# Validarea sistemelor software.
# Tastarea White Box şi Black Box

## 1. Testarea sistemelor software. Testarea unitară

Definiție: Validarea software reprezintă procesul de detectare a erorilor în cadrul unui sistem software. Acest proces poate fi aplicat la mai multe nivele şi din mai multe perspective.

Scopul final al validării aplicațiilor software reprezintă detectarea a cât mai multe erori, cu precădere erori care pot produce probleme semnificative. Erorile detectate, împreună cu rapoartele aferente validării, vor fi utilizate ulterior pentru eliminarea, pe cât posibil, a erorilor.

Testarea software este o modalitate de validare a sistemelor software. În cadrul acestui laborator ne vom axa pe testarea automată a sistemelor software orientate pe obiecte, exemplificată ulterior în limbajul Java. Testarea automată presupune scrierea unor bucăți speciale de cod, numite baterii de teste. Acestea vor rula diverse bucăți din program şi vor compara rezultatele primite cu răspunsurile aşteptate.

Important de reținut: validarea este un proces continuu! Fiecare schimbare, cât de mică, presupune re-rularea întregii bateri de teste sau cel puțin a unui subset a acesteia.

În funcție de nivelul la care se face testarea şi de numărul componentelor invocate există mai multe tipuri de teste. În cadrul acestui laborator, ne interesează doar testarea unitară.

Definiție: Testarea unitară reprezintă testarea metodelor unei clase în izolare. Acest lucru presupune testarea metodelor fără dependența de un cod extern clasei. Dacă clasă conține obiecte de tipul unor alte clase (care nu fac parte din biblioteca standard sau dintr-o altă bibliotecă utilizată), acestea trebuie înlocuite cu obiecte Moc, care vor da de regulă răspunsuri hardcodate, considerate valide pentru program.

În funcție de modul de testare al metodelor, există două tipuri de testare: testare White Box şi testare Black Box. În primul caz, se va lua în considerare structura funcției, testându-se toate firele logice de parcurgere a metodei. În al doilea caz se ia în considerare doar semnătura funcției şi documentația acesteia.

Test case = un test (o funcție de testare), cu prerechizite, valori de testare şi rezultat aşteptat

Baterie de test = o mulțime de test case-uri, de regulă care testează același aspect sau funcționalitate

## 2. Testarea Black Box

Testarea de tip Black Box reprezintă testarea comportamentului codului executabil. Conceperea testelor va lua în considerare semnătura şi documentația funcției. Implementarea efectivă nu este importantă sau luată în considerare pentru conceperea sau scriere testelor. Cu alte cuvinte, funcția testată este tratată ca o cutie neagră. Se cunoaşte ce face şi interfața de comunicare, dar nu se cunosc detaliile de implementare.

### Aspecte importante de testare în testarea Black Box

În testarea Black Box trebuie să luăm în considerare:

1. structura de date pe care o manipulează, dacă se cunoaşte acest lucru şi schimbările suferite de aceasta după apelul funcției;
2. ce trebuie să facă funcția;
3. ce efecte laterale (secundare) are funcției.

În acest sens, ar fi bine să testăm că se respectă corect limitele datelor stocate şi corectitudinea acestora. De exemplu, dacă o metodă setează valoarea unui câmp, să verificăm că aceasta a fost setată corect. Dacă structura internă este de tip tablou sau mulțime, să verificăm că toate elementele sunt accesibile şi că nu se depăşesc accidental limitele.

De exemplu, dacă considerăm clasa ***ArrayList<T>***, aceasta reține o înșiruire de elemente. Dacă considerăm metoda ***T get(int index)***, când testăm această metodă trebuie să încercăm să obținem primul și ultimul elemente introdus, să testăm comportamentul pentru cazul în care lista e goală, dacă are un singur element etc. Poate ar fi o idee bună să verificăm că nu se schimbă dimensiunea listei de elemente după apelarea metodei. Dacă considerăm funcția ***boolean add(T x)***, ar trebui să testăm că elementul a fost introdus în listă pe ultima poziție, ce se întâmplă dacă lista inițială era goală, dacă avea un element sau dacă existau mai multe elemente inițial. Este important să verificăm actualizarea corectă a dimensiuni după adăugarea elementului etc.

Un aspect important de urmărit este cum influențează parametri de intrare valorile de ieșire și demersul funcției. Trebuie luate în considerare, dacă se cunoaște, posibilele condiții ale funcției pentru deciderea valorilor de testare.

Cel mai important aspect și cel mai complicat de stăpânit este generarea corespunzătoare de date de testare. Scopul final al testării, în special al testării Black Box, este generarea unui set de teste care să testeze la limită sistemul.

### Exemplu

În continuare va fi descris un posibil scenariu de testare a metodei

*boolean LinkedList::add(T elem)*

Pornind de la o lista goală:

adăugăm 1 element și testăm dacă s-a adăugat

```
LinkedList<Integer> testObj = new LinkedList<Integer>();
testObj.add(1);
assertEquals(1, testObj.get(0));
```

Adăugăm un al doilea element și testăm din nou. ! Aici folosim funcția get pentru a obține elementul. Dacă se poate evita utilizarea unei alte funcții, se preferă să se evite.

Putem testa și lungimea listei după adăugare fiecărui element pentru a verifica că se actualizează corect.

```
AssertEquals(1, testObj.size());
...
AssertEquals(2, testObj.size());
```

Pentru a testa funcția LinkedList<T>::get(T index), putem testa în felul următor:

| Valoarea inițială;<br>testObject = | Valorile testate pentru get;<br>index = | Răspuns așteptat |
|---|---|---|
| [] | get(0) | *IndexOutOfBoundsException* |
| [1] | get(0) | 1 |
| | get(1) | *IndexOutOfBoundsException* |
| [1,2] | get(0) | 1 |
| | get(1) | 2 |
| | get(2) | *IndexOutOfBoundsException* |
| [1,2,3,4,5,6] | get(-1) | *IndexOutOfBoundsException* |
| | get(6) | *IndexOutOfBoundsException* |
| | get(0) | 1 |

| | | |
|---|---|---|
| | get(3) | 4 |
| | get(5) | 6 |

Pentru a testa funcția LinkedList<T>::contains(Object o), putem testa în felul următor:

| Valoarea inițială;<br>testObject = | Valorile testate pentru get;<br>o = | Răspuns așteptat |
|---|---|---|
| [] | 1 | *false* |
| [1] | 1 | *true* |
| | 2 | *false* |
| [1,2,3,4,5,6,7] | 1 | *true* |
| | 7 | *true* |
| | 5 | *true* |
| | -4 | *false* |

Explicație: pentru o listă de lungime oarecare, ar fi bine să testăm dacă parcurge toate elementele atunci când caută. În acest sens, este important să verificăm că nu ratează primul și ultimul element. Ar fi bine să verificăm și un element din mijlocul listei, ca un „sanity check”. În final, testăm și pentru un element care nu există în listă pentru a verifica că nu returnează *true* tot timpul! Mai ar trebui verificat că returnează corect și dacă avem o listă goală și o listă de un element, acestea fiind niște cazuri excepționale. La nevoie, dacă se încearcă și un load test, se poate testa și dacă lista depășește un număr foarte mare de elemente, ex. pentru ArrayList care memorează lista ca un array și realocă la nevoie, ar fi bine să testăm pentru liste care depășesc 4MB de memorie (dimensiune maximă a unui segment de RAM), dacă folosim o arhitectura pe 32 de biti sau suportăm astfel de arhitecturi.

**Important!** Aici este exemplificată testarea unor funcții din biblioteca standard Java. În general aceste funcții se consideră corecte și în realitate vor fi testate cel mult de dezvoltatori JRE.

## 3. Testarea White Box

Testarea White Box (numită și Glass Box) reprezintă testarea structurii codului. Conceperea acestor teste va lua în considerare structura internă a codului, cu precădere structurile de control a fluxului de date, precum decizii, bucle, tratarea excepțiilor și apelarea altor funcții din interiorul funcțiilor testate. Ideea de bază în testarea White Box este testarea tuturor căilor de trecere printr-o funcție, adică testarea tuturor ramurilor de acces printr-o funcție și testarea corectitudinii buclelor de execuție printr-un un număr restrâns, dar satisfăcător, de cazuri.

### Metode generale indicate în testarea White Box

În continuare vor fi prezentate soluții generale pentru testarea structurii funcțiilor.

#### *I) Testarea deciziilor*

Pentru a testa corect și complet o structură de tip decizional, trebuie să existe teste care să parcurgă ambele ramuri ale unei structuri de tipul if/else sau toate ramurile unor structuri de tipul switch/case sau if/elif/else.

##### Exemplu

Fie următoarea funcție care returnează valoarea maximă dintre 2 numere:

```
int max(int a, int b) {
	if(a>b)
```

```
            return a;
        else
            return b;
    }
```

Se vor realiza minim 2 teste, de ex folosind perechile (1,2) şi (2,1). Ar fi indicat se va testa şi pentru 2 valori egale, ex. (1,1).

## *II) Testarea buclelor*

Testarea completă a buclelor presupune, în general, crearea unui număr de teste care să acopere următoarele 4 cazuri: să nu fie rulată niciodată bucla (să nu se intre în buclă), să fie rulată fix o dată, să fie rulată de numărul maxim de ori – dacă există şi minim un test care să ruleze bucla de un număr arbitrar de ori, dar fără includerea cazurilor anterioare – niciodată, o dată şi numărul maxim de ciclări.

În mod evident, aceste reguli generale trebuie adaptate la situaţiile particulare.

**Exemplu**

```
int sum(int[] inputVector) {
        int _sum = 0;

        for(int i=0; i<inputVector.length; i++)
            _sum += inputVector[i];

        return _sum;
    }
```

Funcţia sum va fi testată cu următoarele valori:

- [] – ce se întâmplă dacă dăm un vector gol
- [1] – ce se întâmplă dacă dăm un tablou cu un element
- [1,2,3,4] – ce se întâmplă dacă dăm un tablou cu mai multe elemente

```
boolean exist(int[] arr, int elem) {
        for(int i = 0; i < arr.length; i++)
            if(arr[i] == elem)
                return true;
        return false;
    }
```

Se va testa cu următoarele valori:

| arr | elem | Explicaţie |
|---|---|---|
| [] | 2 | Bucla nu este rulată niciodată |
| [1] | 1 | Bucla este rulată o dată, elementul este găsit |
| [1] | 2 | Bucla este rulată până la capăt, nu se găsesc elemente |
| [1,2,3,4] | 1 | Elementul este pe prima poziţie, se face un pas în buclă |
| [1,2,3,4] | 2 | Elementul este pe a doua poziţie, se fac un număr aleator de paşi în buclă |
| [1,2,3,4] | 4 | Elementul este pe ultima poziţie, se fac numărul maxim de paşi în buclă |
| [1,2,3,4] | 6 | Elementul nu este prezent, se fac numărul maxim de paşi în buclă şi se iasă din buclă |

### *III) Testarea condițiilor*

Condițiile sunt utilizate atât în cazul structurilor decizionale, cât și în cazul multor bucle. Dacă condițiile sunt simple, testarea lor este evidentă. Probleme pot apărea în cazul condițiilor compuse. Condițiile compuse sunt acele condiții alcătuite din mai multe condiții simple, legate între ele prin operații logice de tipul ȘI, SAU și SAU-EXCLUSIV. Ideea de bază în testarea condițiilor este verificarea că fiecare condiție simplă din condiția compusă influențează răspunsul, și că îl influențează corect.

| Operație | Negare | Și | Sau | Sau exclusiv |
|---|---|---|---|---|
| Notație matematică | ¬ | ∧ | ∨ | $\underline{\vee}$ |
| Simboluri utilizat de regulă în programare | !, ~, not | &, &&, and | \|, \|\|, or | ^, xor |

Pentru simplicitate, în acest laborator vor fi utilizate simbolurile !, &, | și ^, o variantă simplificată a notațiilor folosite din limbaje de programare. **Atenție! Vorbim de operații logice, nu pe biți!**

**Exemplu**

Pentru expresia a & b | c, o validare completă a expresiei este cea prezentată mai jos. Rândurile cu scris îngroșat și italic reprezintă testele minimale care ar trebui să verifice expresia.

| a | b | c | a & b \| c |
|---|---|---|---|
| ***F*** | ***F*** | ***F*** | ***F*** |
| ***T*** | ***F*** | ***F*** | ***F*** |
| ***F*** | ***T*** | ***F*** | ***F*** |
| ***T*** | ***T*** | ***F*** | ***T*** |
| ***F*** | ***F*** | ***T*** | ***T*** |
| F | T | T | T |
| T | F | T | T |
| T | T | T | T |

## 4. White Box vs Black Box

Având în vedere cele două metode de concepere a testelor, vom face o comparație a acestora pentru a aprofunda subiectul.

| **Testare White Box** | **Testare Black Box** |
|---|---|
| Dezvoltarea testelor se bazează pe structura internă a funcției | Dezvoltarea testelor se bazează pe semnătură și specificații |
| Testează mai mult structura | Testează comportamentul |
| | Testele pot fi dezvoltate independent de implementare |

Ambele metode de testare sunt utile în validarea software.

## 5. Code Coverage și eficiență

Code coverage – metrică folosită pentru a vedea numărul sau procentul de instrucțiuni dintr-un program parcurse. Poate fi folosită pentru a vedea cantitatea de cod acoperită de teste. Este un factor care nu poate fi ignorat în testare, dar nu e un factor decisiv de obicei în deciderea continuării testării sau direcției în care testarea trebuie să aibă loc.

Eficiența unei baterii de teste reprezintă probabilitatea ca acestea să descopere greșeli. Eficiența este un concept și nu poate fi măsurat, ca și coverage-ul. Acesta poate fi apreciat în funcție de calitatea testelor și scopul acestora.

O baterie de teste bună este determinată de eficiența acesteia, și nu de coverage, sau cel puțin nu doar de cel din urmă. Cu alte cuvinte, degeaba o baterie de teste are un coverage bun dacă nu identifică probleme în cod, sau probabilitatea de a identifica este foarte mică. Coverage-ul poate indica unde ar mai fi nevoie de testare, dar decizia revine testorului pentru a aprecia dacă efortul se merită. Uneori un cod acoperit 100% de teste este cel care are nevoie de mai multe teste!

## 6. Exemplu de framework de testare: JUnit 5

Scrierea de teste folosind Junit presupune crearea unei clase care conține o mulțime de funcții adnotate corespunzător. Există 3 adnotări diferite pe care le vom utiliza în cadrul acestui laborator, JUnit având definite și altele. Pentru a rula testele va trebui să rulăm clasa de test.

### Adnotările

- @Test – folosit pentru a marca funcțiile care sunt folosite ca teste. Pot exista mai multe funcții de test
- @BeforeEach – o funcție care va fi rulată înainte de fiecare test. Folosită pentru a inițializa obiectele utilizate pentru efectuarea următorului test.
- @AfterEach – o funcție care va fi rulată după fiecare test. Folosită pentru a șterge obiectele utilizate în testul efectuat.

### Testarea valorilor

Pentru a testa valorile returnate de o funcție există o serie de metode statice oferite de JUnit.

- assertEquals(expected, given) – verifică dacă valoare returnată este egală cu cea așteptată. Este supraîncărcată pentru toate tipurile de date primitive și pentru *Object*.
- assertTrue(given) – verifică dacă valoare returnată este booleanul adevărat (*true*).
- assertFalse(given) – verifică dacă valoare returnată este booleanul fals (*false*).
- assertNull(given) – verifică dacă valoare returnată este null.
- assertNotNull(given) – verifică dacă valoare returnată este diferită de null.

### Testarea excepțiilor

- assertThrows(expectedExceptionType, executabil) – se va verifica dacă se aruncă excepția dorită. Parametru *expectedExceptionType* este un obiect de tipul clasei *Class* și va fi obținut din câmpul static *class* al clasei excepției așteptate. Parametrul *executabil* va fi un apel de

funcție sau o funcție lambda. A se vedea exemplu, test case-ul *testm1* pentru clarificare. Returnează referința către excepția aruncată, care poate fi utilizată pentru a verifica mesajul transmis, dacă se dorește acest lucru.

- assertDoesNotThrow(executabil) – va verifica dacă nu s-a aruncat o excepție. Va returna valoarea returnată de executabil, dacă există.
- metoda clasică de testare folosind try-catch (a se vedea test case-ul *testThrowsClaasic_n1* din exemplu) a – se va scrie un bloc try-catch, de regulă cu 1-2 ramuri de catch, și se va utiliza metoda statică fail() în punctele din program unde nu ar trebui să se ajungă.

### Alte funcții utile

- fail() – termina test case-ul cu eroare

### Evaluarea răspunsurilor de tip tablou

Pentru verificarea egalității a două tablouri se poate folosi metoda statică equals a clasei Arrays. Aceasta va retura un boolean, care poate fi dat ca parametru metodei assertTrue sau assertFalse.

```
assertTrue(Arrays.equals(new int[] {1,2,3}, new int[] {1,2,3}));
```

### Precizări suplimentare

Metodele prezentate cuprind principalele metode și adnotări studiate la laborator. Mai există și altele care nu au fost cuprinse. Se va verifica documentația atașată în secțiunea Bibliografie dacă se dorește consultarea acestora.

**!Atenție!** Toate metodele menționate anterior sunt supraîncărcate pentru a accepta și un parametru mesaj, de tipul String, ca ultim parametru. Acest mesaj este un mesaj afișat în caz că testul a eșuat. Se pot transmite mesaje sugestive cu cauza eșecului.

### Exemplu

*Exemplul dat conține următoarele structuri, tipuri și mecanici de limbaj din Java: boxing/unboxing, funcții lambda, clasa LinkedList, adnotatori, import static.*

În exemplul următro, sunt oferite câteva exemplu de scriere de test case-uri pentru testarea metodei get(int index) a clasei LinkedList.

```
package test;

import static org.junit.jupiter.api.Assertions.*;

import java.util.LinkedList;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class TestCase {

	LinkedList<Integer> testObj;

	final int n = 5;

	@BeforeEach
	void init() {
		testObj = new LinkedList<Integer>();

		for(var item=0; item < n; item++) {
			testObj.add(item);
		}
	}

	@AfterEach
	void destroy() {
		testObj = null;
	}

	@Test
	void test0() {
		assertEquals(0, testObj.get(0));
	}


	@Test
	void testnm1() {
		assertEquals(n-1, testObj.get(n-1));
	}

	@Test
	void testm1() {
		assertThrows(IndexOutOfBoundsException.class, () -> testObj.get(-1));
	}

	@Test
	void test2() {
		int x = assertDoesNotThrow(() -> testObj.get(2));
		assertEquals(2,x);
	}
```

```
	@Test
	void testThrowsClaasic_n1() {
		try {
			testObj.get(n);
			fail();
		}
		catch(IndexOutOfBoundsException e) {
			//daca am ajuns aici e bine
			return;
		}
		fail(); //daca am ajuns inseammna ca nu s-a arunca exceptia dorita
	}
}
```

## 7. Sugestie pentru C#: MSTest

Pentru testarea proiectelor C#, puteți folosi biblioteca MSTest[4]. Aceasta seamănă suficient cu biblioteca JUnit.

Clasele care conțin testele vor vi adnotate cu [TestClass].

Metodele de instanțiere și de curățare se vor adnota cu [TestInitialize] și [TestCleanup]

Pentru metodele de aserțiune, citiți [5]. Pe scurt, aveți metode precum AreEqual, IsTrue, IsFalse, IsNull, IsNotNull, ThrowException și Fail.

## 8. Bibliografie


[1] junit.org – documentație JUnit 5, introducere https://junit.org/junit5/docs/current/user-guide/

[2] junit.org – documentație aserțiune: https://junit.org/junit5/docs/5.8.2/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html

[3] laborator vechi FIS https://sites.google.com/site/codrutaistin/laborator-fis-8

[4] MSTest: https://learn.microsoft.com/en-us/visualstudio/test/using-microsoft-visualstudio-testtools-unittesting-members-in-unit-tests?view=vs-2022

[5] MSTest – documentație aserțiune: https://learn.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert?view=visualstudiosdk-2022