Cursul 1: Functional Testing

- Datele de test sunt generate pe baza specificatiilor programului, alcatuite din pre-conditii si post-conditii.
  - *Avantaje:* reduce numarul de date de test doar pe baza specificatiei si sunt potrivite pentru aplicatii de tipul procesarii datelor in care intrarea si iesirea sunt usor de identificat.
  - *Dezavantaje:* modul de definire a claselor nu e evident, uneori poate parea ca unele valori sunt procesate identic desi nu sunt, mai putin aplicabile cand intrarea si iesirea sunt simple iar procesarea este complexa.

1. **Partitionare de echivalenta:** Ideea de baza este de a imparti datele de intrare in *partii de echivalenta* astfel incat datele dintr-o clasa sunt tratare in mod identic.
   - Cum toate valorile dintr-o clasa sunt procesate in acelasi fel, este suficient sa alegem cate o valoarea din fiecare clasa.
   - Clasele de echivalenta nu trebuie sa se suprapuna. Pot fi alese si date invalide.
- Setul de ***date de test*** este determinat prin combinarea claselor individuale din domeniul de intrare si cel de iesire
2. **Analiza valorilor de frontiera:** folosita impreuna cu partitionarea de echivalenta.
   - Datele de frontiera sunt o sursa importanta de erori
3. **Partitionarea in categorii:** generarea de date de test care acopera functionalitatea sistemului si maximizeaza posibilitatea de a gasi erori. Cuprinde urmatorii pasi:
   - 1. Descompunerea in unitati care pot fi testate separat
   - 2. Identificarea *parametrilor* si a conditiilor de mediu
   - 3. Gasirea unor *categorii*, caracteristici importante pentru fiecare parametru
   - 4. Partitionarea fiecarei categorii in alternative: *multimi de valori*
   - 5. Scrierea specificatiei de testare: *lista categoriilor si a alernativelor*
   - 6. Crearea *cazurilor de teste* prin alegerea combinatiilor de alternative
   - 7. Crearea *datelor de test* prin alegerea unei singure valori pentru fiecare alternativa

(calculul x^y, $x > 0, y \geq 0$)

```
1  begin
2      int x, y, z;
3      read(x, y);
4      z = 1;
5      while (y > 0) {
6          z = z*x;
7          y = y -1;
8      }
9      write(z);
10 end
```

LCSAJ	Start	End	Jump to
1	1	8	5
2	5	8	5
3	5	5	9
4	1	5	9
5	9	9	Exit

Considerăm T = {t1, t2}, unde t1 = (x = 3, y = 0), t2 = (x = 3, y = 2)

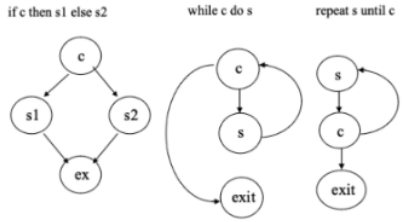
t1: (1, 5, 9) → (9, 9, exit)

t2: (1, 8, 5) → (5, 8, 5) → (5, 5, 9) → (9, 9, exit)

- **Metoda grafului cauza-efect:** partitionarea in categorii poate produce un numar mare de combinatii de intrari dintre care o mare parte pot fi nefezabile.
  - Metoda grafului se concentreaza pe modelarea relatiilor de dependenta intre conditiile de intrare ale programului, **cauza** si conditiile de iesire, **efect**

Cursul 2: Structural Testing

- Datele sunt generate pe baza implementarii, fara a lua in considerare cerintele programului.
  - Programul poate fi reprezentat ca un graf de flux de control
  - Datele de test sunt alese astfel incat sa parcurga toate elementele grafului: instructiunea, ramura sau cale, macar o data



a. **Acoperire la nivel de instructiune**: fiecare instructiune sau nod este parcursa macar o data.

b. **Acoperire la nivel de decizie sau ramura**: genereaza date de test care testează cazurile când fiecare decizie este adevărată sau falsă.

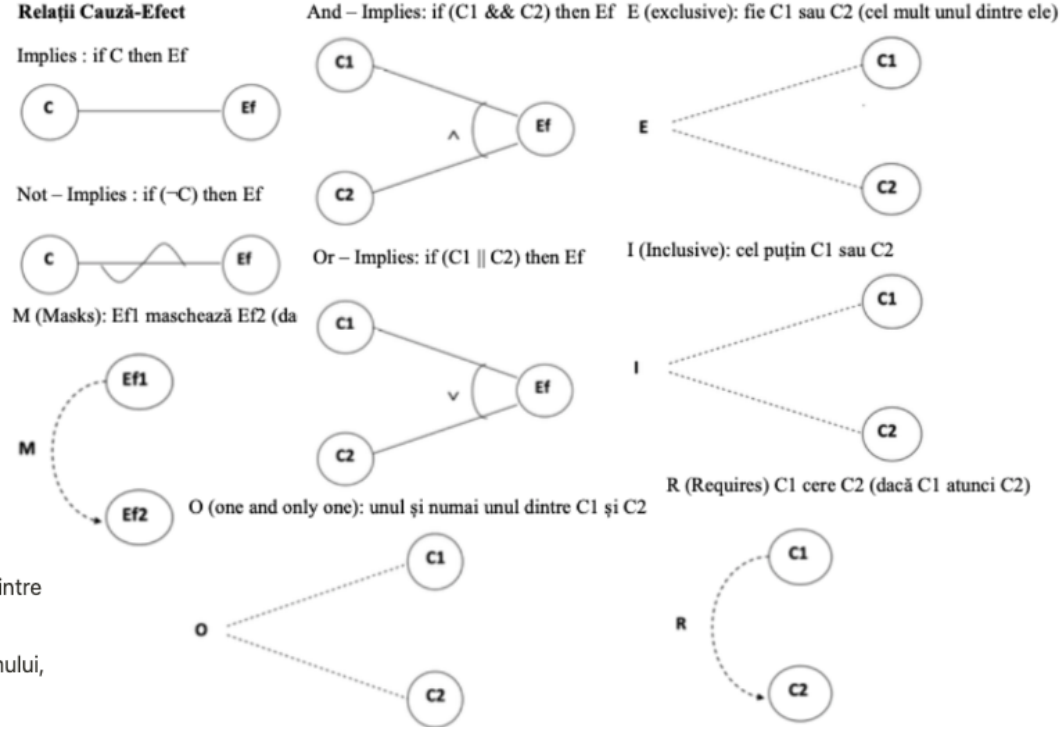
c. **Acoperire la nivel de conditie:** fiecare conditie individuala dintr-o decizie sa ia atat valoarea adevarat cat si valoarea fals. Poate sa nu realizeze acoperire la nivel de decizie.

d. **Acoperire la nivel de conditie si decizie**: fiecare conditie si decizie ia ambele valori. Poate sa nu testeze unele conditii individuale care sunt mascate de alte conditii.

e. **Acoperire la nivel de conditii multiple**: parcurge toate combinațiile posibile de adevărat și fals ale conditiilor individuale. Pentru n conditii pot fi necesare chiar 2^n teste

f. **Acoperire la nivel de cale**: executarea fiecarei cai cel putin o data. Deoarece numarul de cai poate fi foarte mare, acestea se impart in clase de echivalenta.

- **Linerea Code Sequence and Jump coverage (LCSAJ):** o execuție a unui program formată dintr-o secvență de cod liniara urmată de un salt al controlului programului.
  - Un set de test care realizează o acoperire la nivel de decizie nu realizează în mod necesar o acoperire la nivel de LCSAJ



Cursul 3: Mutation Testing

- **Analiza mutantilor**: având un set de teste generat, putem evalua cât de eficient este, pe baza rezultatelor obținute de acest test asupra mutanților programului.
  - **Mutatie**: modificarea foarte mica din punct de vedere sintactic a unui program. Mutantul trebuie sa fie corect din punct de vedere sintactic.
  - Generarea mutanților pentru programul P: rularea setului de teste asupra programului P și asupra setului de mutanți Dacă un test distinge între P și un mutant M spunem că P ***omoară*** mutantul M.
- Mutanti de *primul ordin*: obtinuti facand o singura modificare in program
  - Mutanti de ordin n: sunt facute n modificari in program

Datele de test care disting orice program care diferă cu puțin de programul corect sunt
suficient de puternice pentru a distinge erori mai complexe.

- **Weak mutation:** un test aduce pe P si M in stari diferite dupa rularea instructiunii mutante
  - **Strong mutation:** conditia de weak mutation si schimbarea starii se propaga pana la finalul programului, iar efectul poate fi observat imediat dupa terminarea programului
- Un mutant M al lui P se numeste echivalent daca el se comporta identic cu programul P pentru orice date de intrare. Altfel se spune ce M poate fi distins de P
  - Teoretic determinarea echivalentei unui mutant fata de părinte este nedecidabilă.
  - Practic determinarea echivalentei se face prin analiza codului.
- **Mutation score:** $MS(t) = D/(L + D)$ unde
  - D este numarul de mutanti distinsi, *dead mutants*
  - L este numarul de mutanti nedistinsi, *live mutants,* neechivalenti cu P

Cursul 7: Random testing

- **Random testing** is mostly generating stupid test cases, but if it can generate a clever test case, say one in a million times, then that still might be a more effective use of our testing resources than writing test cases by hand.
  - For our random tests to be usefull, the focus should be on external interfaces provided, things like file I/O and the graphical user interface.
- *Advantages*: less tester bias, once testing is automated human cost if testing goes to nearly zero, often surprises us, every fuzzer finds different bugs
- *Disadvantages*: input validity can be hard, no stopping criteria, may find unimportant bugs, may find the same bugs many times, can be hard to debug

Cursul 8: Testing in practice

- **Bug triage** is the process by which the *severity* of different bugs is determined, and we start to disambiguate between different bugs in order find which bugs we can *report in parallel.*
- **Test-case reduction** or test-case minimization is an automated process of taking some large input that triggers a failure and turning it into a small input.
- **A test suite** is a collection of tests that can often be run automatically and periodically. It contains small feature-specific tests, large realistic tests and regressions tests.

Cursul 4: Testing

- **Fault injection** into a *system under test*, SUT, is the process of intentionally introducing faults that we want out code to be robust to
- **White box testing** refers to is the fact that the tester is using detailed knowledge about the *internals of the system* in order to construct better test cases
  - **Black box testing** refers to the fact that we are rather testing the system based on what we know about *how it's supposed to respond* to our test cases.
- **Unit testing** means looking at some **small software module** at a time and testing it in an *isolated* fashion. Usually the person performing the unit testing is the same person who implemented the module, and in that case we may well be doing white box.
- **Integration testing** refers to taking multiple software modules that have already been unit tested and testing them in *combination with each other.*
- **System testing** validated that the system as a whole meets its goals. Often we are doing black box testing because the system is large enough and not testing all possible use cases.
- In **differential testing** we are taking the same test input delivering it to two different implementations of the SUT and comparing them for equality.
- **Stress testing** is where a system is tested at or beyond its normal usage limits.
- In **random testing** we use the results of a *pseudo-random number generator* to randomly create test inputs, and we deliver those to the SUT.
- **Regression testing** always involves taking inputs that previously made the system fail and replaying them against the system.

Cursul 5: Coverage Testing

- **Test coverage** is an automatic way of partitioning the input domain with some observed features of the source code. It tries to accomplish the exact same thing as **partitioning**.
  - One particular kind of test coverage is called **function coverage** and is achieved when *every function* in our source code is executed during testing.
  - **Test coverage** is a measure of the *proportion* of a program exercised during testing.
  - Coverage is not particularly useful in spotting bugs in the system under test.
- **Statement** coverage vs **Line** coverage**:** a line can contain multiple statements
- **Branch** coverage vs **Statement** coverage: branch or *decision* coverage is a metric where a branch in a code is covered if it executes both ways. Statement coverage doesn't include "if" statements that have no "else" branch, while branch coverage does.
- **Loop coverage** specifies that we execute each loop 0 times, once, and more than once.
- **Modified condition decision coverage** starts off with a branch coverage, it additionally states that every condition involved in a decision takes on every possible outcome.
- **Path coverage:** cares about how you got to a certain piece of code by following a path. It is impossible to achieve in real applications.
- **Boundary value coverage** is when a program depends on some numerical range and when the program has different behaviors based on numbers within that range then we should test *numbers close to the boundary.*
- We believe that if we have a good test suite, and we measure its coverage, the coverage will be good. We do not believe, on the other hand, that if we have a test suite which gets good coverage, it must be a good test suite.