

# Metode de dezvoltare software

---

Testare - partea a 2-a

24.04.2017

Alin Ștefănescu

# Testare... în practică





# **testare de tip cutie albă**

Prezentare bazată pe materiale de Florentin Ipate (UniBuc)

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

- am văzut în cursul anterior că 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

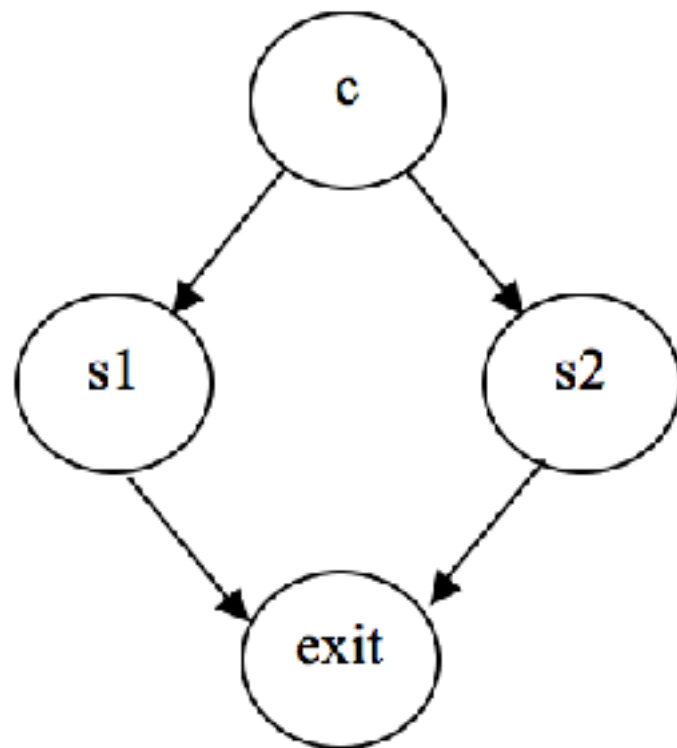
- datele de test sunt generate pe baza implementării (programului), fără a lua în considerare specificația (cerințele) programului
- pentru a utiliza structura programului, acesta poate fi reprezentat sub forma unui **graf orientat**
- datele de test sunt alese astfel încât să parcurgă toate elementele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



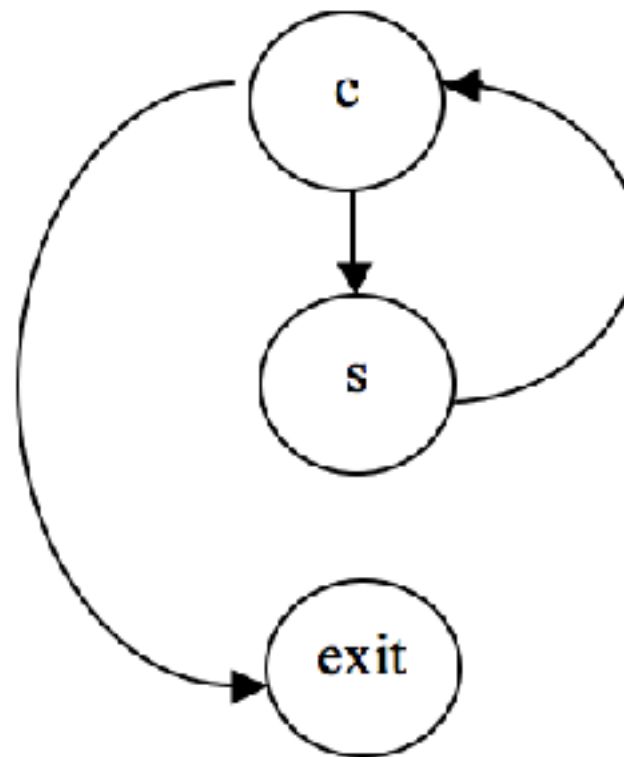
# Transformarea programului într-un graf orientat

- pentru o secvență de instrucțiuni se introduce un nod
- două noduri  $n1$  și  $n2$  sunt conectate dacă este posibil să se execute  $n2$  **imediat** după  $n1$

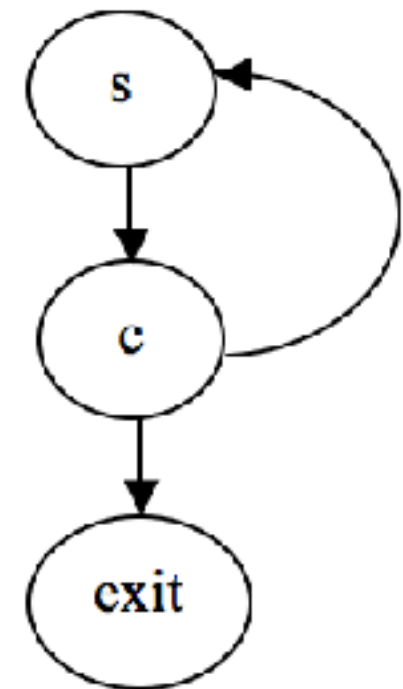
if c then s1 else s2



while c do s

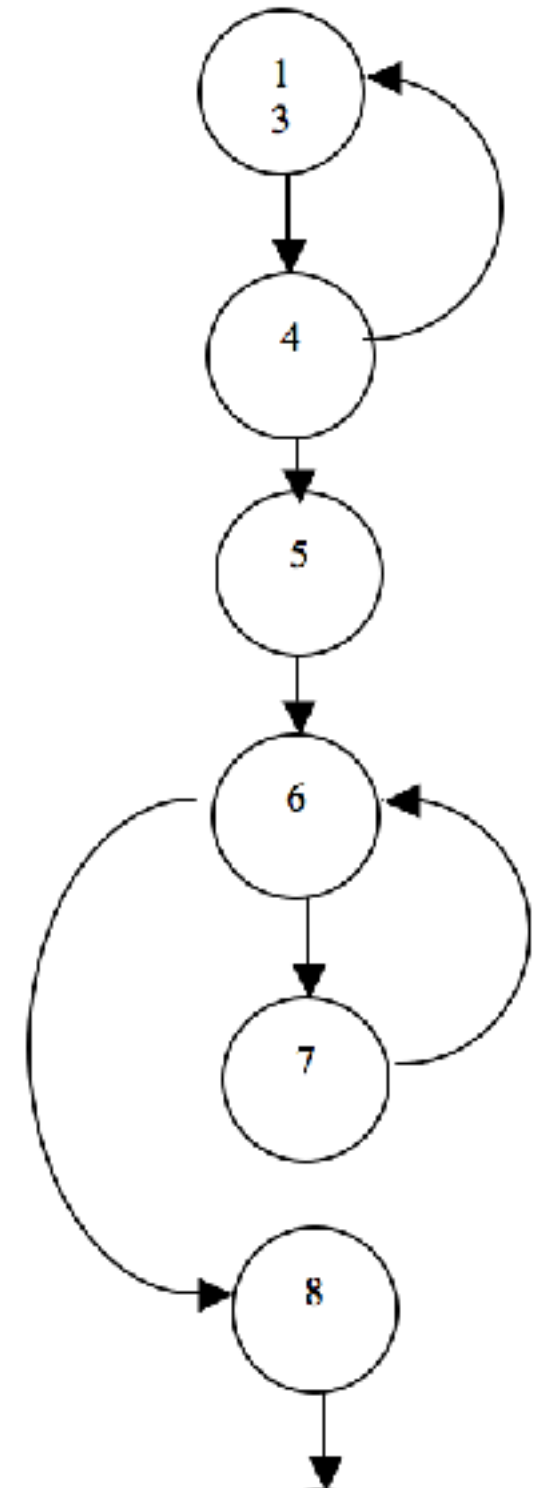


repeat s until c



# Exemplu: program + graf asociat

```
public class Test {  
    public static void main(String[] arg) {  
        KeyboardInput in = new KeyboardInput();  
        char s, c, nl;  
        boolean found;  
        int n,i;  
        char[] x = new char[20];  
  
1    do {  
2        System.out.println("Input an integer  
                               between 1 and 20: ");  
3        n = in.readInteger();  
4    } while ( n<1 || n>20 );  
  
5    System.out.println("input "+n+" character(s):");  
  
6    for (i=0; i<n; i++)  
7        x[i] = in.readCharacter();  
8    nl = in.readCharacter();  
  
... continuare
```



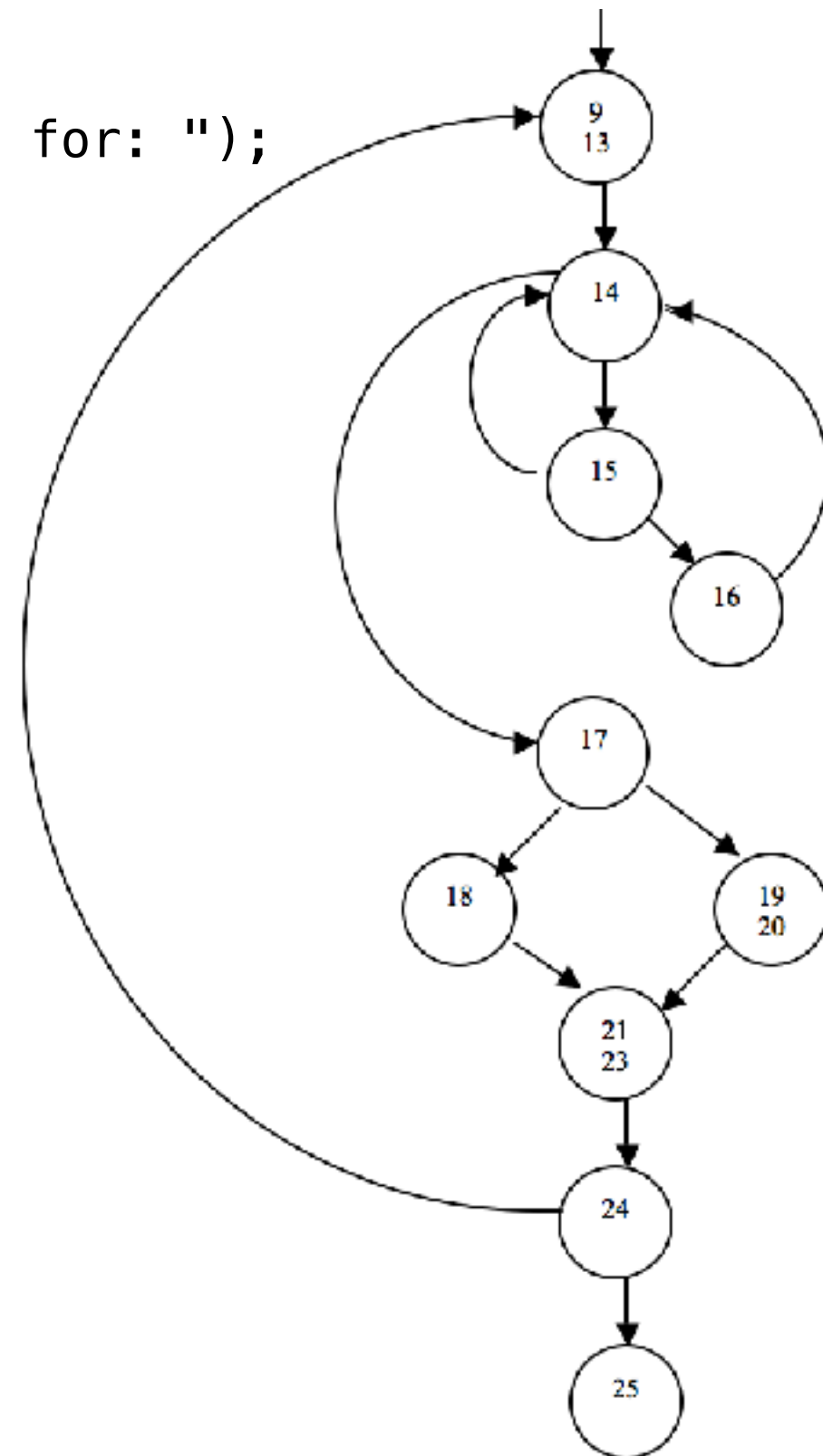
# Exemplul continuat

... *continuare*

```
9  do {
10    System.out.println("Input character to search for: ");
11    c = in.readCharacter();
12    nl = in.readCharacter();
13    found=false;

14    for (i=0; !found && i<n; i++)
15        if (x[i]==c)
16            found=true;
17    if (found)
18        System.out.println("character "+c+
19                           " appears at position "+i);
19    else
20        System.out.println("character "+c+"
21                           does not appear in string");

21    System.out.println("Search for
22                        another character? [y/n]: ");
22    s = in.readCharacter();
23    nl=in.readCharacter();
24 } while ((s=='y') || (s=='Y'));
25 }
```





# Acoperiri

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

# Acoperire la nivel de instrucțiune (*statement coverage*)

Pentru a obține o acoperire la nivel de instrucțiune, trebuie să ne concentrăm asupra acelor instrucțiuni care sunt controlate de condiții (acestea corespund ramificațiilor din graf)

| Intrări  |          |          |          | Instrucțiuni parcurse          |
|----------|----------|----------|----------|--------------------------------|
| <i>n</i> | <i>x</i> | <i>c</i> | <i>s</i> |                                |
| 1        |          |          |          | 1..3, 4, 5, 6                  |
|          | a        |          |          | 7, 6, 8, 9..13                 |
|          |          | a        |          | 14, 15, 16, 14, 17, 18, 21..23 |
|          |          |          | y        | 24, 9..13                      |
|          |          | b        |          | 14, 15, 14, 17, 19..20, 21..23 |
|          |          |          | n        | 24, 25                         |

# Acoperire la nivel de instrucțiune

- testarea la nivel de instrucțiune este privită de obicei ca nivelul minim de acoperire testarea structurală. Atunci când oferi un software e bine să fi executat măcar o dată fiecare instrucțiune a lui.
- totuși, destul de frecvent, acesta acoperire nu poate fi obținută, din următoarele motive:
  - existența unei porțiuni izolate de cod, care nu poate fi niciodată atinsă. Această situație indică o eroare de design și respectiva porțiune de cod trebuie înlăturată.
  - existența unor porțiuni de cod sau subrutine care nu se pot executa decât în situații speciale (subrutine de eroare, a căror execuție poate fi dificilă sau chiar periculoasă). În astfel de situații, acoperirea acestor instrucțiuni poate fi înlocuită de o inspecție riguroasă a codului.

# Avantaje și dezavantaje

## Avantaje

- realizează execuția măcar o singură dată a fiecărei instrucțiuni
- în general, ușor de realizat

## Dezavantaje

- nu testează fiecare condiție în parte în cazul condițiilor compuse (de exemplu, pentru a se atinge o acoperire la nivel de instrucțiune în programul anterior, nu este necesară introducerea unei valori mai mici ca 1 pentru  $n$ )
- nu testează fiecare ramură (tranzițiile în graful asociat)
- probleme suplimentare apar în cazul instrucțiunilor *if* a căror clauză *else* lipsește. În acest caz, testarea la nivel de instrucțiune va forța execuției ramurii corespunzătoare valorii “adevărat”, dar, deoarece nu există clauza *else*, nu va fi necesară și execuția celeilalte ramuri. Metoda poate fi extinsă pentru a rezolva această problemă.

# Acoperire la nivel de ramură (*branch coverage*)

- este o extindere naturală a metodei precedente.
- generează date de test care testează cazurile când fiecare decizie este adevărată sau falsă.
- se mai numește și “*decision coverage*”

| Intrări  |          |          |          | Rezultatul afișat  |
|----------|----------|----------|----------|--|
| <i>n</i> | <i>x</i> | <i>c</i> | <i>s</i> |  |
| 25       |          |          |          | cere introducerea unui întreg între 1 și 20                |
| 1        | a        | a        | y        | afișează poziția 1; se cere introducerea unui nou caracter |
|          |          | b        | n        | caracterul nu apare  |

**Dezavantaj:** nu testează condițiile individuale ale fiecărei decizii

# Acoperire la nivel de condiție (*condition coverage*)

- generează date de test astfel încât fiecare condiție individuală dintr-o decizie să ia atât valoarea *adevărat* cât și valoarea *fals* (dacă acest lucru este posibil)
- de exemplu, dacă o decizie ia forma  $c1 \parallel c2$  sau  $c1 \ \&\& \ c2$ , atunci acoperirea la nivel de condiție se obține astfel încât fiecare dintre condițiile individuale  $c1$  și  $c2$  să ia atât valoarea *adevărat* cât și valoarea *fals*

**Notă:** “decizie” înseamnă orice ramificare în graf, chiar atunci când ea nu apare explicit în program. De exemplu, pentru construcția “*for (i = 0; i < n; i++)*” condiția implicită este  $i < n$



# Decizii

| Decizii                       | Condiții individuale |
|-------------------------------|----------------------|
| while (n<1    n>20)           | n < 1, n > 20        |
| for (i=0; i<n; i++)           | i < n                |
| for (i=0; !found && i<n; i++) | found, i < n         |
| if (a[i]==c)                  | a[i] == c            |
| if (found)                    | found                |
| while ((s=='y')    (s=='Y'))  | (s=='y'), (s=='Y')   |

# Acoperire la nivel de condiție

| Intrări |   |   |   | Rezultatul afișat   |
|---------|---|---|---|---|
| n       | x | c | s |   |
| 0       |   |   |   | cere introducerea unui întreg între 1 și 20                 |
| 25      |   |   |   | cere introducerea unui întreg între 1 și 20                 |
| 1       | a | a | y | afișează poziția 1; se cere introducerea unui nou caracter  |
|         |   | b | Y | caracterul nu apare; se cere introducerea unui nou caracter |

# Avantaje și dezavantaje

## Avantaje


- se concentrează asupra condițiilor individuale

## Dezavantaje

- poate să nu realizeze o acoperire la nivel de ramură. De exemplu, datele anterioare nu realizează ieșirea din bucla *while ((s=='y') || (s=='Y'))* (condiția globală este în ambele cazuri adevărată). Pentru a rezolva aceasta slăbiciune se poate folosi testarea la nivel de condiție/decizie.

# Acoperire la nivel de condiție/decizie (C/D)

- generează date de test astfel încât fiecare condiție individuală dintr-o decizie să ia atât valoarea *adevărat* cât și valoarea *fals* (dacă acest lucru este posibil) și fiecare decizie să ia atât valoarea *adevărat* cât și valoarea *fals*.

| Intrări |   |   |   | Rezultatul afișat   |
|---------|---|---|---|---|
| n       | x   | c | s |   |
| 0       |   |   |   | cere introducerea unui întreg între 1 și 20                 |
| 25      |   |   |   | cere introducerea unui întreg între 1 și 20                 |
| 1       | a   | a | y | afișează poziția 1; se cere introducerea unui nou caracter  |
|         |   | b | Y | caracterul nu apare; se cere introducerea unui nou caracter |
|         |  | b | n | caracterul nu apare   |

# Acoperirea MC/DC

Pentru a îmbunătăți acoperirea anterioară se poate încerca **acoperirea la nivel de condiții multiple** (*multiple condition coverage*):

- se generează date de test astfel încât să se parcurgă toate combinațiile posibile de *adevărat* și *fals* ale condițiilor individuale.
- problema: poate genera o explozie combinatorică (pentru  $N$  condiții pot fi necesare  $2^N$  teste).

**Soluție:** o formă modificată a “**condition/decision coverage**” (prescurtat **MC/DC**)

- fiecare condiție individuală dintr-o decizie ia atât valoarea *true* cât și valoarea *false*
- fiecare decizie ia atât valoarea *true* cât și valoarea *false*
- fiecare condiție individuală influențează în mod independent decizia din care face parte

# Acoperirea MC/DC

“fiecare condiție individuală influențează în mod independent decizia din care face parte” înseamnă că: decizia se schimbă când schimbăm condiția respectivă și păstrăm neschimbate restul celorlalte condiții.

**De exemplu:** pentru o expresie AND cu două condiții C1 și C2

| Test | C1    | C2    | $C1 \wedge C2$ |
|------|-------|-------|----------------|
| t1   | true  | true  | true           |
| t2   | true  | false | false          |
| t3   | false | true  | false          |
| t4   | false | false | false          |

t1 și t4 acoperă dpdv C/D  
dar nu MC/DC

---

t1 și t3 acoperă C1,  
t1 și t2 acoperă C2,  
deci **{t1, t2, t3} acoperă MC/DC**



# Acoperirea MC/DC - alt exemplu

Un alt exemplu:  $C = C1 \wedge C2 \vee C3$

| Test | C1    | C2    | C3    | C     |
|------|-------|-------|-------|-------|
| t1   | true  | true  | false | true  |
| t2   | false | true  | false | false |
| t3   | true  | true  | false | true  |
| t4   | true  | false | false | false |
| t5   | true  | false | true  | true  |
| t6   | true  | false | false | false |

t1 și t2 acoperă C1; t3 și t4 acoperă C2; t5 și t6 acoperă C3

deci {t1, t2, t3, t4, t5, t6} acoperă C dpdv MC/DC

Există însă un **set minimal** {t1,t2,t4,t5} care acoperă C dpdv MC/DC

(t1 și t2 acoperă C1; t1 și t4 acoperă C2; t4 și t5 acoperă C3)

# Acoperirea MC/DC

## Avantaje:

- acoperire mai puternică decât acoperirea condiție/decizie simplă, testând și influența condițiilor individuale asupra deciziilor
- produce teste mai puține – depinde liniar de numărul de condiții

**Notă:** Standardul pentru software pentru aviație DO-178B cere ca tot software-ul de nivel A (pentru controlul zborului și al aterizării) să fie testat cu acoperirea MC/DC.

# Acoperirea la nivel de cale (*path coverage*)

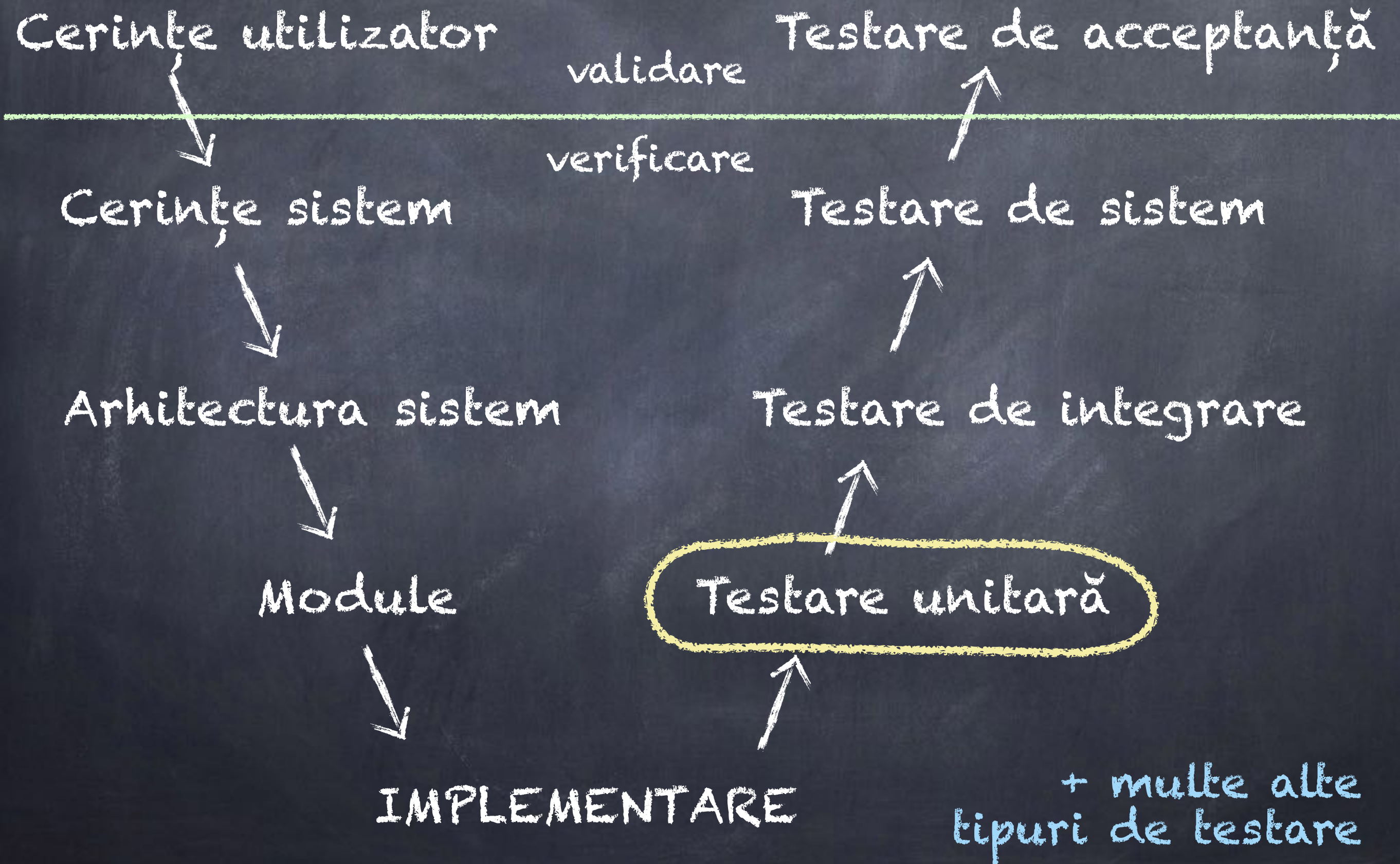
- generează date pentru executarea fiecărei căi măcar o singură dată
- **Problemă:** în majoritatea situațiilor există un număr infinit (foarte mare) de căi
- **Soluție:** Împărțirea căilor în clase de echivalență.
- De exemplu: 2 căi pot fi considerate *echivalente* dacă diferă doar prin numărul de apariții a buclelor în căile respective
- O altă metodă (Paige si Holthouse - 1977): Dacă un program este structurat, atunci acesta poate fi caracterizat de o expresie regulată formată din nodurile grafului. Se obțin teste din cuvinte care satisfac expresia regulată.
  - avantaje: sunt selectate căi netestate de metodele anterioare
  - dezavantaje: pot fi multe căi, dintre care o parte nefezabile; nu exersează condițiile individuale ale deciziilor

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)

- reamintire: 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. Eclipse)

# 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.

# 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", 4.4);
        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", 4.4);
        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(670, 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", 4.4);  
        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(670, 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](http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks)



# **Alte tipuri de testare**

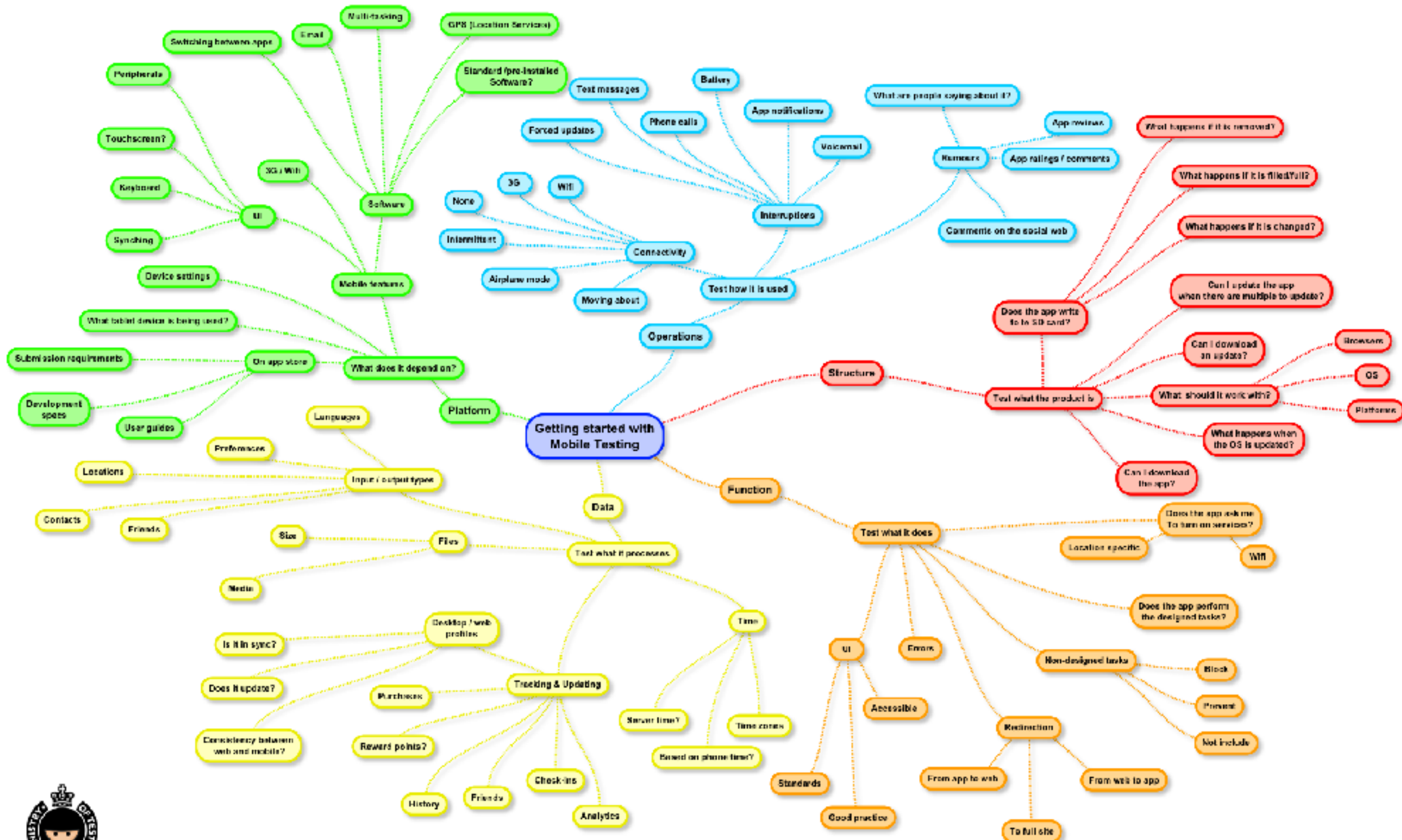


# Alte tipuri de teste

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](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**

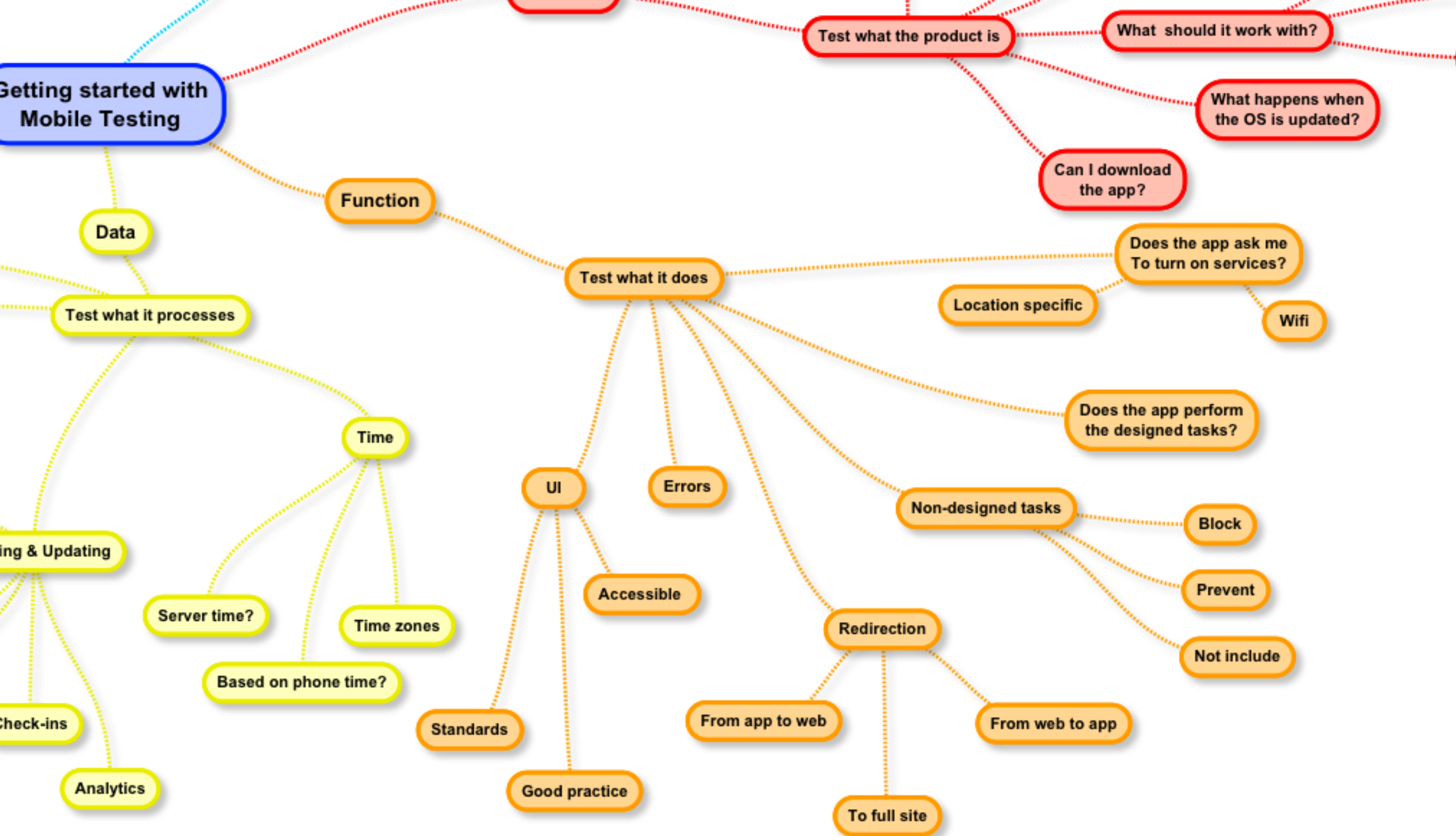
# Aspecte ale testării aplicațiilor mobile





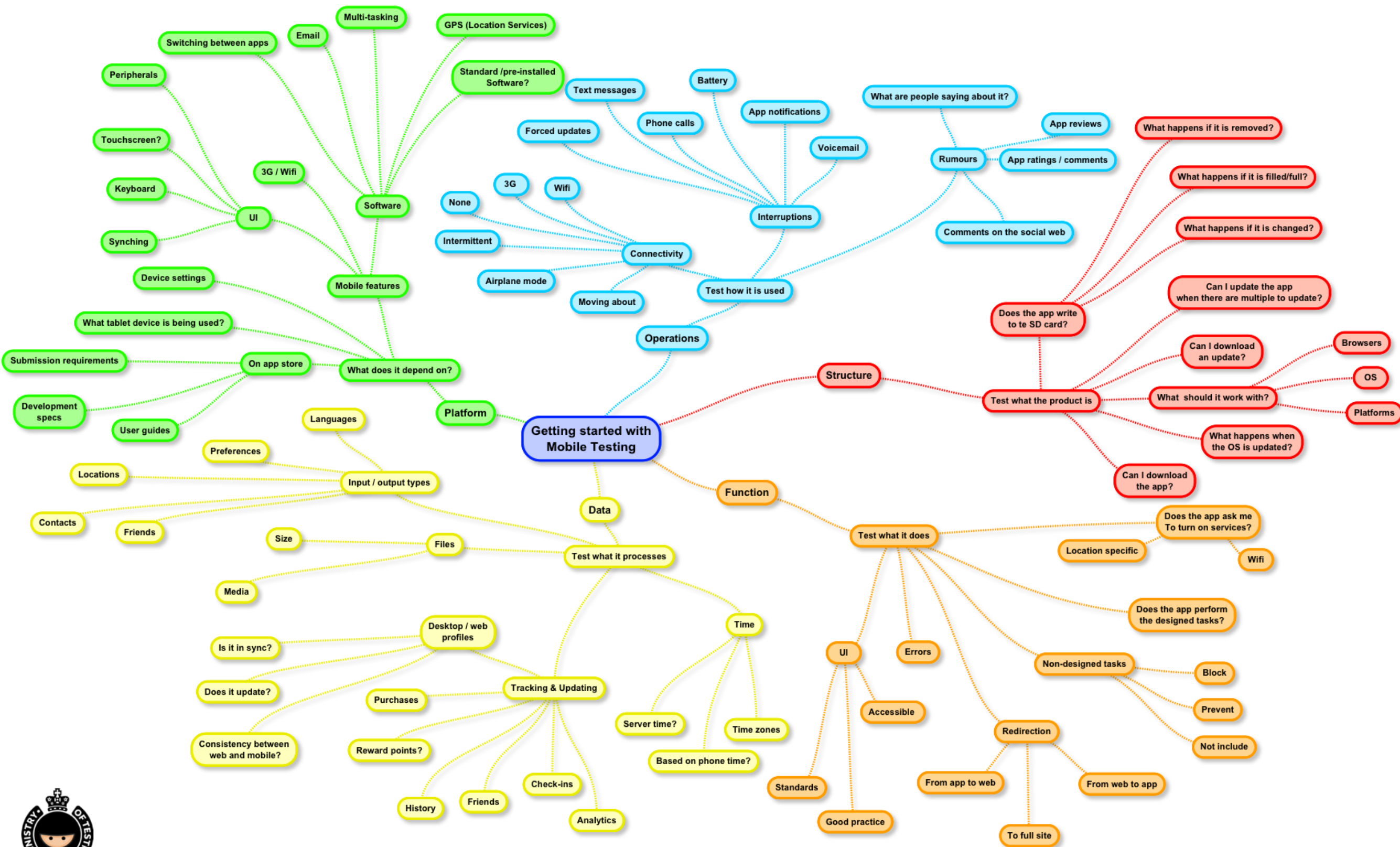






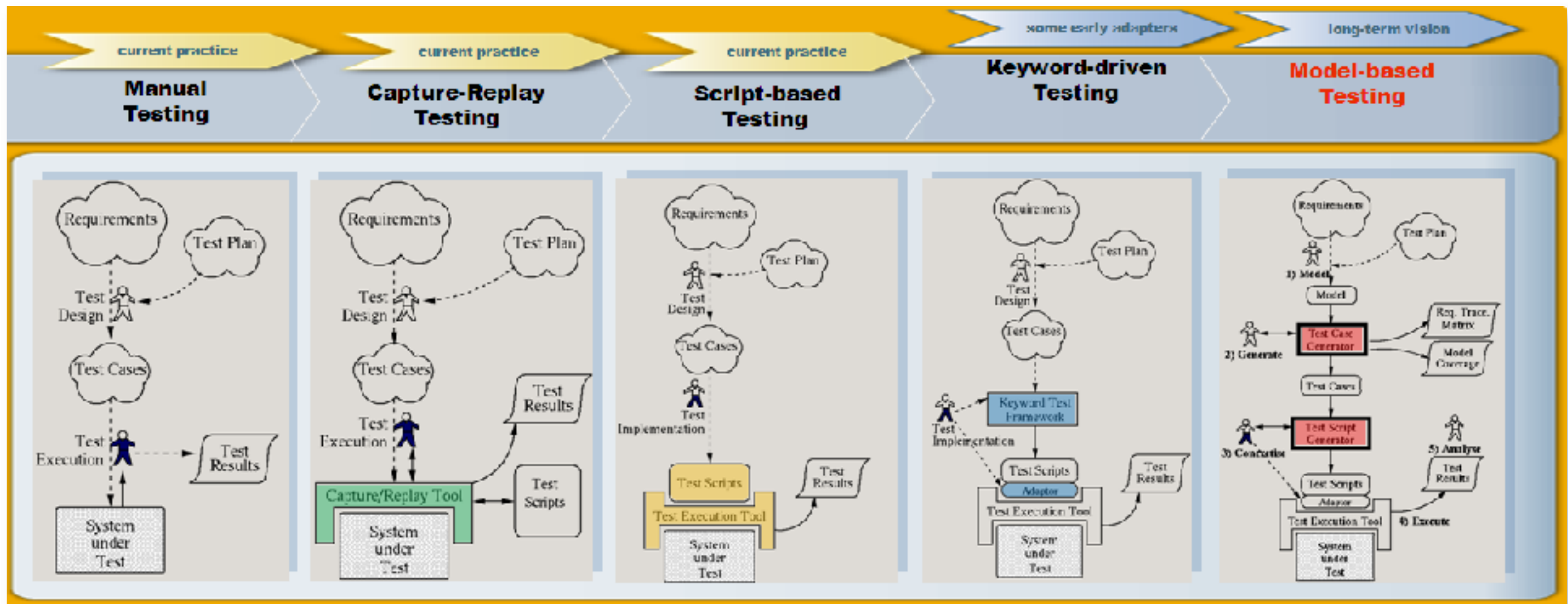






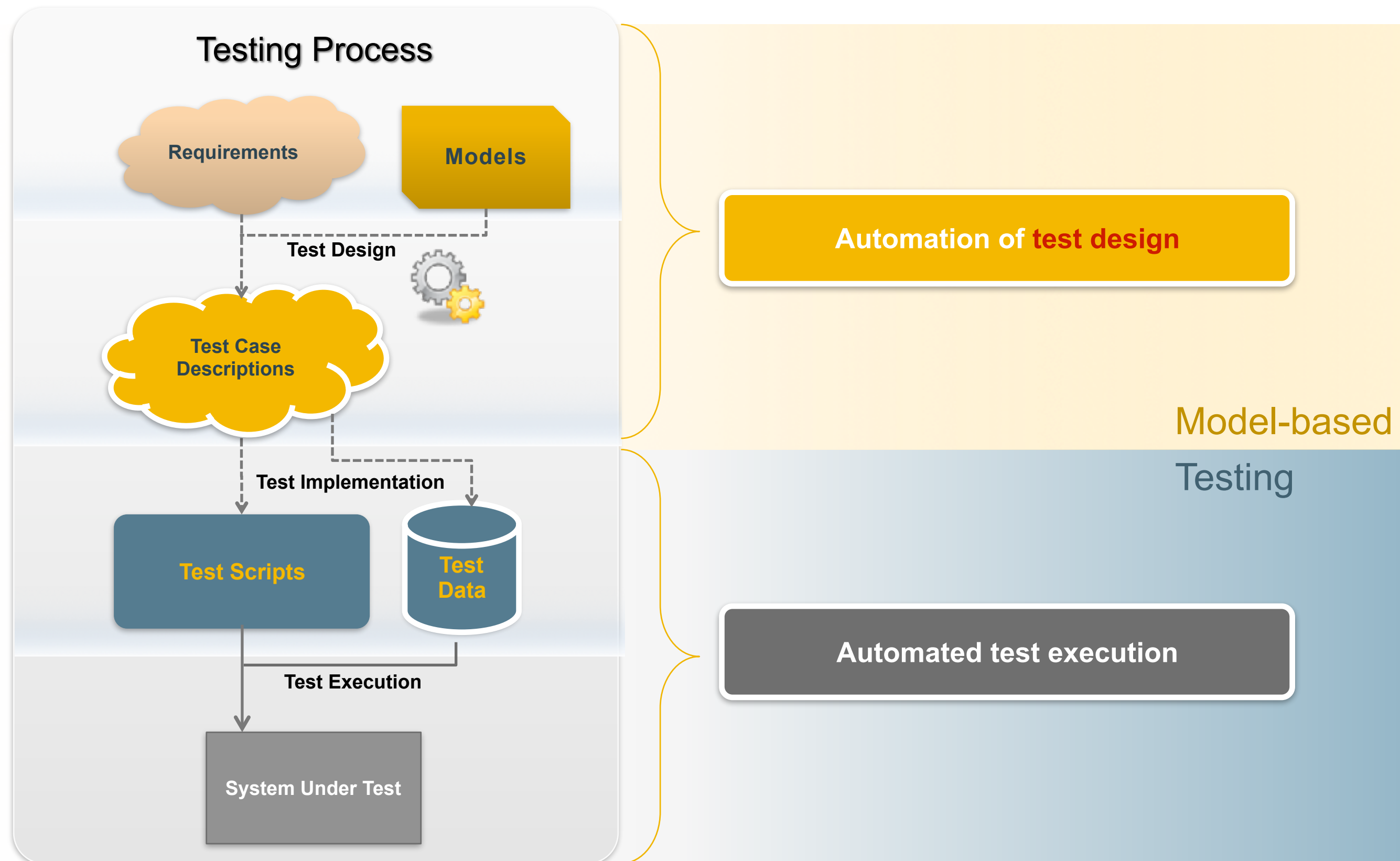


# Evoluția automatizării testelor



source: „Practical Model-based Testing“ Book, 2007

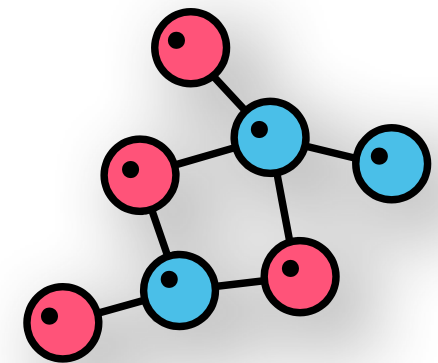
# Testare bazată pe modele (Model-based Testing - MBT)



# Testarea bazată pe modele (MBT)

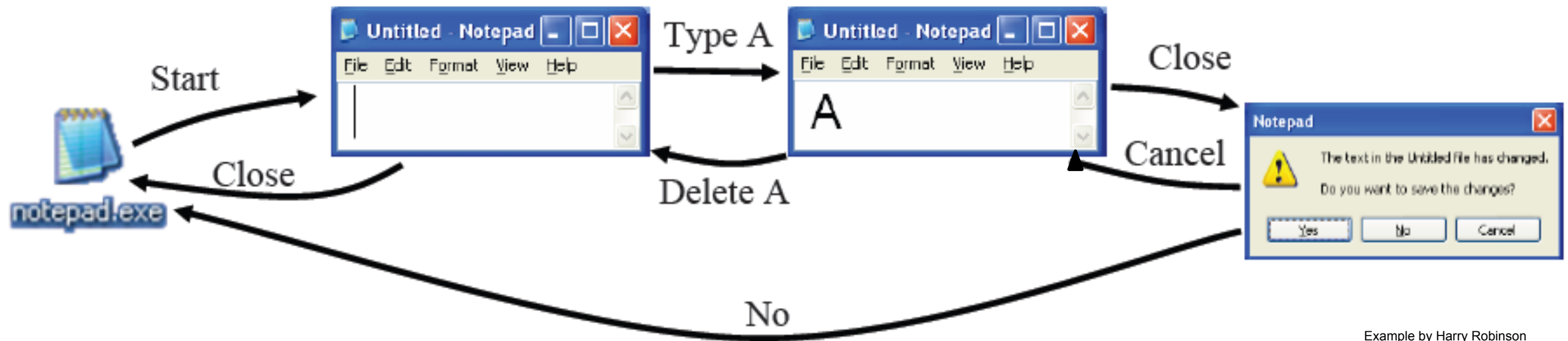
MBT = metodă de generate de test pe baza unor modele

- Un **model** este o descriere abstractă a comportamentului unui sistem.
- În MBT modelele sunt:
  - formalizate
  - salvate într-o anumită formă
  - folosite pentru generarea de teste și oracole
- Modele pot fi de **diferite tipuri**:
  - bazate pe stări (FSM, UML statecharts, LTS)
  - pre- și post-condiții (JML, OCL, B, Z, Spec#)
  - bazate pe interacții (UML diagrame de secvență)
  - operaționale (algebre de proces, rețele Petri), etc.



# Un mic exemplu de MBT

Cum poate arăta un model cu stări:



Example by Harry Robinson

... și ce tipuri de teste pot fi generate:

## Test 1

Execute 'Start'  
Execute 'Close'

## Test 2

Execute 'Start'  
Execute 'Type A'  
Execute 'Delete A'  
Execute 'Type A'  
Execute 'Delete A'  
Execute 'Close'

## Test 3

Execute 'Start'  
Execute 'Type A'  
Execute 'Close'  
Execute 'No'

## Test 4

Execute 'Start'  
Execute 'Type A'  
Execute 'Close'  
Execute 'Cancel'  
Execute 'Delete A'  
Execute 'Close'