

JUnit

Unit Testing cu JUnit

Conf. univ. dr. Catalin Boja

catalin.boja@ie.ase.ro

<http://acs.ase.ro>



Dep. de Informatică și Cibernetică Economică
ASE București

Resurse

- Lasse Koskela - *Effective Unit Testing*, Manning, 2013
- Lasse Koskela - *Practical TDD and Acceptance TDD for Java Developers*, Manning, 2007
- Andrew Hunt, David Thomas - *Pragmatic Unit Testing in Java with JUnit*, The Pragmatic Programmers, 2004
- <http://www.junit.org>
- <http://www.vogella.com/tutorials/JUnit/article.html>

Ce este Unit Testing?

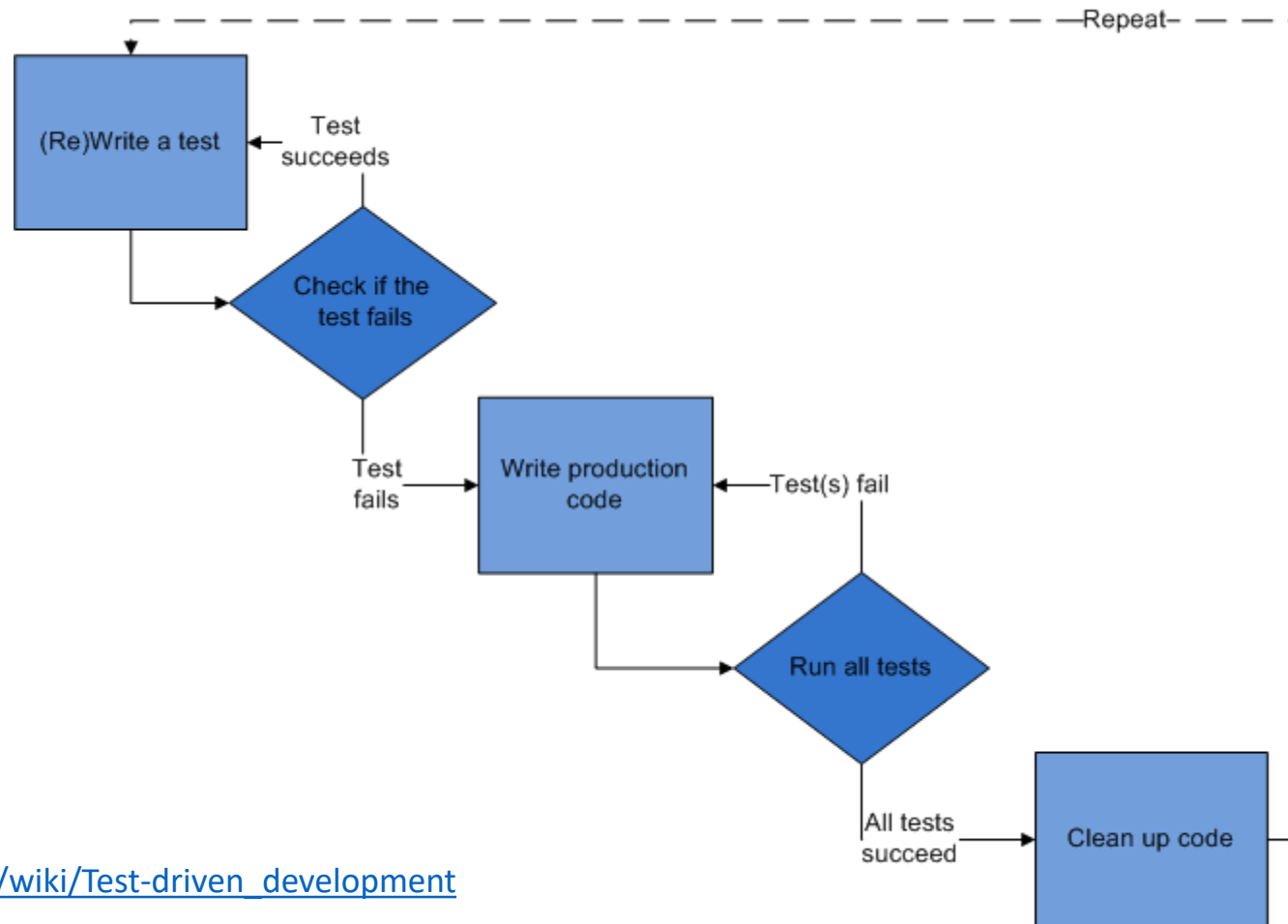
- Metoda simpla si rapida de testare a codului sursa de către programatori
- Are loc în faza de dezvoltare și este un instrument destinat programatorilor
- Un *unit test* este o secvență de cod scrisa de un programator pentru a evalua o parte bine definite, de mici dimensiuni, din codul sursa testat – clasă sau metodă
- Un *unit test* evaluează modul de funcționare al unei metode într-un context bine definit
- Un *unit test* este blocul de baza pentru abordarea *Test-Driven Development*

Ce înseamnă Test-Driven Development (TDD)

Test-driven development (TDD) *is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards* [http://en.wikipedia.org/wiki/Test-driven_development]

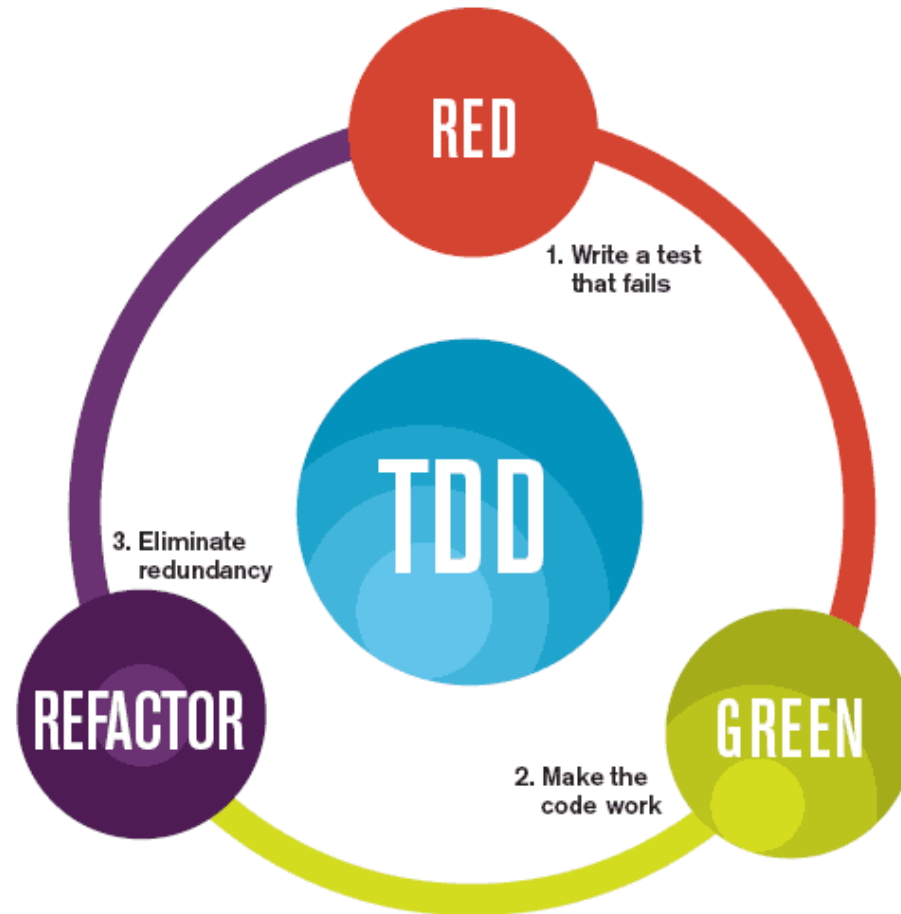


Ce înseamnă Test-Driven Development (TDD)



http://en.wikipedia.org/wiki/Test-driven_development

Ce înseamnă Test-Driven Development (TDD)



Tipuri de testare a codului sursă

Acceptance testing

- Teste realizate de client. Testează dacă design-ul corespunde cu ceea ce s-a dorit

System testing

- Testarea întregului sistem (toate componentele). Testează dacă sistemul funcționează conform design-ului

Integration Testing

- Testarea mai multor componente ce sunt combinate sau integrate

Unit Testing

- Testare celor mai mici părți din cod (clase sau metode)

Tipuri de testare a codului sursă

Regression Testing

- Testarea automata a aplicației după implementarea unor modificări astfel încât să fie evitată reapariția unor erori (bug-uri) anterioare

Black-box testing

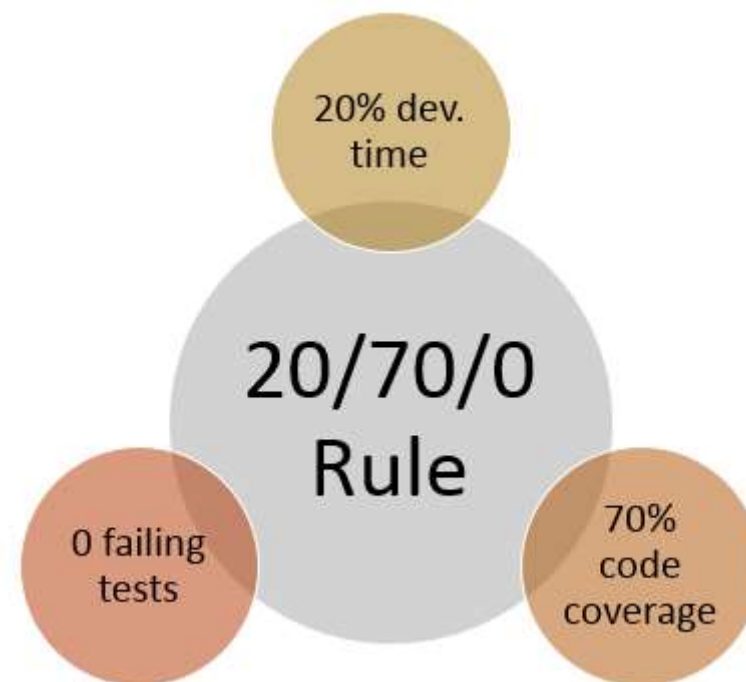
- Testarea interfeței publice a unor componente fără a avea informații cu privire la implementare, structura internă, etc

White-box testing (glass-box testing)

- Testarea unor componente despre care există informații complete

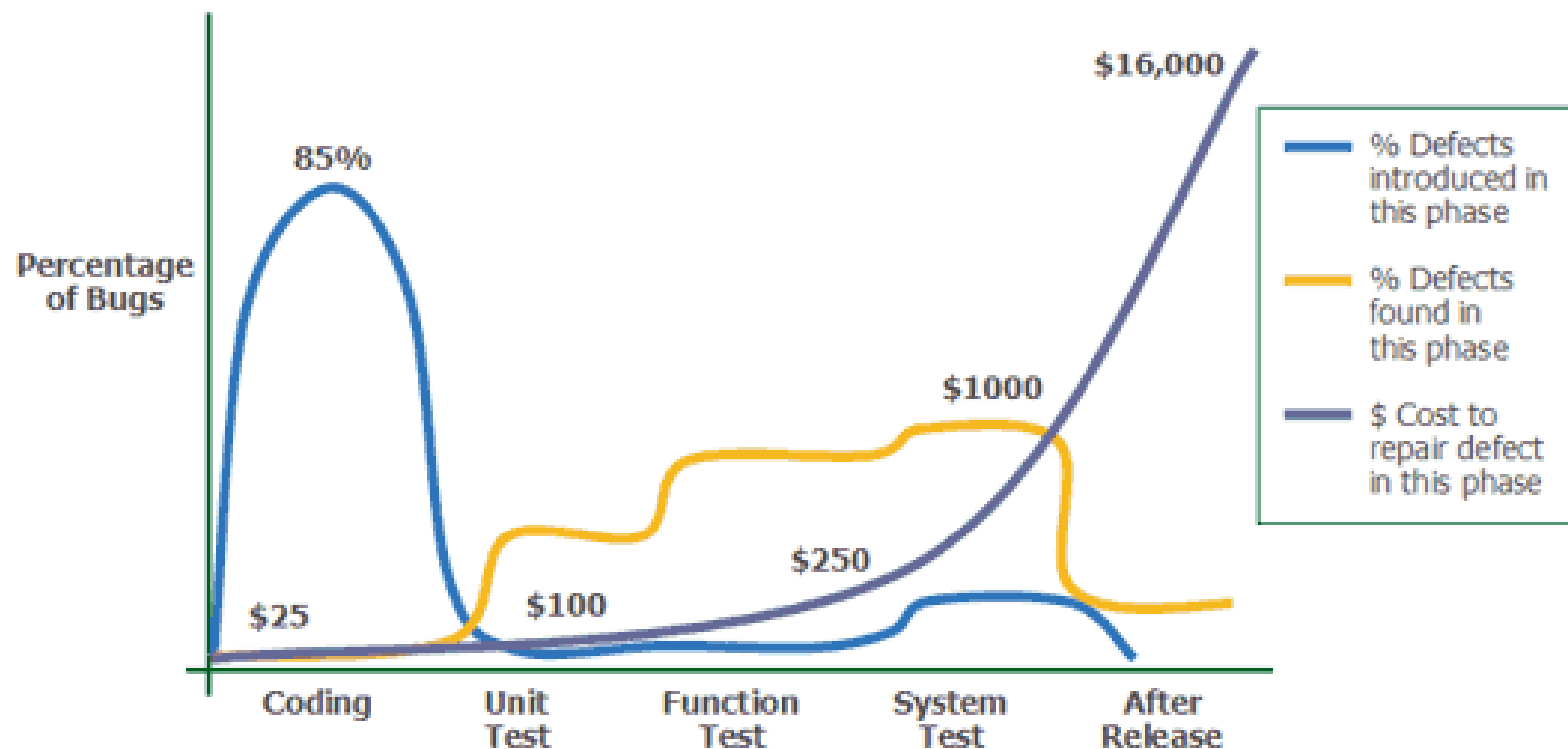
Motive să folosești Unit Testing

- Ușor de scris
- Testele pot fi scrise ad-hoc atunci când ai nevoie de ele
- Deși simple, pe baza lor se pot defini colecții de teste – **Test Suites**
- Pot fi rulate automat de fiecare dată când e nevoie (*write once, use many times*)
- Există multe framework-uri și instrumente ce simplifică procesul de scriere și rulare
- Reduc timpul pierdut pentru debugging și pentru găsirea bug-urilor
- Reduc numărul de bug-uri în codul livrat sau integrat
- Crește rata bug-urilor identificate în faza de scrierea a codului



Source: Anthony Langsworth, Unit Testing: The 20/70/0 Rule

Motive să folosești Unit Testing



Source: *Applied Software Measurement*, Capers Jones, 1996

Motive sa NU folosești Unit Testing

- Chiar trebuie să îmi testez codul ? Dar este scris de mine !!!
- Nu am timp. Sunt atât de ocupat. Am altceva de făcut.
- Este greu
- Eu nu știu să fac asta
- Dar nu e trecut în specificații



<http://www.dylanmeconis.com/how-not-to-write-comics-criticism/>

Ce să testezi ?

- Scenarii reale
- Limitele intervalelor in care exista valorile de test
 - Mai mici ca 0, cum ar fi -1, -2, ..., -100, ...
 - 0
 - Mai mare ca 0, cum ar fi 1, 2, ..., 100...
 - null
 - CORRECT Boundary Conditions [Alasdair Allan - *Pragmatic Unit Testing in Java with Junit*]
- The Right-BICEP [Alasdair Allan - *Pragmatic Unit Testing in Java with Junit*]

CORRECT Boundary Conditions

Andrew Hunt, David Thomas - *Pragmatic Unit Testing in Java with JUnit*

- *Conformance* – Valoarea are formatul corect ?
- *Ordering* – Setul de valori trebuie să fie ordonat sau nu ?
- *Range* – este valoarea între limitele (maxim și minim) acceptate?
- *Reference* – Codul referă componente externe care NU sunt controlate direct?
- *Existence* – Valoarea există (ex. este non-null, non-zero, parte dintr-un set, etc.)?
- *Cardinality* – Setul de test conține suficiente valori (regula 0-1-n) ?
- *Time* (absolut și relativ) – Totul se întâmplă în ordine ? La momentul potrivit ? Într-un timp finit ? (UTC vs. DST)

CORRECT Boundary Conditions

Andrew Hunt, David Thomas - *Pragmatic Unit Testing in Java with JUnit*

- Valorile de intrare total necorespunzătoare sau inconsecvente, cum ar fi una din „! * W: Xn & Gi / w> g / h # WQ @”.
- Datele formatate greșit, cum ar fi o adresă de e-mail fără un domeniu de nivel superior („fred @ foobar”).
- Valori invalide sau lipsă (cum ar fi 0, 0: 0, "" sau nul).
- Valorile care depășesc limite rezonabile, cum ar fi vârsta unei persoane de 10.000 de ani
- Duplicat în liste care trebuie să conțină valori unice.
- Listele ordonate care nu sunt, și invers. Încercați să transmiteți o listă pre-sortată unui algoritm de sortare, de exemplu sau chiar o listă sortată invers.
- Lucruri care nu sunt într-o ordine cerută sau care nu se întâmplă în ordinea scontată, cum ar fi încercarea de a imprima un document înainte de logare, de exemplu..

Andrew Hunt, David Thomas - *Pragmatic Unit Testing in Java with JUnit*

The Right-BICEP

Andrew Hunt, David Thomas - *Pragmatic Unit Testing in Java with JUnit*

- Right . Sunt rezultatele corecte ?
- B . Sunt limitele (***B**oundary conditions*) definite CORRECT?
- I . Poți verifica opusul situației (***I**nverse relationships*)?
- C . Se poate verifica rezultatul si prin alte metode (**C**ross-check)?
- E . Se pot evalua (forța) condițiile care generează erori (**E**rror-conditions)?
- P . Performanta execuției este intre limite (**P**erformance characteristics) ?

Condiții pentru erori/exceptii

- Sistemul rămâne fără memorie
- Lipsa de spațiu disponibil pe disc
- Probleme cu ora sistemului
- Disponibilitatea rețelei și erori specifice lucrului în rețea
- Gradul de încărcare al sistemului (memorie, procesor, etc)
- Paleta de culori limitată (pentru testare interfață utilizator)
- Rezoluție video foarte mare sau foarte mică

Exemplu

```
public interface StackExercise {  
    /**  
     * Return and remove the most recent item from the top of the stack.  
     * Throws StackEmptyException if the stack is empty  
     */
```

```
    public String pop() throws  
    StackEmptyException;
```

```
    /**  
     * Add an item to the top of the stack.  
     */
```

```
    public void push(String item);
```

```
    /**  
     * Return but do not remove the most recent item from the top of the  
     stack.
```

```
     * Throws StackEmptyException if the stack is empty  
     */
```

```
    public String top() throws StackEmptyException;  
    /**
```

```
     * Returns true if the stack is empty.  
     */
```

```
    public boolean isEmpty();  
}
```

SCENARIU – O metoda OK

```
int sum(int* values, int no){
    int result = 0;
    for(int i = 0; i<no; i++){
        result+=values[i];
    }
    return result;
}
```

Calculul sumei elementelor unui vector cu valori întregi

Implementată în C

It's simple, right ?

SCENARIU – UNIT Testing

1. unit test pentru condiții NORMALE cu valorile = {1,2,3}, no = 3; expected = 6
2. unit test pt. condiții EXTREME cu values = null, no = 0; expected exception sau error code
3. unit test pt. condiții EXTREME cu values = {1,2,3}, no = 0; expected exception sau error code
4. unit test pt. condiții WRONG cu values = {1,2,3}, no = -3; expected exception sau error code
5. unit test pt. condiții LIMITA cu values = {INT_MIN,-23}, no = 2; expected exception

SCENARIU – UNIT TESTING

6. unit test pt. condiții LIMITA cu values = {INT_MIN,-23}, no = 2; expected exception
7. Cardinality unit test cu values = {3}, no = 1; expected = 3
8. Ordering unit test (nu e necesar) cu values = {3,2,1}, no = 3; expected = 6
9. Buffer overflow unit test (pentru suma a maxim 1000 valori) cu values = {1,2,3...}, no = 1001; expected exception

SCENARIU – O metoda OK

În C nu avem excepții sau nu ni se permite să generăm excepții

Metoda returnează un cod de eroare de fiecare dată când o condiție nu este verificată

Este aceeași metodă

```
int sum_final(int* values, int no, int* result) {  
    //testing limits and special cases  
    if (values == NULL || no <= 0 || no > 1001) {  
        return ERROR_CODE;  
    }  
  
    int value = 0;  
    for (int i = 0; i < no; i++) {  
        //testing for overflow  
        if ((value > 0) && (values[i] > INT_MAX - value)){  
            //will overflow  
            return ERROR_CODE;  
        }  
        if ((value < 0) && (values[i] < INT_MIN - value)){  
            //will overflow  
            return ERROR_CODE;  
        }  
  
        //do the sum  
        value += values[i];  
    }  
  
    *result = value;  
    return SUCCESS_CODE;  
}
```

Obiecte Mock

Descriere:

- Un unit test trebuie sa evalueze execuția unei metode, însă uneori acest lucru necesita obiecte/condiții externe metodei
- Este un *testing pattern*
- Reprezintă un înlocuitor al obiectului real

Definire:

- Obiectele *mock* sunt descrise prin interfață
- Interfața este implementata în soluția reala dar și în Unit Test
- <http://easymock.org/>

Obiecte Mock – Când ai nevoie de ele ?

- Obiectul real are un comportament nedeterminist (produce rezultate imprevizibile; de ex. un flux de cotații bursiere)
- Obiectul real este dificil de configurat.
- Obiectul real are un comportament greu de declanșat (de exemplu, o eroare de rețea).
- Obiectul real este lent.
- Obiectul real are (sau este) o interfață de utilizator.
- Testul trebuie să întrebe obiectul real despre cum a fost folosit (de exemplu, este posibil să fie nevoie să verificați dacă metoda callback chiar a fost apelată).
- Obiectul real nu există încă (o problemă comună când interacționăm cu alte echipe sau cu noi sisteme hardware).

[Tim Mackinnon]

TESTARE ramuri de execuție

- Numărul de teste depinde de complexitatea ciclomatica a metodei
- Testarea ramurilor de execuție/ acoperirea condițiilor (Branch Testing/ Condition Coverage) – Implementarea unui număr suficiente de teste pentru a se asigura că fiecare alternativă de execuție a fost executată cel puțin o dată într-un test; pentru fiecare ramură de execuție (branch), oferiți valori care generează rezultate adevărate și false

TESTARE metode externe

- NU le testați
- Folosiți mock-uri sau metode stub pentru a le înlocui
- Utilizați principiile Clean Code (cum ar fi SOLID) pentru a refactoriza metoda testată
 - Împărțiți-o în mai multe metode simple
 - Utilizați interfețe pentru a elimina dependențele externe

Reguli simple pentru metode ok

- Scrie teste în paralel cu implementarea metodei
- Single responsibility
- Keep It Simple & Stupid
- Deleagă
- Folosește interfețe



Librării pentru testare unitară

- **AppleScript:** ASUnit, ATest
- **C/C++:** Boost Test Library, Google Test, QT Test, unity
- **Java:** JUnit
- **C#:** NUnit
- **Python:** unittest

https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

Caracteristici ale testelor unitare

- *simple – foarte simple*
- *rapide*
- ***Fiecare test trebuie să verifice o singură condiție – un singur apel assert***
- *independente și repetabile (folosește mockup-uri sau stub-uri)*
- *portabile și reutilizabile*



Caracteristici ale testelor unitare

- *bine* organizate (clean)
- reflecta structura codului testat
- testele aferente unei metode/clase sunt grupate Test Case-uri / suite
- trebuie să ofere cât mai multe *informații* despre problema testată



Concepte ale testării unitare

- ***assert*** este o acțiune/instrucțiune care verifică dacă o condiție este adevărată
- un ***test case*** conține unul sau mai multe teste
- un ***test suite*** este o colecție de unul sau mai multe cazuri de testare
- obiectele comune și subrutinele des folosite sunt definite într-un test case ca și ***test fixture***
- atunci când o metodă assert eșuează, tipărește numele fișierului, numărul liniei și un mesaj pe care îl puteți personaliza
- rezultatul unui assert poate fi ***success***, ***nonfatal failure*** sau ***fatal failure*** (anulează funcția curentă)

Proprietăți ale testelor

Principii FIRST

- Fast
- Isolated/Independent
- Repeatable
- Self-Validating
- Thorough

[“Pragmatic Unit Testing in Java 8 with JUnit”](#) by *Andy Hunt; Jeff Langr; Dave Thomas*

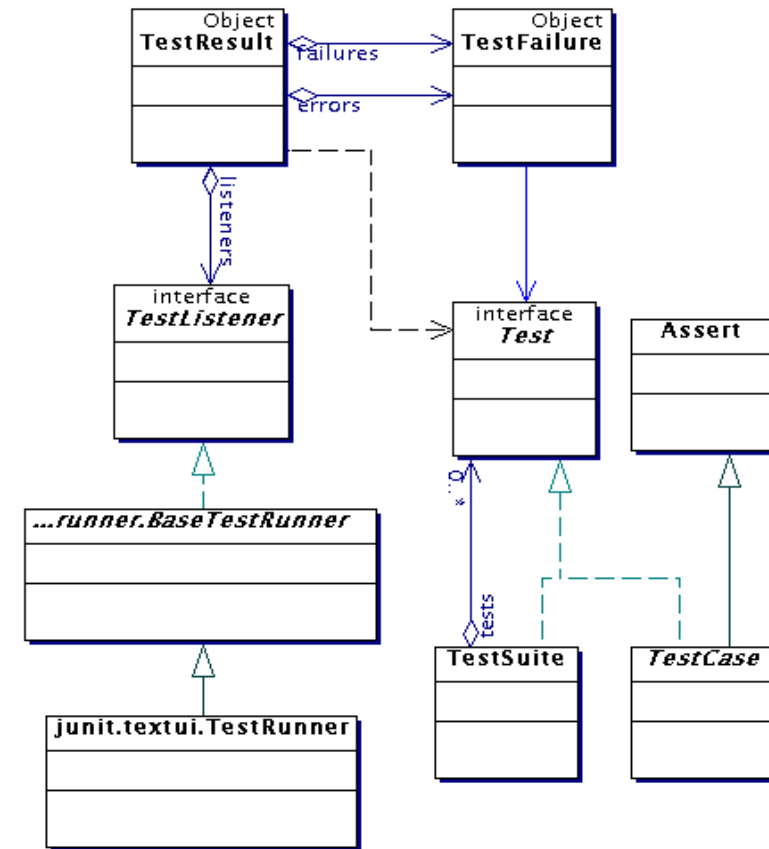
Proprietăți ale testelor

- **Automate** – pot fi rulate automat
- **Repetabile** – pot fi rulate de nenumărate ori cu aceleași rezultate
- **Independente** – independente de condițiile specifice mediului de execuție și independente de alte teste
- **Complete** – testează tot ce poate genera erori la un moment dat
- **Reale** – sunt la fel de importante ca și codul testat; sunt tot programe și trebuie tratate ca atare (NU sunt secvențe de cod scrise pentru un moment anume și apoi aruncate)

[“Pragmatic Unit Testing in Java 8 with JUnit”](#) by *Andy Hunt; Jeff Langr; Dave Thomas*

Ce este JUnit

- Un instrument pentru *Test-Driven Development*
- JUnit este un framework de clase ce permite scrierea și execuția de teste pentru diferite metode/clase din cod



Istoric

- Kent Beck a dezvoltat in anii 90 primul instrument de testare automata, xUnit, pentru *Smalltalk*
- Beck si Gamma (Gang of Four) au dezvoltat JUnit in timpul unui zbor de la Zurich la Washington, D.C.
- Martin Fowler: *“Never in the field of software development was so much owed by so many to so few lines of code.”*
- Junit a devenit instrumentul standard pentru procesele de dezvoltare de tip TDD - *Test-Driven Development* in Java
- Junit este componenta standard in multiple IDE-uri de Java (Eclipse, BlueJ, Jbuilder, DrJava, IntelliJ)
- Instrumentele de tip Xunit au fost dezvoltate si pentru alte limbaje (Perl, C++, Python, Visual Basic, C#, ...)

Arhitectura JUnit

- **TestRunner** execută teste și raportează **TestResults**
- Testul se scrie prin extinderea clasei abstracte **TestCase**
- Pentru a scrie teste, trebuie să înțelegi clasa **Assert**
- Fiecare metodă de tip *assert* are următorul set de parametri: *message*, *expected-value*, *actual-value*
- Metodele *assert* pentru valori reale primesc un parametru suplimentar – variația
- Fiecare metodă *assert* are o formă echivalentă ce nu primește parametrul de tip mesaj – varianta NU este recomandată
- Mesajul metodei *assert* este folosit la documentarea testului și ajută la parcurgerea log-urilor asociate testelor eșuate

Metode de tip *assert*

- assertEquals(expected, actual)
- assertEquals(message, expected, actual)
- assertEquals(expected, actual, delta)
- assertEquals(message, expected, actual, delta)
- assertFalse(condition)
- assertFalse(message, condition)
- Assert(Not)Null(object)
- Assert(Not)Null(message, object)
- Assert(Not)Same(expected, actual) – verifica daca cele 2 referinte (NU) sunt identice
- Assert(Not)Same(message, expected, actual)
- assertTrue(condition)
- assertTrue(message, condition)
- fail(message)

Concepte JUnit

- **Fixture** – set de obiecte utilizate în test
- **Test Case** – clasă ce definește setul de obiecte (fixture) pentru a rula mai multe teste
- **Setup** – o metodă/etapă de definire a setului de obiecte utilizate (fixture), înainte de testare.
- **Teardown** – o metodă/etapă de distrugere a obiectelor (fixture) după terminarea testelor
- **Test Suite** – colecție de cazuri de testare (test cases)
- **Test Runner** – instrument de rulare a testelor (test suite) și de afișare a rezultatelor

Caracteristici JUnit

- Pachetul de funcții este destul de simplu
- Codul JUnit nu este livrat clientului la build/export
- Testele se scriu în Java
- Dacă un test eșuează celelalte sunt executate în continuare
- Testul are o structură simplă:
 - Pregătește condițiile inițiale (creare obiecte, inițializare resurse)
 - Apelează metodele ce urmează să fie testate
 - Verifică dacă metoda funcționează comparând rezultatele generate cu cele așteptate
 - Eliberează resursele utilizate

Caracteristici JUnit

- O metoda de testare poate conține mai multe assert-uri, deși este recomandată una singură
- Simplificarea testelor ajuta la identificarea rapida a bug-urilor
- Obiectivul este de a scrie teste care eșuează astfel încât să fie corectate erorile din timp
- Testele pot fi combinate în colecții – *Test Suites*

TestCase tutorial – JUnit3

1. Definieste o subclasa pentru *TestCase*

```
import junit.framework.TestCase;
public class TestMath extends TestCase {
} //
```

- Numele clasei este important si trebuie să fie de forma – ***TestMyClass*** sau ***MyClassTest***
- Convenția de nume permite componentei TestRunner să identifice automat automat clasele utilizate la testare

TestCase tutorial – JUnit3

2. În clasă se definește un constructor prin intermediul căruia se poate selecta anumite metode din test (Test Suites)

```
import junit.framework.TestCase;

public class TestMath extends TestCase{

    public TestMath(String name) {
        super(name);
    }

}
```

TestCase tutorial – JUnit3

3. In clasa ce extinde `TestCase` se definesc unit teste ce folosesc metodele `assert` pentru a evalua modul de executie a metodelor din clasa testate

Recomandari:

- Un unit test trebuie sa evalueze maxim o metoda
- Fiecare unit test trebuie sa evalueze un scenariu bine definit – maxim un `assert` pe unit test
- Unit test-ele trebuie sa fie independente (sa poata fi evaluate usor si sa poata fi inserate in `TestSuites`)

TestCase tutorial – JUnit3

4. In clasa ce extinde TestCase se definesc obiectele necesare testarii (fixture) si se implementeaza metodele setUp() si tearDown()

```
public class TestStudent extends TestCase {  
    //fixture pt TestCase  
    String examen = null;  
    Student stud= null;  
  
    //setUp - functie care se executa inaintea fiecarui unit test  
    public void setUp(){  
        System.out.println("Apel setUp");  
        examen = "CTS";  
        stud= new Student();  
    }  
  
    //tearDown - functie care se executa dupa fiecare unit test  
    public void tearDown(){  
        System.out.println("Apel tearDown");  
    }  
}
```

TestCase tutorial – JUnit3

5. Teste pot fi combinate prin definirea colecții externe ce încarcă toate metodele din clasa de test sau doar pe cele din suite.

```
public class TestComposite extends TestCase {
    public TestComposite(String name){
        super(name);
    }

    static public Test suite(){
        TestSuite suite = new TestSuite();

        //incarca toata clasa
        suite.addTestSuite(TestLargest.class);
        //incarca doar metodele din suit
        suite.addTest(new TestMath("testMedie"));

        return suite;
    }
}
```

TestCase tutorial – JUnit3

6. Pot fi definite si metode **setUp()** si **tearDown()** globale

```
//setUp global
public static void setUpGlobal(){
    System.out.println("setUp GLOBAL");
}

//tearDown global
public static void tearDownGlobal(){
    System.out.println("tearDown GLOBAL");
}

public static Test suite(){
    TestSuite wrapper = new TestSuite();
```

```
    wrapper.addTestSuite(TestStudent.class);

    TestSetup setupGlobal = new TestSetup(wrapper) {
public void setUp(){
        TestStudent.setUpGlobal();
}
public void tearDown(){
        TestStudent.tearDownGlobal();
}
};

    return setupGlobal;
}
```

TestCase tutorial – JUnit3

7. Pot fi definite metode *assert* proprii prin definirea unei clase de baza de tip Test care sa fie extinsa in alte teste

Structura Test Case

- `setUp()`
 - definește și construiește resursele sau obiectele (*fixture*) necesare rulării testelor
 - se apelează înaintea fiecărei metode de testare
 - in cazul seturilor de teste, metoda poate fi una generica, executată o singură dată, daca este definite in interiorul unui *TestSetup*
- `tearDown()`
 - eliberează/distruge resursele alocate testului
 - se apelează după fiecare metoda de testare
 - in cazul seturilor de teste, metoda poate fi una generica, executată o singură dată, daca este definite in interiorul unui *TestSetup*

Test Suites – JUnit3

- Colecție de teste ce vor fi evaluate împreună
- Test unit definit prin combinarea totala sau parțială a altor test case-uri
- Unit teste-ele trebuie sa fie definite fără a avea legătură între ele

Test Suites – JUnit3

- **Colecție de teste parțiale – executate în ordinea în care sunt adăugate în suită:**

```
public static Test suite() {  
    suite.addTest(new TestMath("testMedie"));  
    suite.addTest(new TestMath("testSuma"));  
    return suite;  
}
```

- **TestSuite ce include integral alte Test Case-uri:**

```
public static Test suite() {  
    TestSuite suite = new TestSuite();  
    suite.addTestSuite(TestMath.class);  
    suite.addTestSuite(TestStudent.class);  
    return suite;  
}
```

JUnit 4

- Uses annotations to mark framework specific methods
- *@Test* – marks a method as a unit test
- *@Before* – marks the setUp method (the method name must be setUp)
- *@After* – marks the tearDown method
- *@BeforeClass* – marks the setUpBeforeClass method
- *@AfterClass* – marks the tearDownAfterClass method

JUnit 4

- `@Ignore / @Ignore("Is disabled")` – ignores the unit test
- `@Test (expected = Exception.class)` – assesses if the tested method is throwing an exception
- `@Test(timeout=100)` – assesses if the unit test finishes before the timeout (used for Performance tests)

Test Suites – Junit 4

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestCase1.class,
    TestCase2.class
})
public class ExempluSuite {
    ...
}
```

Test Suites – Junit 4

```
@RunWith(Categories.class)
@IncludeCategory(NeedTest.class)
@Suite.SuiteClasses({
    Test2.class,
    Zest.class
})
public class TestSuitePartial {

}
```

Test Suites – Junit 4

@IncludeCategory(NeedTest.class) – marchează clasa sau metoda care sa fie inclusa in suite-ul parțial

@IncludeCategory({Type1.class, Type2.class, ...}) – poți adăuga mai multe categorii

@ExcludeCategory(NeedTest.class) – marchează clasa sau metoda care sa fie exclusă din suite-ul parțial

NeedTest.class – o clasa definite doar pentru a fi folosita la definirea unei categorii (clasa nu are o implementare specifica)

TestSuitePartial – nu trebuie sa includă metode @Test

Resurse JUnit4

- http://www.tutorialspoint.com/junit/junit_suite_test.htm
- <http://www.vogella.com/tutorials/JUnit/article.html>
- <http://junit.org/junit4/>
- <https://junit.org/junit5/>

Resurse recomandate

Andrew Hunt, David Thomas -
*Pragmatic Unit Testing in Java
with JUnit*, The Pragmatic
Programmers, 2004

