolá TimeoutException značící neúspěch testu.

- *checkExpectedException()* – tato metoda demonstruje testování předpokládané vyjímky. Uvedením názvu její třídy specifikujeme, že test je úspěšný pouze tehdy, pokud daná metoda vyvolá tuto vyjímku.

- *temporarilyDisabledTest()* – zde je uveden způsob (použití anotace @Ignore), kterým lze dočasně vyřadit konkrétní testovací metodu z dané testovací třídy.

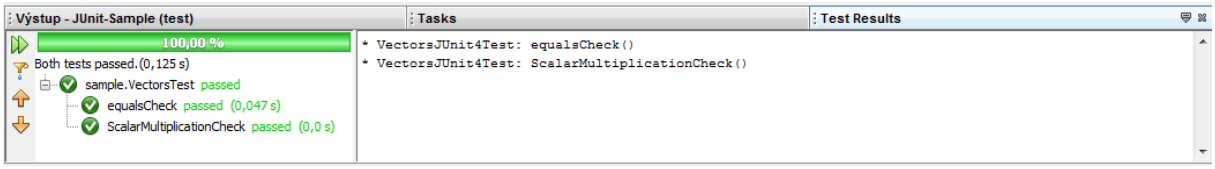
- *Spouštění testů:*

Vytvořili jsme jednotlivé testovací třídy. Samotný test pak můžeme spustit různými způsoby. Mezi ty základní patří:

- v nabídce Run kliknutím na volbu Test Project („název projektu“) spustit všechny vytvořené testy najednou;

- v nabídce Run kliknutím na volbu Test File spustit test konkrétní třídy nebo aktivní testovací třídu;

- v záložce Projects pravým/levým kliknutím na zvolený testovací soubor a vybráním volby Run File/Test File.



obr. 4: výsledky testu

V nově otevřených výsledcích testu (okono TestResults, viz obr. 4) můžeme vidět v levé části obrazovky zařazení jednotlivých testů pod danou testovací třídu, úspěšnost testu a tlačítka pro jeho opakování, filtraci úspěšných testů a přechod k předchozí či následující chybě. V pravé části se pak nachází výpis testové konzole.

- ## *Vytváření TestSuites (balíků testů):*

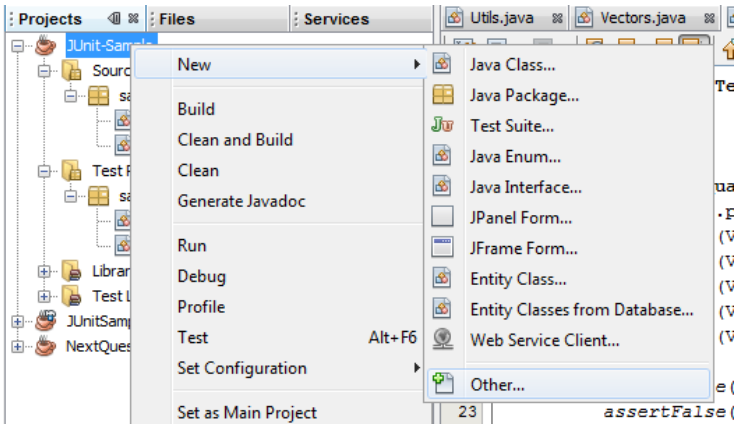
Při vytváření testů pro projekt obvykle skončíme se spoustou testovacích tříd. Ačkoliv je můžeme spouštět samostatně nebo naráz všechny pro celý projekt, v mnoha případech by se hodilo spouštět v jistém pořadí pouze jejich danou podmnožinu.

Toho lze docílit vytvořením jednoho nebo více tzv. TestSuites (jakýchsi balíků testů). Jeden takový balík nám pak může zastupovat otestování pouze specifického aspektu našeho kódu nebo vytvořit specifické podmínky testování.

TestSuite je sám o sobě obyčejně třídou s metodami vyvolávajícími příslušné testovací případy, kterými jsou testovací třídy, testovací metody v testovacích třídách a další balíky testů (TestSuites). TestSuites můžete tvořit ručně nebo opět využít k jejich vygenerování služeb IDE. Defaultně IDE vytvoří kód volající všechny testovací třídy ve stejném balíčku v jakém se TestSuite nachází. To však můžeme po jeho vytvoření jednoduše upravit přepsáním částí jeho obsahu jenž nám nevyhovují.

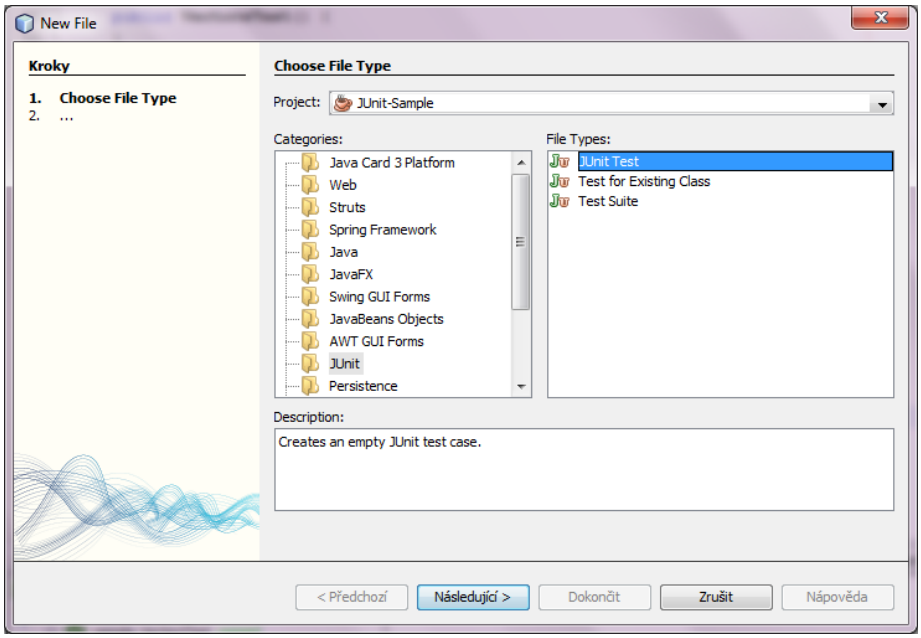
Postupujte následovně:

1. Pravým tlačítkem klikneme na uzel projektu v záložce Projects a vybereme New > Other…



obr. 5: vytvoření TestSuite pomocí IDE

2. Vybereme kategorii JUnit a Test Suite.



obr. 6: vytvoření TestSuite pomocí IDE

3. Vybereme balíček, abychom TestSuite vytovřeili ve složce sample balíčků testovacích tříd.

4. Odznačíme volby TestInitializer a TestFinalizer a klikneme na tlačítko Finish.

Vygenerovaný kód by měl být podobný následující ukázce:

```
@RunWith(Suite.class)
@Suite.SuiteClasses(value={UtilsTest.class, VectorsTest.class})
public class JUnit4TestSuite {
}
```

Pro nastavení spouštění jiných testovacích tříd tedy stačí připsat jejich názvy.

TestSuites spouštíme obdobným způsobem jako samotné testovací třídy.

Ostatní:

- *Black-box testing:*

  Black box testování, známé také jako opaque box, closed box, behavioral nebo funkční je realizováno bez znalosti vnitřní datové a programové struktury.

  To znamená, že tester nemá k dispozici žádnou dokumentaci, binární ani zdrojové kódy. Tento způsob testování vyžaduje testovací scénáře, které jsou buď poskytnuty testerovi, nebo si je tester u některých typů testů sám vytváří. Vzhledem k tomu, že jsou obvykle definovány typy a rozsahy hodnot přípustných a nepřípustných pro danou aplikaci a tester ví, jaký zadal vstup, tak ví i jaký výstup nebo chování může od aplikace očekávat. Black box testy mohou stejně jako white box testy probíhat ručně nebo automatizovaně za použití nejrůznějších nástrojů. I v tomto případě se s oblibou využívá obou přístupů. Black box se jeví jako ideální tam, kde jsou přesně definované vstupy a rozsahy možných hodnot.

  Využití

  Black box testu lze využít např. na zjištění problémů typu odepření služby (DoS) nebo aktuálních zranitelností již běžícího systému nebo aplikace. V případě použití tohoto způsobu testování je třeba vždy počítat s určitým rizikem pádu daného systému. Black box testem je také možné demonstrovat chybu, která byla objevena během white box testu. K provedení testů stačí mít k dispozici jen daný program nebo znát jeho umístění, v takovém případě může být proveden test i vzdáleně po síti.

  Výhody

  - Snadnost – test může být proveden bez znalosti programovacích jazyků, operačních systémů.
  - Rychlost – lze rychle v krátkém období otestovat i rozsáhlé systémy.
  - Transparentnost – test je pro zákazníka srozumitelný – chápe, co se bude a jak testovat, může a často to bývá i on, kdo testovací scénáře vytváří a testování pak sám provádí.
  - Testovací scénáře mohou být napsány v okamžiku, kdy je kompletní specifikace (některé prameny uvádí, že je možné je psát už v průběhu vzniku specifikace.)
  - Testování není založeno na aktuální implementaci. I když se změní programovací jazyk, operační systém a HW, testování bude probíhat pořád stejně. Testovací scénář není nutno měnit.
  - Testerovi není nutné zpřístupňovat zdrojový kód.

Nevýhody

- Nižší kvalita kódu – to, že se na výstupu objeví očekávaná hodnota, neznamená, že aplikace je správně napsaná. Kód může být značně neefektivní.
- Nežádoucí chování aplikace – kromě požadované funkcionality může produkt provádět i jiné akce, které nejsou ve specifikaci a jejichž výstup se na standardním výstupu neobjeví a test je proto neodhalí.

Zdroje:

Clever and Smart [online]. 2009 [cit. 2011-11-09]. Clever and Smart. Dostupné z WWW: <http://www.cleverandsmart.cz/black-box-test/>.

Cgit.spermik.info [online]. 2010 [cit. 2011-11-14]. Cgit.spermik.info. Dostupné z WWW: <http://cgit.spermik.info/cgit/progtest/plain/dokumentace/referaty-clenu/referat-javaframework/referat/report.tex?id=75d8aad27bf9c4160cd27ef2efa6128ccc219863>.

NetBeans.org [online]. 2011 [cit. 2011-11-14]. NetBeans. Dostupné z WWW: <http://netbeans.org/kb/docs/java/junit-intro.html#Exercise_10>.

JUnit. In Wikipedia : the free encyclopedia [online]. St. Petersburg (Florida) : Wikipedia Foundation, , last modified on 16. 10. 2011 [cit. 2011-11-14]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/JUnit>.