



UNIVERSITATEA
DIN BUCUREȘTI

Metode de dezvoltare software

Testare

29.04.2020

Alin Ștefănescu

A black laptop is shown from a front-facing perspective, slightly angled to the right. The screen is white and displays the text 'Testare software' in a bold, dark gray sans-serif font. The laptop's keyboard and trackpad are visible below the screen. The laptop is set against a plain white background with a soft shadow underneath.

Testare software


Prezentare bazată pe materiale de Florentin Ipate (UniBuc) și Florin Leon (UT Iași)

Testare... în practică




Generalități - validare și verificare (V&V)

Verificare

- 
- construim corect produsul?
 - se referă la dezvoltarea produsului

Validare

- 
- construim produsul corect?
 - se referă la respectarea specificațiilor, la utilitatea produsului

- Verificarea și validarea trebuie să stabilească încrederea că produsul este potrivit pentru scopul său
- Aceasta **nu** înseamnă că produsul este lipsit de defecte
- Doar că produsul trebuie să fie suficient de bun pentru utilizare

Evaluarea unui produs software

Depinde de scopul produsului, de așteptările utilizatorilor și factorii de piață:

■ Funcționalitatea programului

- nivelul de încredere depinde de cât de critic este sistemul pentru utilizatori

■ Așteptările utilizatorilor

- utilizatorii pot avea grade diferite de așteptări pentru diferite tipuri de produse software

■ Mediul de afaceri

- lansarea rapidă pe piață a unui produs poate fi uneori mai importantă decât găsirea tuturor defectelor în program

Testarea unui program

- evidențiază prezența erorilor și *nu* absența lor
- este singura tehnică de validare pentru cerințe non-funcționale, deoarece programul trebuie executat pentru a i se analiza comportamentul
- este utilizată de obicei alături de verificarea statică (*static analysis*) pentru o siguranță cât mai mare

Terminologie (IEEE)

■ Eroare (engl. “*error*”)

- o acțiune umană care are ca rezultat un defect în produsul software

■ Defect (engl. “*fault*”)

- consecința unei erori în produsul software
- un defect poate fi latent: nu cauzează probleme cât timp nu apar condițiile care determină execuția anumitor linii de cod

■ Defecțiune (engl. “*failure*”)

- manifestarea unui defect: când execuția programului întâlnește un defect, acesta provoacă o defecțiune
- abaterea programului de la comportamentul așteptat

Bug

“Bug”: termen colocvial utilizat deseori ca sinonim pentru „defect”



de la dilbert.com

Testarea și depanarea

■ testarea de validare

- intenționează să arate că produsul nu îndeplinește cerințele
- testele încearcă să arate că o cerință nu a fost implementată adecvat

■ testarea defectelor

- teste proiectate să descopere prezența defectelor în sistem
- testele încearcă să descopere defecte

■ depanarea (“*debugging*”)

- are ca scop localizarea și repararea erorilor corespunzătoare
- implică formularea unor ipoteze asupra comportamentului programului, corectarea defectelor și apoi re-testarea programului

Asigurarea calității

- testarea se referă la detectarea defectelor
- **asigurarea calității** se referă la prevenirea lor
 - se ocupă de procesele de dezvoltare care să conducă la producerea unui software de calitate
 - include procesul de testare a produsului

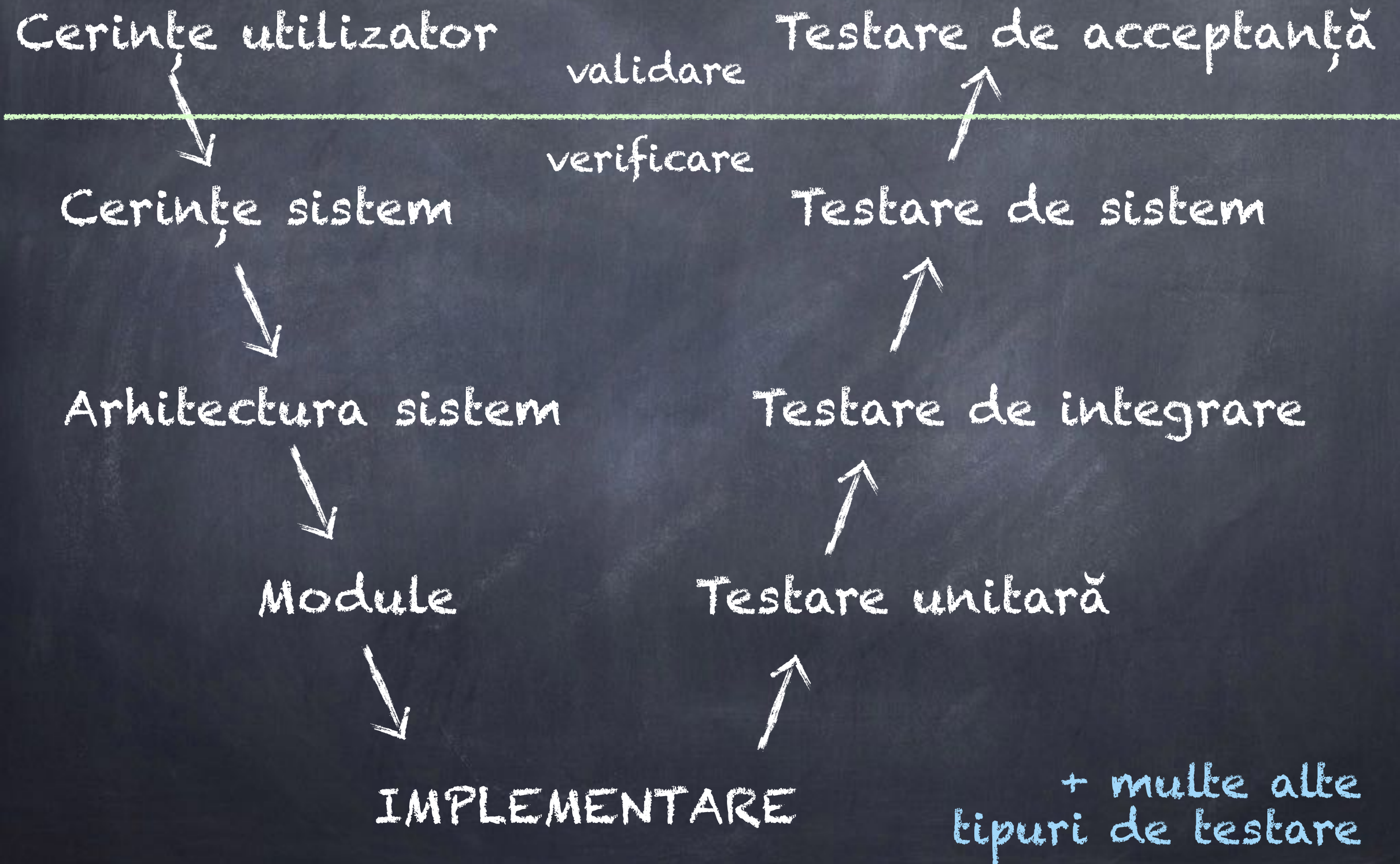
Câteva principii de testare

- o parte necesară a unui caz de test este definirea **ieșirii** sau **rezultatului** așteptat
- programatorii nu ar trebui să-și testeze propriile programe (**excepție** face testarea de nivel foarte jos - testarea unitară)
- ideal, organizațiile ar trebui să folosească și companii (sau departamente) externe pentru testarea propriilor programe
- rezultatele testelor trebuie analizate amănunțit
- trebuie scrise cazuri de test atât pentru condiții de intrare invalide și neașteptate, cât și pentru condiții de intrare valide și așteptate

Câteva principii de testare (continuare)

- programul trebuie examinat pentru a vedea dacă nu face ce trebuie; de asemenea, trebuie examinat pentru a vedea dacă nu cumva face ceva ce nu trebuie
- pe cât posibil, cazurile de test trebuie salvate și re-executate după efectuarea unor modificari
- probabilitatea ca mai multe erori să existe într-o secțiune a programului este proporțională cu numărul de erori deja descoperite în acea secțiune
- efortul de testare nu trebuie subapreciat
- creativitatea necesară procesului de testare nu trebuie subapreciată

Modelul V



Testarea unitară (unit testing)

- o unitate (sau un modul) se referă de obicei la un element atomic (clasă sau funcție), dar poate însemna și un element de nivel mai înalt: bibliotecă, driver etc.
- testarea unei unități se face în izolare
 - pentru simularea apelurilor externe se pot utiliza funcții externe fictive (engl. “stubs”)

Testarea de integrare (integration testing)

- Testează interacțiunea mai multor unități
- Testarea este determinată de arhitectură

Testarea sistemului (system testing)

- testarea sistemului testează aplicația ca întreg și este determinată de scenariile de analiză
- aplicația trebuie să execute cu succes toate scenariile pentru a putea fi livrată clientului
- spre deosebire de testarea internă și a componentelor, care se face prin program, testarea aplicației se face de obicei cu script-uri care rulează sistemul cu o serie de parametri și colectează rezultatele
- testarea aplicației trebuie să fie realizată de o echipă independentă de echipa de implementare
- testele se bazează pe specificațiile sistemului

Testarea de acceptanță (acceptance testing)

- testele de acceptanță determină dacă sunt îndeplinite cerințele unei specificații sau ale contractului cu clientul.
- sunt de diferite tipuri:
 - teste rulate de dezvoltator înainte de a livra produsul software
 - teste rulate de utilizator (*user acceptance testing*)
 - teste de operaționalitate (*operational testing*)
 - testare alfa și beta: *alfa* la dezvoltator, *beta* la client cu un grup ales de utilizatori

Testele de regresie (regression testing)

- un test valid generează un set de rezultate verificate, numit “standardul de aur”
- testele de regresie sunt utilizate la **re-testare**, după realizarea unor modificări, pentru a asigura faptul că modificările nu au introdus noi defecte în codul care funcționa bine anterior
- pe măsură ce dezvoltarea continuă, sunt adăugate alte teste noi, iar testele vechi pot rămâne valide sau nu
- dacă un test vechi nu mai este valid, rezultatele sale sunt modificate în standardul de aur
- acest mecanism previne *regresia* sistemului într-o stare de eroare anterioară

Testarea performanței (performance testing)

- O parte din testare se concentrează pe evaluarea proprietăților non-funcționale ale sistemului, cum ar fi:
 - siguranța (“reliability”) - menținerea unui nivel specificat de performanță
 - securitatea - persoanele neautorizate să nu aibă acces, iar celor autorizate să nu le fie refuzat accesul
 - utilizabilitatea - capacitatea de a fi înțeles, învățat și utilizat
 - load & stress testing (v. următoarele două slide-uri)

Testarea la încărcare (load testing)

- asigură faptul că sistemul **poate gestiona un volum așteptat de date**, similar cu acela din locația-destinație (de exemplu la client)
- verifică eficiența sistemului și modul în care scalează acesta pentru un **mediu real** de execuție

Testarea la stres (stress testing)

- solicită sistemul **dincolo de încărcarea maximă** proiectată
- supraîncărcarea testează modul în care „cade” sistemul
 - sistemele nu trebuie să eșueze catastrofal
 - testarea la stres verifică pierderile inacceptabile de date sau funcționalități
- deseori apar aici conflicte între teste. Fiecare test funcționează corect atunci când este făcut separat. Când două teste sunt rulate în paralel, unul sau ambele teste pot eșua
 - cauza este de obicei managementul incorect al accesului la resurse critice (de exemplu, memoria)
- o altă variantă, “soak testing”, presupune rularea sistemului pentru o perioadă lungă de timp (zile, săptămâni, luni)
 - în acest caz, de exemplu scurgerile nesemnificative de memorie se pot acumula și pot provoca căderea sistemului

Testarea interfeței cu utilizatorul (GUI testing)

- majoritatea aplicațiilor din zilele noastre au interfețe grafice cu utilizatorul (GUI)
 - testarea interfeței grafice poate pune anumite probleme
 - cele mai multe interfețe, dacă nu chiar toate, au bucle de evenimente, care conțin cozi de mesaje de la mouse, tastatură, ferestre, touchscreen etc.
 - asociate cu fiecare eveniment sunt coordonatele ecranului
 - testarea interfeței cu utilizatorul presupune memorarea tuturor acestor informații și elaborarea unei modalități prin care mesajele să fie trimise din nou aplicației, la un moment ulterior
- de obicei se folosesc scripturi pentru testare sau frameworkuri precum Selenium (sau se testează manual)

Testarea utilizabilității (usability testing)

- testează cât de ușor de folosit este sistemul
- se poate face în laboratoare sau „pe teren” cu utilizatori din lumea reală
- exemple de metode folosite:
 - testare “pe hol” (hallway testing): cu câțiva utilizatori aleatori
 - testare de la distanță: analizarea logurilor utilizatorilor (dacă își dau acordul pentru aceasta)
 - recenzii ale unor experți (externi)
 - A/B testing: în special pentru web design, modificarea unui singur element din UI (d.ex. culoarea sau poziția unui buton) și verificarea comportamentului unui grup de utilizatori



**testare
de tip
cutie neagră**

Testarea de tip “cutie neagră”

- testarea exhaustivă nu este fezabilă
- generarea aleatorie a cazurilor de test nu este eficientă
- posibilă soluție: se iau în considerare numai intrările (într-un modul, componentă sau sistem) și ieșirile dorite, conform specificațiilor
 - structura internă este ignorată (de unde și numele de “**black box testing**”)
 - deoarece se bazează pe funcționalitatea descrisă în specificații, se mai numește și **testare funcțională**
 - poate fi folosită în principiu la orice nivel de testare (unitară, integrare, sistem)

Observații

- datele de test sunt generate pe baza specificației (cerințelor) programului, structura programului ne jucând nici un rol
- tipul de specificație ideal pentru testarea “cutie neagră” este alcătuit din pre-condiții și post-condiții
- exemple de metode de testare de tip “cutie neagră”:
 - partiționare în clase de echivalență
 - analiza valorilor de frontieră
 - partiționarea în categorii
 - graful cauză-efect
 - testarea tuturor perechilor
 - testarea bazată pe modele
 - etc.



**testare
de tip
cutie albă**

Testarea de tip “cutie albă” (white-box testing)

- testarea „cutie neagră” tratează funcționalitatea unui modul luând în calcul doar intrările și ieșirile și relațiile dintre ele definite în cerințe
- **testarea de tip „cutie albă”** ia în calcul **codul sursă** al metodelor testate
- testarea „cutie albă” vizează acoperirea diferitelor structuri ale programului
 - de aceea se mai numește și **„testare structurală”**
- în practică se folosește de multe ori o combinație între testarea de tip “cutie albă” și “cutie neagră”, numită “cutie gri”

Structura programului ca graf

- datele de test sunt generate pe baza implementării (programului), fără a lua în considerare specificația (cerințele) programului
- pentru a utiliza structura programului, acesta poate fi reprezentat sub forma unui **graf orientat**
- datele de test sunt alese astfel încât să parcurgă toate elementele grafului (instrucțiune, ramură sau cale) măcar o singură dată.

Acoperiri pentru testare cutie alba

Pe baza grafului se pot defini diverse **acoperiri**:

- **acoperire la nivel de instrucțiune**: fiecare instrucțiune (sau nod al grafului) este parcursă măcar o dată
- **acoperire la nivel de ramură**: fiecare ramură a grafului este parcursă măcar o dată
- **acoperire la nivel de cale**: fiecare cale din graf este parcursă măcar o dată
- și alte variante

Alte tipuri de testare

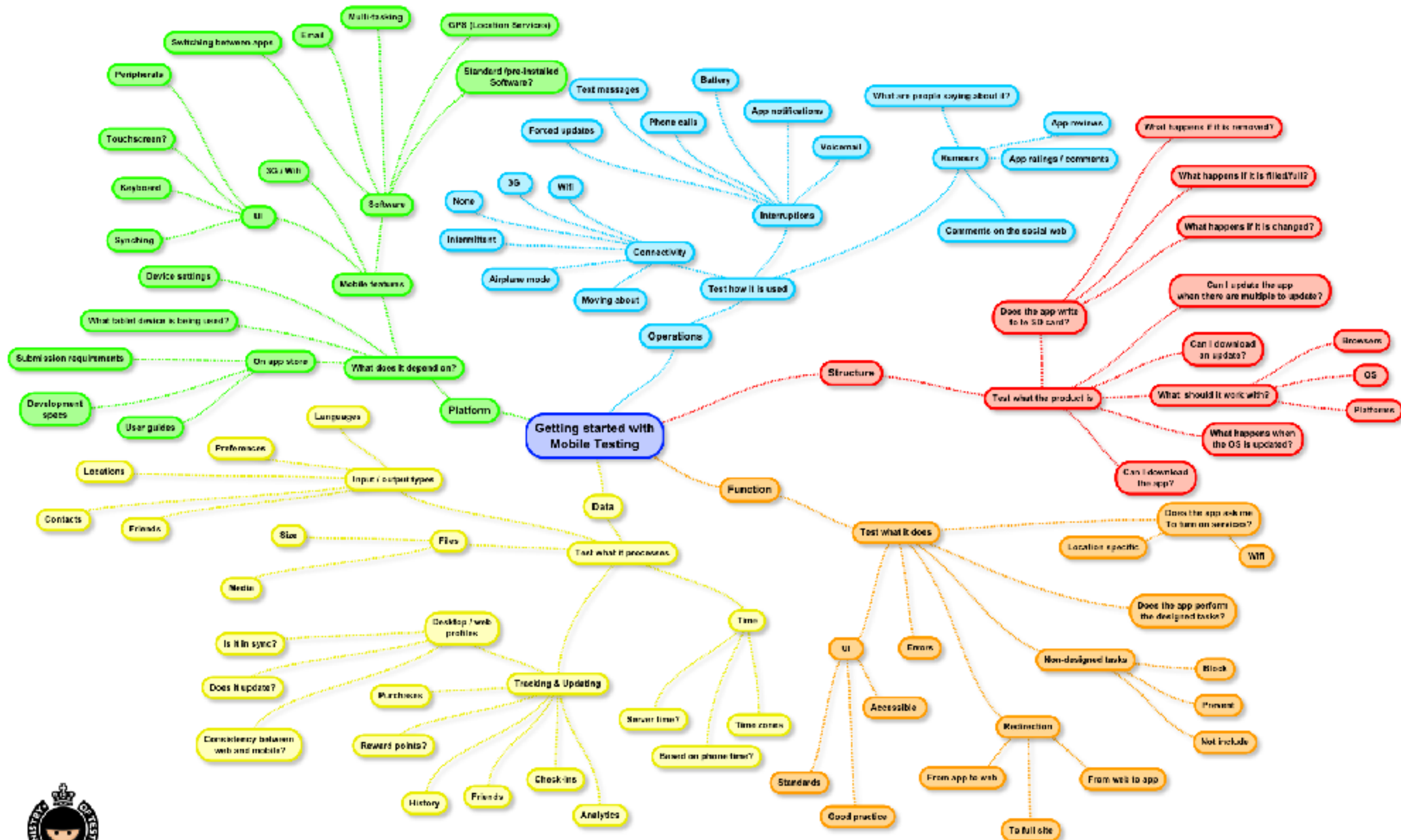


Alte tipuri de teste

Există bineînțeles multe alte aspecte și tipuri de testare

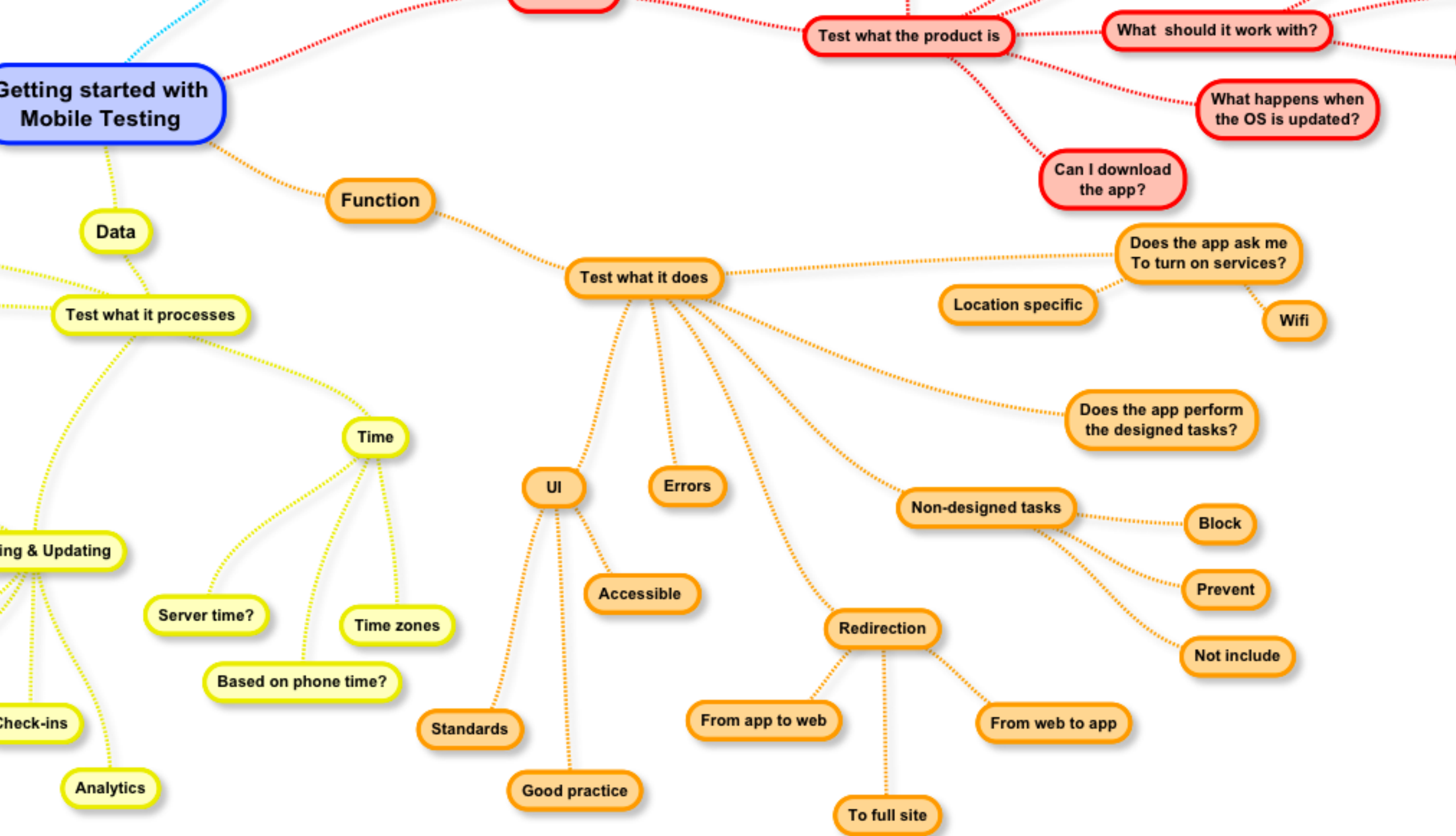
- testarea claselor
- înregistrarea și urmărirea defectelor
- automatizarea testării
 - testarea bazată pe modele
 - generarea de teste
- tooluri de acoperire (coverage tools)
- testare pentru domenii specifice:
 - pentru aplicații web (d.ex. Selenium)
 - pentru aplicații mobile (v. http://en.wikipedia.org/wiki/Mobile_application_testing, d.ex. Appium)
 - pentru jocuri (d.ex. Unity test framework)
- în practică: **testarea e importantă și consumă multe resurse**

Aspecte ale testării aplicațiilor mobile

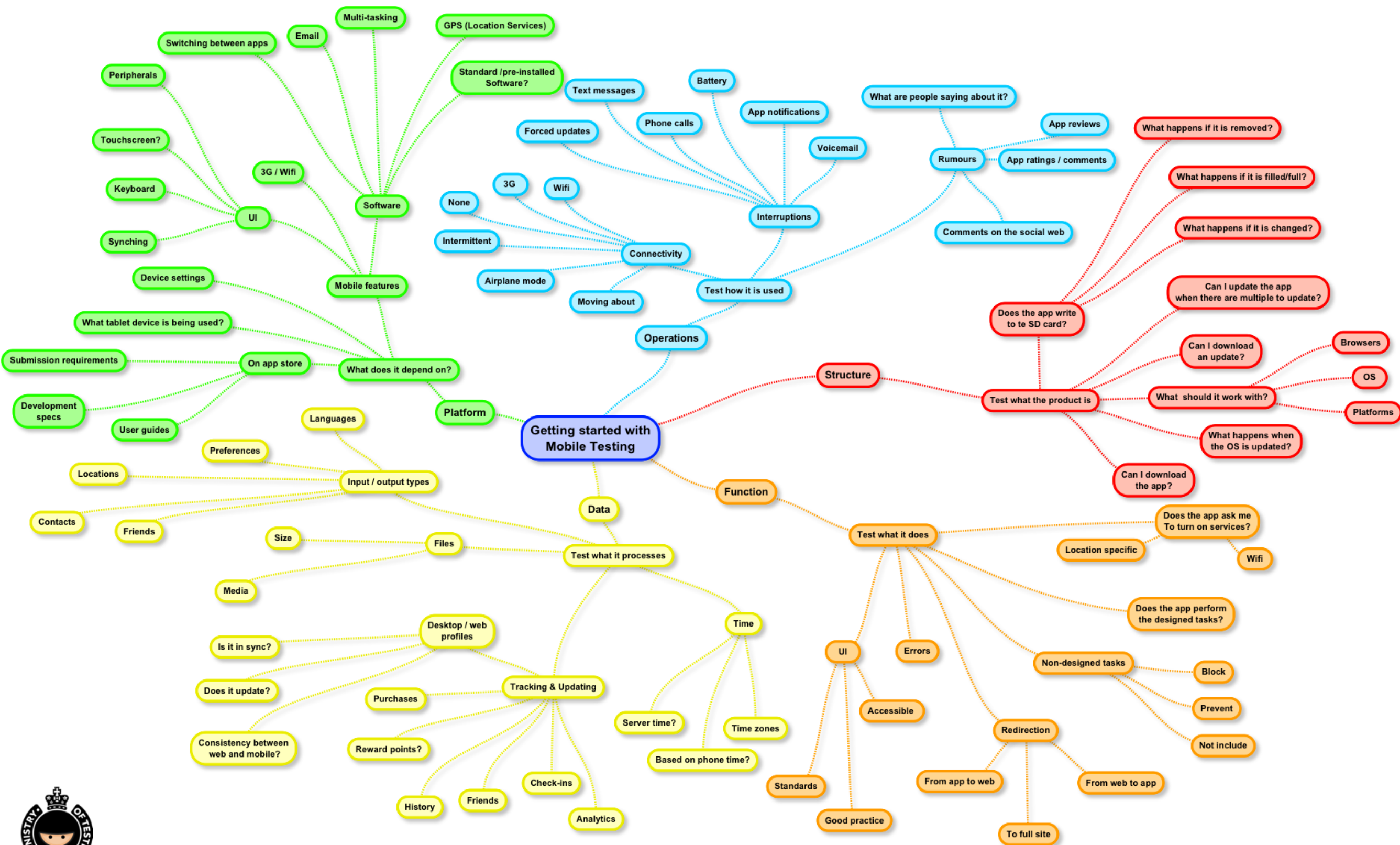










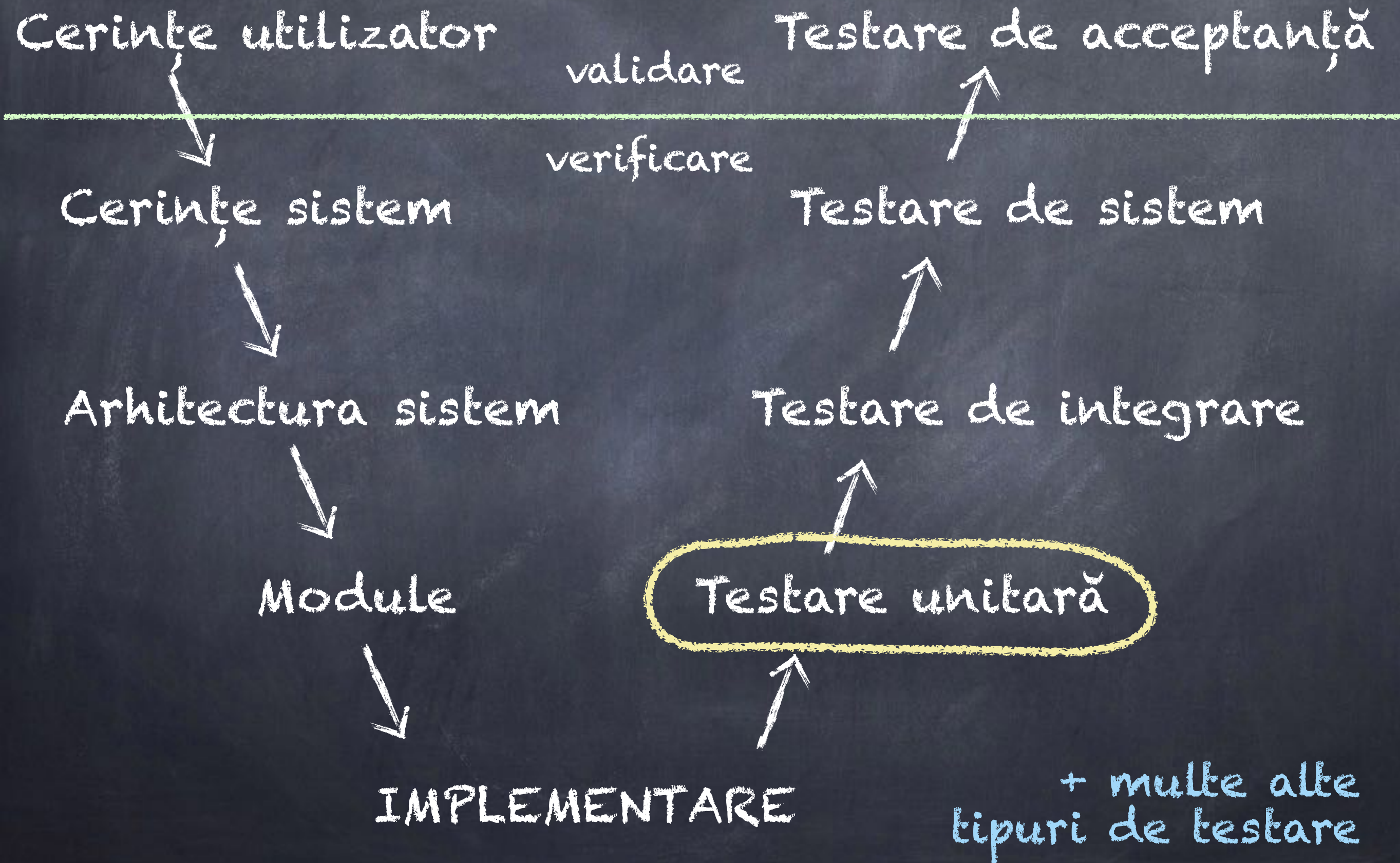


A black laptop is shown from a front-facing perspective, slightly angled to the right. The screen is white and displays the title 'Testare unitară cu JUnit' in a bold, dark gray sans-serif font. The laptop's keyboard and trackpad are visible below the screen.

Testare unitară cu JUnit

Prezentare bazată pe materiale de
M. Johansson, W. Ahrendt, V. Klebanov (Chalmers University)

Modelul V



Testarea unitară (unit testing)

- o unitate (sau un modul) se referă de obicei la un element atomic. În particular, testarea unei unități = testarea unei proceduri, iar în programarea orientată pe obiecte a unei metode
- un **test** conține
 - inițializarea (clasei sau a argumentelor necesare)
 - apelul metodei testate
 - decizia (**oracolul**) dacă testul **a reușit** sau **a eșuat**
 - ➔ acesta e foarte important pentru evaluarea automată a testului
 - ➔ compară valorile produse de metodă cu cele corecte
- o **suită de teste** este o colecție de teste

JUnit

- exemplificăm testarea unitară cu **JUnit**
- JUnit este un tool simplu, dar foarte popular, care oferă:
 - funcționalitatea necesară pentru execuția repetată a scrierii de teste pentru **Java**
 - un mod de a adnota metode ca fiind teste
 - un mod de a executa și evalua suite de teste
 - poate fi rulată atât în linie de comandă sau integrată într-un IDE (d.ex. majoritatea IDE-urilor permit folosirea JUnit)
 - versiunea curentă este JUnit 5 - <https://junit.org/junit5> (însă puteți folosi și JUnit 4 - versiunea folosită în următoarele slideuri)

Un prim exemplu

```
public class Ex1 {  
  
    // requires: a is non-null, non-empty  
    // ensures:  result is equal to a minimal element in a  
    public static int find_min(int[] a) {  
        int x, i;  
        x = a[0];  
        for (i = 1; i < a.length; i++) {  
            if (a[i] < x)  
                x = a[i];  
        }  
        return x;  
    }  
}
```

...

Un prim exemplu ... continuat

... continuare clasa Ex1:

```
// requires: x is non-null and sorted in increasing order
// ensures:  result is sorted and contains
//           the elements in x and n, but no others
public static int[] insert(int[] x, int n) {
    int[] y = new int[x.length + 1];
    int i;
    for (i = 0; i < x.length; i++) {
        if (n < x[i]) break;
        y[i] = x[i];
    }
    y[i] = n;
    for (; i < x.length; i++) {
        y[i+1] = x[i];
    }
    return y;
}
```

Un prim exemplu ... testat

JUnit poate să testeze **valorile returnate așteptate**:

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

public class Ex1Test {
    @Test public void test_find_min_1() {
        int[] a = {5, 1, 7};
        int res = Ex1.find_min(a);
        assertTrue(res == 1);
    }

    @Test public void test_insert_1() {
        int[] x = {2, 7};
        int n = 6;
        int[] res = Ex1.insert(x, n);
        int[] expected = {2, 6, 7};
        assertTrue(Arrays.equals(expected, res));
    }
}
```

Execuție în consolă

```
>
>
> javac Ex1Test.java
>
> java org.junit.runner.JUnitCore Ex1Test
JUnit version 4.11
Time: 0.005
OK (2 tests)
>
>
```

Testarea excepțiilor

JUnit poate să testeze diverse alte aspecte, cum ar fi **excepțiile**:

```
@Test(expected = IndexOutOfBoundsException.class)
public void outOfBounds() {
    new ArrayList<Object>().get (1);
}
```

expected declară că outOfBounds trebuie să arunce o excepție de tip IndexOutOfBoundsException.

Dacă metoda outOfBounds aruncă o altă excepție sau nu aruncă nici una, testul eșuează.

Exemplul anterior în GitHub și GitLab

Am încărcat exemplul anterior în **GitHub** la:

`https://github.com/alinstef/junitexample`

În README.md am descris tot procesul de configurare pentru a rula testele folosind Gradle (inclusiv cu GitHub Actions, după ce proiectul a fost încărcat în GitHub)

De asemenea, am încărcat exemplul și în **GitLab** la:

`https://gitlab.com/alinstef/junitexample`

În README.md am descris tot procesul de configurare pentru a rula testele folosind Gradle (inclusiv cu GitLab CI/CD, după ce proiectul a fost încărcat în GitLab)

Al doilea exemplu... test-driven development

În test-driven development, se scrie mai întâi testul și apoi implementarea:

```
class Money {
    public int amount;
    private Currency currency;

    public Money(int amount, Currency currency) {
        this.amount = amount;
        this.currency = currency;
    }

    public Money add(Money m) {
        // NEIMPLEMENTATĂ ÎNCĂ, DE SCRIS TESTUL PRIMA DATĂ
    }
}

class Currency {
    private String name;
    public Currency(String name) {
        this.name = name;
    }
}
```

Se scrie un test pentru `Money.add()`

În test-driven development, se scrie **mai întâi testul și apoi implementarea**:

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

public class MoneyTest {

    @Test public void simpleAdd() {
        Currency ron = new Currency("RON");
        Money m1 = new Money(120, ron);
        Money m2 = new Money(160, ron);
        Money result = m1.add(m2);
        Money expected = new Money(280, ron);
        assertTrue(expected.equals(result));
    }
}
```

Exemplul Money

Acum se implementează metoda, dar prima oară ne asigurăm ca testul eșuează (pentru a fi siguri ca nu cumva testul să aibă succes în orice condiții)

```
class Money {  
    public int amount;  
    private Currency currency;  
  
    ...  
  
    public Money add(Money m) {  
        return null;  
    }  
}
```

Exemplul Money

După ce testul eșuează, implementăm metoda ca mai jos:

```
class Money {  
    public int amount;  
    private Currency currency;  
  
    ...  
  
    public Money add(Money m) {  
        return new Money (amount + m.amount, currency);  
    }  
}
```

Verificăm din nou, dar testul eșuează din nou.

Problema: equals pentru obiecte

Exemplul Money... completat

```
class Money {  
    public int amount;  
    private Currency currency;  
  
    public Money(int amount, Currency currency) {  
        this.amount = amount;  
        this.currency = currency;  
    }  
  
    public Money add(Money m) {  
        return new Money (amount + m.amount, currency);  
    }  
  
    public boolean equals(Object o) {  
        if (o instanceof Money) {  
            Money other = (Money)o;  
            return (currency == other.currency  
                    && amount == other.amount);  
        }  
        return false;  
    }  
}
```

Încă o **problemă**: ce se întâmplă când valuta e diferită?

Se scrie un test pentru Money.add()

Extindem clasa Money cu rata de schimb Euro, dar **prima oară în test**

```
public class MoneyTest {

    @Test public void simpleAdd() {
        Currency ron = new Currency("RON", 4.8);
        Money m1 = new Money(120, ron);
        Money m2 = new Money(140, ron);
        Money result = m1.add(m2);
        Money expected = new Money(260, ron);
        assertTrue(expected.equals(result));
    }

    @Test public void addDifferentCurrency() {
        Currency ron = new Currency("RON", 4.8);
        Money m1 = new Money(120, ron);
        Currency usd = new Currency("USD", 1.2);
        Money m2 = new Money(150, usd);
        Money result = m1.add(m2);
        Money expected = new Money(720, ron);
        assertTrue(expected.equals(result));
    }
}
```

Modificăm în implementare acum:

Exemplul Money... final

```
class Money {
    public int amount;
    private Currency currency;

    public Money(int amount, Currency currency) {
        this.amount = amount;
        this.currency = currency;
    }

    public Money add(Money m) {
        return new Money(amount +
            (int)(m.inEuro()*currency.rate()), currency);
    }

    public double inEuro() {
        return ((double) amount)/currency.rate();
    }

    public boolean equals(Object o) {
        if (o instanceof Money) {
            Money other = (Money)o;
            return currency == other.currency
                && amount == other.amount;
        }
        return false;
    }
}
```

```
class Currency {

    private String name;
    private double euroRate;

    public Currency(String name,
        double euroRate) {
        this.name = name;
        this.euroRate = euroRate;
    }

    public String name() {
        return name;
    }

    public double rate(){
        return euroRate;
    }
}
```

Testele trec cu **succes**.

Preambulul setUp()

Anumite părți comune pot fi puse în preambulul testelor (@Before este executat înaintea fiecărui test).

```
public class MoneyTest {  
  
    private Currency ron;  
    private Money m1;  
  
    @Before public void setUp() {  
        ron = new Currency("RON", 4.8);  
        m1 = new Money(120, ron);  
    }  
  
    @Test public void simpleAdd() {  
        Money m2 = new Money(140, ron);  
        Money result = m1.add(m2);  
        Money expected = new Money(260, ron);  
        assertTrue(expected.equals(result));  
    }  
  
    @Test public void addDifferentCurr() {  
        Currency usd = new Currency("USD", 1.2);  
        Money m2 = new Money(150, usd);  
        Money result = m1.add(m2);  
        Money expected = new Money(720, ron);  
        assertTrue(expected.equals(result));  
    }  
}
```

Alte tooluri de testare unitară

- JUnit este unul din reprezentanții cei mai populari a unei clase de framework-uri numite **xUnit**:
 - <http://en.wikipedia.org/wiki/XUnit>
 - acestea folosesc următoarele elemente comune (adaptate la diverse limbaje și sisteme: “test runner”, “test case”, “assertions”, “text fixtures” (care includ un “setup” și “teardown”), “test suites” și “test execution”, “test result formatter” (pentru afișarea rezultatelor)
- o listă exhaustivă de alte **tooluri de testare unitară**:
http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks