Testarea unitară

1. Scopul lucrării

Obiectivele de învățare ale acestei sesiuni de laborator sunt:
- Înțelegerea și utilizarea conceptului de testare unitară
- Scrierea și aplicarea metodelor de test

2. Ce este testarea unitară?

Termenul de 'testare unitară' se referă la testarea individuală a unor unități separate dintr-un sistem software. În sistemele orientate pe obiecte, aceste 'unități' sunt de regulă clase și metode.

Elementele de testare unitară, pun la dispoziție unelte pentru a înregistra și repeta teste, pentru ca testele unitare să poată fi repetate ușor mai târziu (de regulă când se schimbă o parte din sistem), astfel încât dezvoltatorul să fie convins că noile modificări nu au stricat vechea funcționalitate. Acest lucru e cunoscut sub numele de *testare regresivă*.

Conceptele testării unitare și testării regresive sunt destul de vechi, dar popularitatea lor a crescut, după publicarea *metodologiei de programare extreme* și după apariția unei unelte de testare unitară pentru Java: *JUnit*.

JUnit este un cadru de testare regresivă scris de Erich Gamma și Kent Beck. Puteți găsi o mulțime de informații despre el la adresa: *http://www.junit.org*

3. Caracteristici JUnit

Framework-ul de testare JUnit include următoarele caracteristici principale:
- Aserțiuni (*Asserts*)
- Stări fixe (*Fixtures*)
- Executori de teste (*Test runners*)
- Suite de teste (*Test suites*)

3.1. Aserțiuni (*Asserts*)

Aserțiunile sunt metode de bază din clasa **Assert** utilizate pentru testare. Ordinea parametrilor dintr-o metodă de tip *assert* este: parametru așteptat, parametru actual. Opțional, primul argument al unei metode *assert* poate fi un mesaj String ce se afișează în caz de eșec. În mod normal un test oarecare eșuează dacă o metodă *assert* va eșua. Cele mai utilizate metode ale clasei Assert sunt:

void assertEquals(primitive expected, primitive actual) – verifică dacă două valori primitive sunt egale
void assertEquals(Object expected, Object actual) – verifică dacă două obiecte sunt egale (prin apelul metodei equals() cu acele obiecte)
void assertSame(Object expected, Object actual) – verifică dacă două obiecte referă aceeași adresă de memorie
void assertNotSame(Object expected, Object actual) – verifică dacă două obiecte nu referă aceeași adresă de memorie
void assertFalse(boolean condition) – Verifică dacă condiția este falsă
void assertTrue(boolean condition) – Verifică dacă condiția este adevărată
void assertNotNull(Object object) – Verifică dacă obiectul nu este null
void assertNull(Object object) – Verifică dacă obiectul este null

Petru a putea utiliza metodele de mai sus este necesar să facem import la metodele statice din clasa Assert:

```
import static org.junit.Assert.*;
```

3.2. Stări fixe (*Fixtures*)

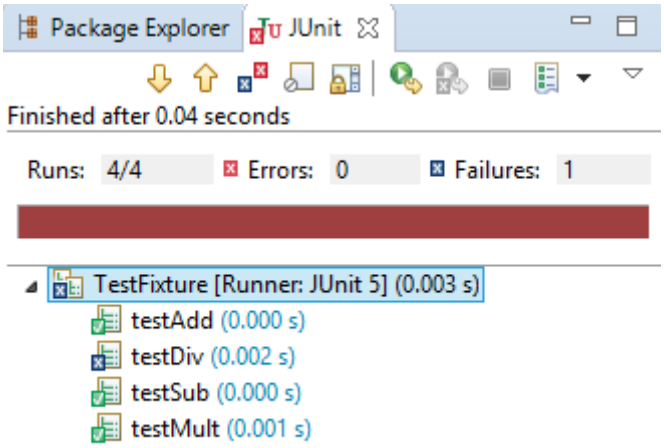
O stare fixă de test este un set de obiecte pregătit ca un punct de start pentru teste. Rularea testelor pornește întotdeauna de la această stare. Metodele utilizate sunt:

- *setUp()* – metoda ce este apelată automat înainte de fiecare test
- *tearDown()* – metodă ce este apelată automat după fiecare test

Exemplu:

```
import junit.framework.*;

public class TestFixture extends TestCase {
   public int n1, n2;

   // setarea valorilor n1, n2
   public void setUp(){
      n1=2;
      n2=3;

      System.out.println("setUp: n1: " + n1 + ", n2: " + n2);
   }


   // operatii cu cele doua valori
   public void testAdd(){
      double result= n1 + n2;
      assertTrue(result == 5);
   }

   public void testSub(){
      double result= n1 - n2;
      assertTrue(result < 0);
   }

   public void testMult(){
      double result= n1 * n2;
      assertTrue(result > 5);
   }

   public void testDiv(){
      double result= n1 / n2;
      assertTrue(result > 0);
   }

   //metoda rulata dupa executarea testelor
   public void tearDown(){
         n1 = 0;
         n2 = 0;
         System.out.println("tearDown: n1: " + n1 + ", n2: " + n2);
   }

}
```

La rularea testului **TextFixture** se poate observa că cele două metode setUp() și tearDown() sunt rulate de 4 ori: înainte și după fiecare dintre cele 4 metode de test: testAdd(), testSub(), testMult() și testDiv(). Rezultatul execuției **TextFixture** eșuează în cazul testDiv():



3.3. Executori de teste (*Test runners*)

JUnit oferă instrumente pentru definirea testelor, pentru execuția lor și pentru afișarea rezultatelor. Pentru scrierea unei clase care să execute toate testele dintr-o anumită clasă de test (derivată din TestCase) putem folosi:

```
import org.junit.runner.Result;
import org.junit.runner.JUnitCore;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result rezultat = JUnitCore.runClasses(TestFixtura.class);
      System.out.println("Test rulat cu succes: " +
                                              rezultat.wasSuccessful());
      if(!rezultat.wasSuccessful())
      {
        System.out.println("Teste esuate:");
        for (Failure esec : rezultat.getFailures()) {
             System.out.println(esec.toString());
        }
      }
   }
}
```

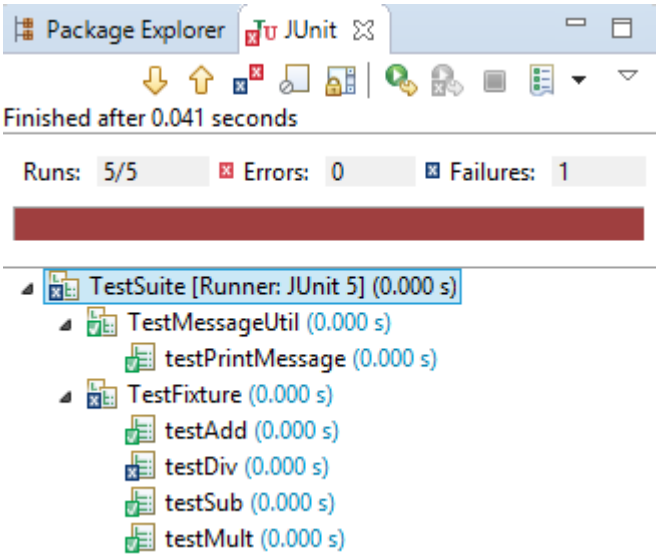
3.4. Suite de teste (*Test suites*)

Testele pot fi agregate în suite. Clasa Suite este un tip specializat de *test runner*. Aceasta permite construirea unui grup de teste din diferite clase. Pentru a utiliza această funcționalitate, vom crea o clasă nouă, care va fi adnotată cu **@RunWith(Suite.class)** și **@SuiteClasses(ClasaTest1.class, ClasaTest2.class, etc.)**. Testele scrise în clasele de test **TestFixture** (vezi secțiunea 3.2) și **TestMessageUtil** (vezi secțiunea 4.3) ar putea fi grupate într-o suită conform următorului exemplu:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
  TestMessageUtil.class,
  TestFixture.class
})
```

```
public class TestSuite {
	//clasa fara implementare
	//utilizata doar pentru a insera adnotarile de mai sus
}
```

Observație: ordinea claselor specificate **în @Suite.SuiteClasses({..})** contează. Testele vor fi rulate în ordinea în care au fost trecute în listă:



4. Integrarea bibliotecii JUnit în Eclipse

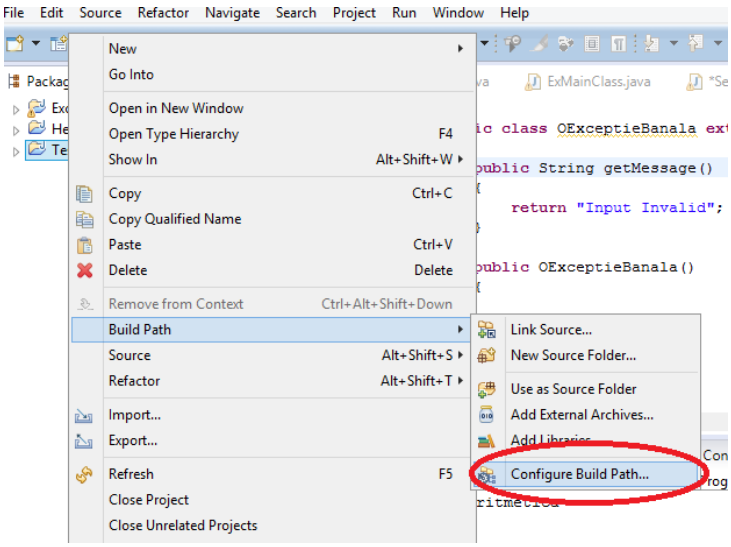
4.1. Integrarea JUnit, în Eclipse for Java Developers

Pas 1: Verificați versiunea mediului Eclipse

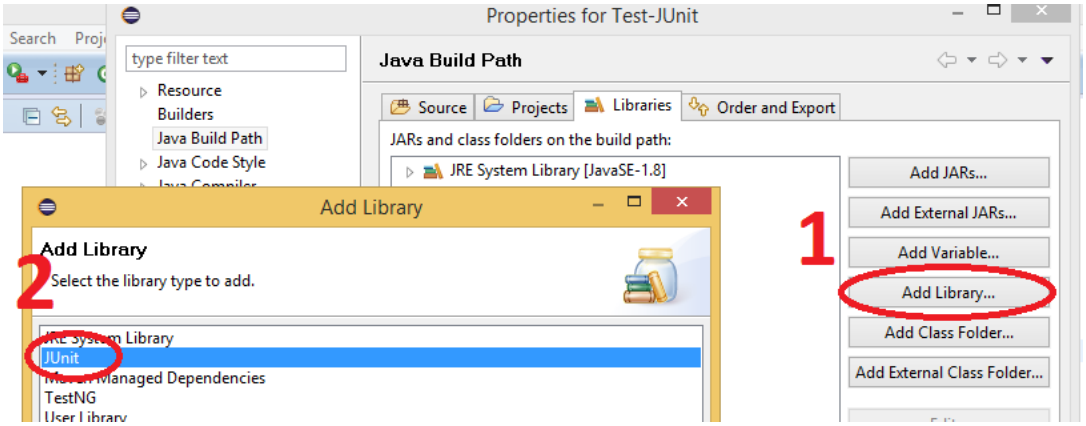
În mod normal, dacă se descarcă versiunea "Eclipse for Java Developers", atunci această distribuție include componentele din pachetul Java Developer Tools (JDT), care implicit conține și JUnit. Puteți verifica ce versiune de Eclipse aveți prin accesarea meniului Help -> About Eclipse.

Pas 2: Adăugați funcționalitățile JUnit

Creați un proiect nou. Faceți click dreapta pe noul proiect creat -> Build Path -> Configure Build Path



Accesați tab-ul **Libraries**. Faceți click pe butonul **Add Library**. Din fereastra deschisă, meniul "Select the library type to add" selectați JUnit. Click Next -> Finish.



4.2. Integrarea JUnit, în alte versiuni de Eclipse

Pas 1: Descărcați biblioteca JUnit

Biblioteca JUnit poate fi descărcată de la adresa http://www.junit.org sau de pe site-ul unde se poate găsi ultima versiune: https://github.com/junit-team/junit/wiki/Download-and-Install.

De la adresa menționată mai sus este necesar să descarcați, două fișiere .jar:

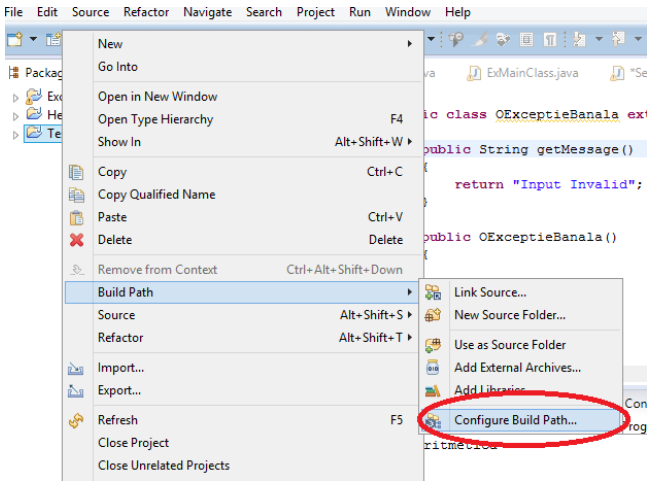
- Junit-x.xx.jar
- hamcrest-core-y.y.jar

Descărcați fișierele .jar și salvați-le undeva local, pe calculatorul dumneavoastră. La data creării acestui tutorial numele arhivelor sunt:

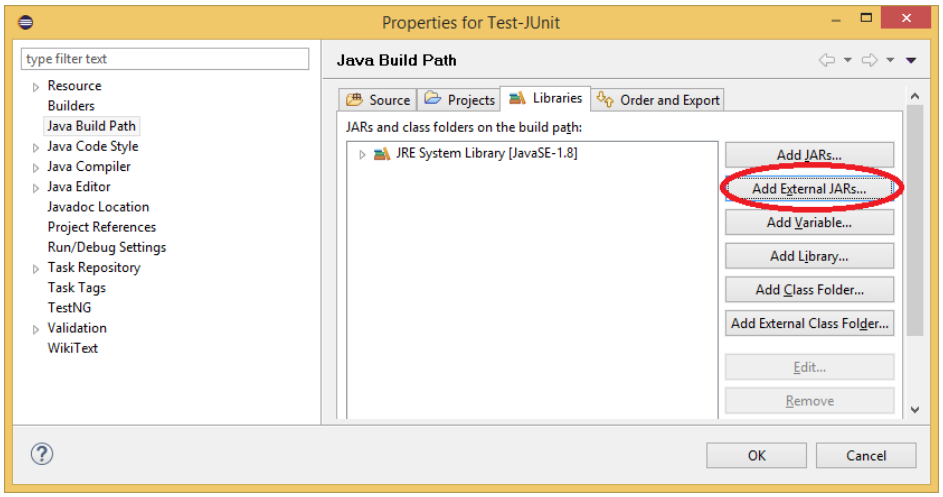
- **junit-4.12.jar** - arhiva poate fi descărcată direct de la adresa: http://search.maven.org/remotecontent?filepath=junit/junit/4.12/junit-4.12.jar
- **hamcrest-core-1.3.jar** - arhiva poate fi descărcată direct de la adresa: http://search.maven.org/remotecontent?filepath=org/hamcrest/hamcrest-core/1.3/hamcrest-core-1.3.jar

Pas 2: Setarea unui proiect în Eclipse pentru lucrul cu JUnit

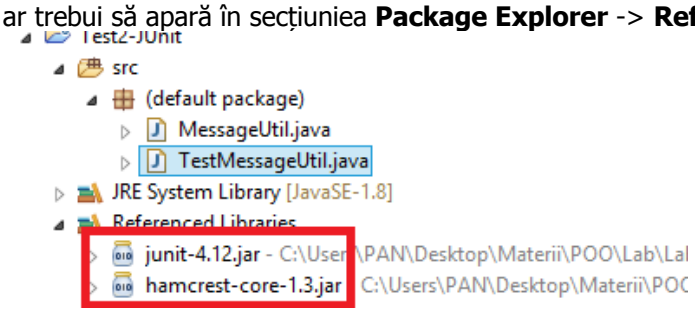
În Eclipse creați un proiect nou. Faceți click dreapta pe noul proiect creat -> Build Path -> Configure Build Path



Accesați tab-ul **Libraries**. Faceți click pe butonul **Add External JARs**:



Adăugați arhiva **junit-X.XX.jar** salvată local, pe calculator, la pasul 1
Repetați procedura de mai sus pentru a adauga și arhiva **hamcrest-core-y.y.jar**
Cele două arhive .jar ar trebui să apară în secțiuniea **Package Explorer** -> **Referenced Libraries**



4.3. Verificarea instalării corecte a bibliotecii JUnit în Eclipse

- Creați o clasă nouă **MessageUtil** care urmează a fi testată

```
public class MessageUtil {
   private String message;
   // Constructor
   public MessageUtil(String message) {
         this.message = message;
   }
   // afiseaza un mesaj
   public String printMessage() {
         System.out.println(message);
         return message;
   }
}
```

- Creați o clasă de test **TestMesageUtil** pentru testarea funcționalităților clasei **MessageUtil** (nu contează denumirea, însă este bine ca să existe o corelație între denumirile celor două clase)

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class TestMessageUtil {
   String message = "Hello World";
   MessageUtil messageUtil = new MessageUtil(message);

   @Test
   public void testPrintMessage() {
         assertEquals(message, messageUtil.printMessage());
   }
}
```

5. Testarea clasei CalcModel din aplicația CalcMVC utilizând JUnit

Descărcați proiectul **CalcMVC_JUnit** pus la dispoziție. Acesta conține fișierul JUnitTest.java în care s-au prezentat câteva teste demonstrative asupra clasei CalcModel din aplicația CalcMVC studiată în laboratorul de GUI-I.

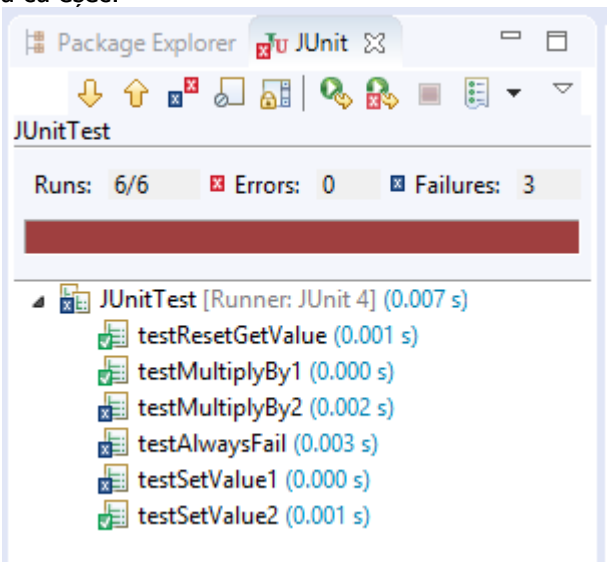
Studiați clasa JUnitTest, observați prezența adnotărilor @BeforeClass, @AfterClass, @Before, @After, @Test și analizați când și cum se execută fiecare metodă:

- JUnitTest() – Constructor executat înaintea fiecărui test
- setUpBeforeClass() – Executată o singură dată înaintea creării clasei de test
- tearDownAfterClass() – Executată o singură dată după distrugerea clasei de test
- setUp() – Executată înaintea începerii execuției fiecărui test
- tearDown()– Executată după terminarea execuției fiecărui test
- testResetGetValue(), testMultiplyBy1(), testMultiplyBy2(), testSetValue1(), testSetValue2(), testAlwaysFail() – Metodele de test care sunt executate pe rând

```
import static org.junit.Assert.*;
import org.junit.*;


public class JUnitTest {

	private static CalcModel m;

	private static int nrTesteExecutate = 0;

	private static int nrTesteCuSucces = 0;


	public JUnitTest() {
		System.out.println("Constructor inaintea fiecarui test!");
	}

	@BeforeClass
	public static void setUpBeforeClass() throws Exception {
		System.out.println("O singura data inaintea executiei setului
                                                    de teste din clasa!");
		m = new CalcModel();
	}

	@AfterClass
	public static void tearDownAfterClass() throws Exception {
		System.out.println("O singura data dupa terminarea executiei
                                        setului de teste din clasa!");
		System.out.println("S-au executat " + nrTesteExecutate + "
			teste din care "+ nrTesteCuSucces + " au avut succes!");
	}

	@Before
	public void setUp() throws Exception {
		System.out.println("Incepe un nou test!");
		nrTesteExecutate++;
	}

	@After
	public void tearDown() throws Exception {
		System.out.println("S-a terminat testul curent!");
	}
```

```
	@Test
	public void testResetGetValue() {
		m.reset();
		String t = m.getValue();
		assertNotNull(t); // verifica t sa nu fie null
		assertEquals(t,"1"); // verifica continutul lui t sa fie
                                                identic cu "1"
		nrTesteCuSucces++;
	}

	@Test
	public void testMultiplyBy1() {
		m.reset();
		m.multiplyBy("7");
		m.multiplyBy("12");
		String t = m.getValue();
		assertNotNull(t); // verifica t sa nu fie null
		assertEquals(t,"84"); // verifica t sa nu fie null
		nrTesteCuSucces++;
	}

	@Test
	public void testMultiplyBy2() {
		m.reset();
		m.multiplyBy("7");
		m.multiplyBy("12");
		String t = m.getValue();
		assertTrue(t=="84"); // verifica referinta lui t sa fie
                                identica cu referinta lui "84"
		nrTesteCuSucces++;
	}

	@Test
	public void testSetValue1() {
		m.setValue("50");
		String t = m.getValue();
		assertNotNull(t); // verifica t sa nu fie null
		assertSame(t,"50"); // verifica referinta lui t sa fie identica
                                                cu referinta lui "50"
		nrTesteCuSucces++;
	}

	@Test
	public void testSetValue2() {
		m.setValue("50");
		String t = m.getValue();
		assertNotNull(t); // verifica t sa nu fie null
		assertEquals(t,"50"); // verifica continutul lui t sa fie
                                                        identic cu "50"
		nrTesteCuSucces++;
	}

	@Test
	public void testAlwaysFail() {
		fail("Esuat!"); // intotdeauna esueaza
		nrTesteCuSucces++;
	}

}
```

Rulați clasa JUnitTest ca și JUnit Test. Așteptați execuția testelor. Analizați rezultatele referitoare la terminarea lor cu succes sau cu eșec:



Analizați conținutul afișat la consolă din care reiese clar modul în care este creată clasa de test și cum sunt pregătite și apoi executate pe rând testele:

```
O singura data inaintea executiei setului de teste din clasa!
Constructor inaintea fiecarui test!
Incepe un nou test!
S-a terminat testul curent!
                                        se repetă pentru fiecare test!
O singura data dupa terminarea executiei setului de teste din clasa!
S-au executat 6 teste din care 3 au avut succes!
```

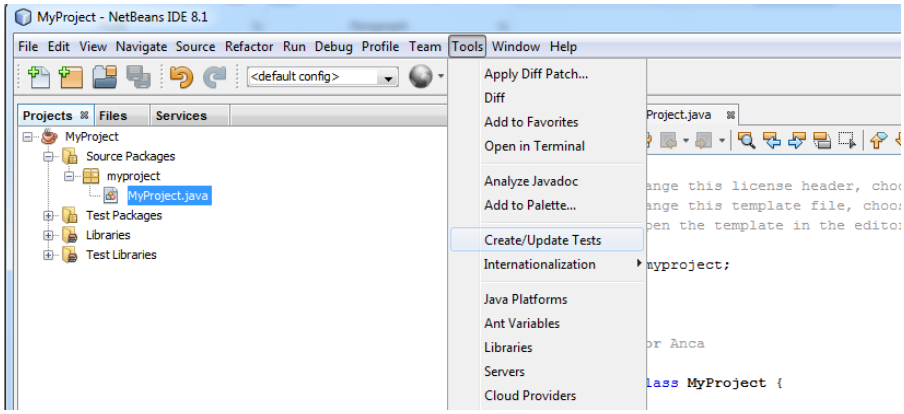
6. JUnit în alte medii de programare

6.1. NetBeans

Biblioteca JUnit este integrată în mediul NetBeans. Nu este nevoie de activități suplimentare pentru integrarea acesteia.

Pașii necesari pentru crearea de teste în NetBeans:

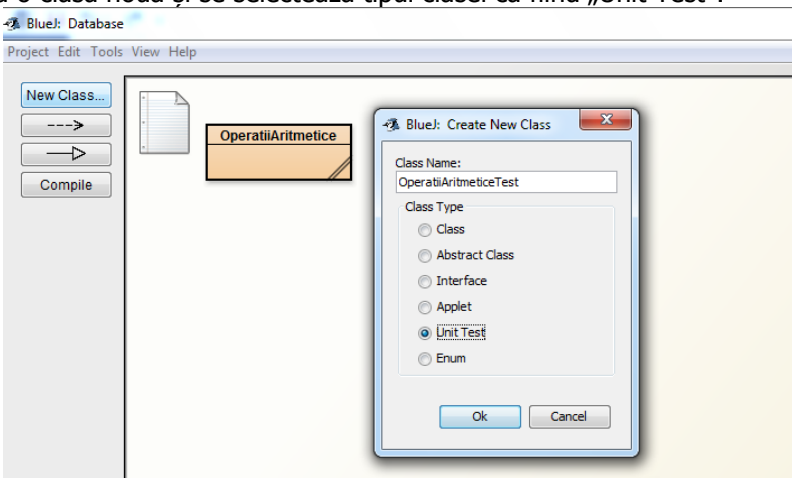
- Tools -> Create Tests



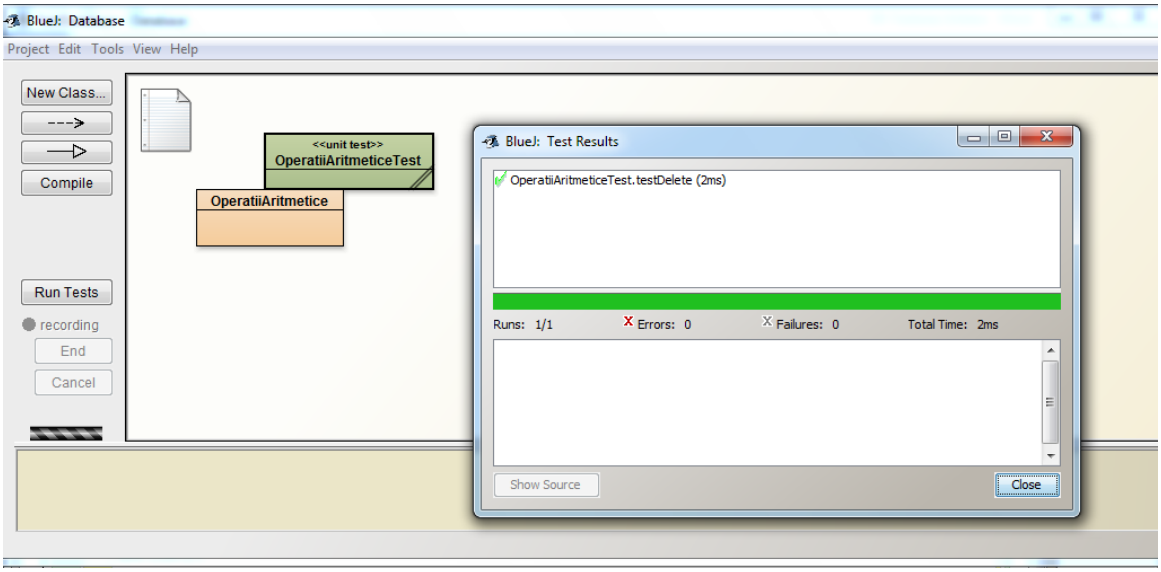
- Setați numele clasei de test cu numele clasei care trebuie testată urmată de String-ul „Test” la final, și se va genera automat o clasă de test
- Rularea testelor se face din meniu: Run -> Run File

6.2. BlueJ

În BlueJ testarea unitară este de asemenea inclusă implicit. În prealabil uneltele de testare trebuie activate din meniul Tools->Preferences->Interface->Show unit testing tools. Pentru crearea unei clase de test se creează o clasă nouă și se selectează tipul clasei ca fiind „Unit Test”:



Pentru rularea testelor se apasă butonul de „Run Tests”, rezultatele fiind afișate într-o fereastră nouă ca în următoarea figură, sau dând click dreapta pe clasa de test și selectând „Test All”:



7. Mersul lucrării

7.1. Parcurgeți și înțelegeți toate exemplele din laborator.

7.2. Implementați problema care determină dacă un an este sau nu bisect. Un an este bisect dacă este divizibil cu 4, exceptând cazurile când este divizibil cu 100 fără a fi divizibil cu 400:

- Creați scheletul clasei **AnBisect**, cu metoda **public boolean esteAnBisect(int an).** Atenție, deocamdată lăsați metoda fără implementare.

- Creați o clasă de test **TestAnBisect**. Implementați teste pentru a verifica execuția metodei **esteAnBisect()** din clasa **AnBisect** pentru următoarele exemple: 203 (an comun), 4 (an bisect), 99 (an comun), 100 (an comun), 200 (an comun), 400 (an bisect), 500 (an comun), 1000 (an comun), 1600 (an bisect), 2018 (an comun).

Condiție: folosiți doar metoda assertTrue(...) pentru verificarea condițiilor. Pentru ca testul să treacă, aceasta va trebui să verifice condițiile în funcție de datele de intrare ca în următoarele două exemple: **assertTrue(esteAnBisect(4))** sau **assertTrue(!esteAnBisect(99))**.

- Implementați metoda **esteAnBisect(int an)** din clasa **AnBisect**.
- Rulați testele. Toate testele au trecut cu succes? Dacă nu, atunci implementați metoda astfel încât să returneze rezultate corecte pentru toate cazurile.
- Creați o altă clasă de test **TestAnComun**. Implementati teste pentru a verifica execuția corectă a metodei **esteAnBisect()** din clasa **AnBisect** pentru aceleași exemple: 203 (an comun), 4 (an bisect), 99 (an comun), 100 (an comun), 200 (an comun), 400 (an bisect), 500 (an comun), 1000 (an comun), 1600 (an bisect), 2015 (an comun).
  Condiție: Spre deosebire de pasul 6.3.2, folosiți doar metoda assertFalse(...) pentru verificarea condițiilor. Pentru ca testul sa treacă, aceasta va trebui să verifice condițiile în funcție de datele de intrare ca în următoarele două exemple: **assertFalse(!esteAnBisect(4))** sau **assertFalse(esteAnBisect(99))**.
- Folosiți funcționalitatea ***Test suites*** din secțiunea 3.4 pentru rula cele două clase de test **TestAnBisect** și **TestAnComun**.

7.3. Extindeți un proiect existent, implementat la laboratoarele anterioare, cu mai multe cazuri de test folosind JUnit.

Referințe

[1] http://www.tutorialspoint.com/junit/junit_tutorial.pdf
[2] http://www.junit.org
[3] https://github.com/junit-team/junit/wiki/
[4] https://ro.wikipedia.org/wiki/An_bisect