Tematică de Examen
Verificarea şi Validarea Sistemelor Soft

May 26, 2025

Contents

1 Curs 01. Verificare şi Validare. Inspectare

1.1 Concepte, caracteristici, asemănări şi diferenţe

1.1.1 Verificare, Validare; Verificare vs. Validare

- **Verificare (Verification)**: Procesul prin care se asigură că produsul este dezvoltat conform cerinţelor, specificaţiilor şi standardelor. Întrebare asociată: Dezvoltăm corect produsul? (Are we building the product right?). Stabilește dacă rezultatul unei etape de dezvoltare satisface cerințele acelei etape; implică asigurarea consistenței, completitudinii, corectitudinii şi aplică metode de control al calităţii.

- **Validare (Validation)**: Procesul prin care se asigură că produsul dezvoltat satisface cerinţele utilizatorului. Întrebare asociată: Dezvoltăm produsul corect (de care are nevoie clientul)? (Are we building the right product?). Confirmă că produsul satisface cerințele de utilizare; se desfășoară spre sfârşitul procesului de dezvoltare pentru a demonstra că întregul sistem satisface nevoile și așteptările; se aplică asupra întregului sistem, în contextul real în care va funcţiona, folosind diferite tipuri de testare.

- **Verificare vs. Validare în modelul V**: Modelul V ilustrează corespondența dintre etapele de dezvoltare (stânga) și activitățile de testare (dreapta), unde verificarea se face între etape succesive de pe partea stângă și validarea se face prin compararea rezultatelor testării cu cerințele din etapele corespunzătoare de pe partea stângă. (Vezi Curs 01A, pagina 12 pentru diagramă).

- Activităţi de Verificare includ: style checkers, static analysis, proofs of correctness, consistency checking, code inspection, model/specification inspection, modeling (e.g., UML, formal methods), goal analysis.

- Activităţi de Validare includ: unit test, integration test, system test, customer acceptance test, beta test, usability test, prototyping, automated testing, robustness analysis.

1.1.2 Eroare, Defect/Bug, Defecţiune; Eroare vs. Defect/Bug vs. Defecţiune

- **Eroare (Error, Mistake; Greşeală)**: O acțiune umană care are ca rezultat un defect în produsul software.

- **Defect (Fault, i.e., Bug)**: Consecinţă a unei erori. Un defect poate fi latent: nu cauzează probleme până când nu apar anumite condiții (engl. failure triggers) care determină execuția anumitor linii de cod sursă. Orice aspect al unui produs soft care cauzează reducerea inutilă și inadecvată a calității produsului soft sau constituie o amenințare asupra imaginii produsului. Exemple: deficienţe de proiectare, greşeli în documentații, utilizare cu dificultate a programului. Anumite aspecte ale produsului pot limita calitatea acestuia, dar nu pot fi considerate defecte (e.g., constrângeri de utilizare). În cadrul cursului, orice deficienţă sau problemă a produsului soft este denumită bug (defect). Sinonime: variance, problem, inconsistency, error, incident, anomaly.

- **Defecţiune (Failure)**: Devierea de la comportamentul obişnuit al unei componente software. Apare atunci când comportamentul observabil al programului nu corespunde specificației sale. Procesul de manifestare a unui defect: când execuția programului întâlnește un defect, acesta provoacă o defecţiune.
- **Eroare vs. Defect/Bug vs. Defecţiune**: O **eroare** (acțiune umană) produce un **defect/bug** (o problemă în cod/documentație), care, în anumite condiții de execuție, poate duce la o **defecţiune** (comportament incorect observabil al programului).

1.1.3 Stakeholders, Calitate, QA, QC

- **Stakeholder (rom. beneficiar, utilizator)**: O persoană care manifestă un interes particular pentru succesul sau eşecul unui produs soft.
  - **Tipuri de Stakeholders**:
    - *Primar/Secundar*: Beneficiar primar - direct afectat; Beneficiar secundar - nu este afectat direct.
    - *Preferat/Nedorit/Neutru/Ignorat*: Preferat (avantajat) - produsul este proiectat pentru el; Nedorit (dezavantajat) - produsul e proiectat să creeze dificultăți; Neutru - produsul nu e pentru el și nu îl poate influența; Ignorat (neglijat) - produsul nu e proiectat pentru el.
- **Calitatea Produselor Soft (Definiții)**:
  - Produsul soft este conform cu cerințele documentate.
  - Produsul soft este conform cu cerințele reale ale utilizatorului.
  - Produsul soft este adecvat pentru a fi utilizat (satisfiers vs. dissatisfiers).
  - Produsul soft este relevant/important pentru o persoană (calitatea este subiectivă).
- **Asigurarea Calităţii (Quality Assurance - QA)**: Abordare din perspectiva procesului. Obiectiv: asigură respectarea standardelor, planurilor și etapelor proceselor de dezvoltare. Prevenţie bug-uri, orientare pe proces, planificarea şi monitorizarea activităţilor.
- **Controlul Calităţii (Quality Control - QC)**: Abordare din perspectiva produsului. Obiectiv: identifică deficienţele în produsul obţinut. Detecţie bug-uri, orientare pe produs, căutare şi eliminare bug-uri.

1.1.4 Analiza Statică vs. Analiza Dinamică

- **Analiza Statică (Static Testing)**: Examinarea unor documente (specificații, modele, cod sursă, planuri de testare, etc.) fără execuția propriu-zisă a programului. Exemple: inspectarea codului, analiza algoritmului, demonstrarea corectitudinii. Poate fi bazată pe factorul uman (reviews) sau tool-uri. Permite identificarea mai multor erori care pot fi corectate simultan.

- **Analiza Dinamică (Dynamic Testing)**: Examinarea comportamentului programului prin execuție, cu scopul de a evidenția defecţiuni posibile. Include activitatea de execuție propriu-zisă a programului (testare). Exemple: tipuri de testare (regresie, funcţională, non-funcţională), niveluri de testare. Sugerează doar un simptom, fiecare eroare fiind eliminată individual. Poate evidenţia o defecţiune doar în anumite situaţii.

- Sunt metode de analiză complementare; dezvoltatorii aplică metode hibride.

1.1.5 HbT (Human-based Testing), Motivaţie

- **HbT (Human-based Testing) - Metode bazate pe factorul uman (Reviews)**: Verificare efectuată de o persoană sau un grup de persoane la sfârşitul unei etape a procesului de dezvoltare și înainte de a demara următoarea fază. Exemplu: inspectarea codului sursă după implementare și înainte de testare.

- **Motivaţie HbT**: Utilizarea metodelor HbT contribuie la creșterea productivităţii şi a gradului de încredere că rezultatul obţinut îndeplineşte cerinţele. Costul de corectare al defectelor creşte odată cu parcurgerea etapelor de dezvoltare. Modificarea comportamentului programatorilor la depanare (se pot introduce mai multe bug-uri).

- **Obiective HbT**: Identificarea defectelor.

1.1.6 Inspectare Fagan, Walkthroughs, Technical Review, Informal Review

- **Inspectare Fagan**:

  - *Descriere*: Proces structurat de identificare a defectelor din documente pe baza unor criterii prestabilite. Activitate riguroasă.
  - *Rolurile membrilor echipei* (4 membri):
    - Moderator: Distribuie materiale, planifică și conduce sesiunile, urmărește corectarea erorilor.
    - Autorul documentului: Răspunde la întrebări, clarifică, participă la discuții, remediază defecțiunile.
    - Secretar: Redactează concluziile, înregistrează defectele și problemele discutate.
    - Prezentator (Reader): Citeşte părţi ale documentului în ședință.
    - Inspectori: Toți membrii, cu excepția autorului, analizează documentul pentru defecte.
  - *Activităţile asociate* (6 etape):
    - Planificarea (Planning): Alegerea echipei, distribuirea materialelor, verificarea completitudinii documentului.
    - Prezentarea (Overview) (nu este obligatorie): Prezentarea detaliilor materialului.
    - Pregătirea individuală (Preparation): Citirea atentă, înțelegerea documentului, formularea observațiilor și întrebărilor.

    - Şedinţa de inspectare (Inspection meeting): Discutarea observațiilor, notarea defectelor, predarea concluziilor autorului.
    - Corectarea (Rework): Autorul efectuează modificările și corectează erorile.
    - Reinspectarea (Follow-up): Verificarea eliminării erorilor; poate fi o întâlnire între autor și moderator.
  - *Avantaje*: Descoperirea defectelor devreme, reducerea costului și timpului de dezvoltare, metodă de grup, modalitate de învăţare, stabileşte sursa defecţiunii, elimină stresul depanării. Utilizează checklists (liste cu defecte frecvente) adaptate tipului de document.

- **Walkthroughs**:
  - *Descriere*: Proces de identificare a defectelor din documente sub îndrumarea autorului. Activitate mai puţin riguroasă decât inspectarea Fagan. Nu folosește checklists, ci scenarii prestabilite.
  - *Rolurile membrilor echipei* (3-5 membri): Secretar, inspectori și moderator (autorul documentului). Autorul conduce echipa.
  - *Activităţile asociate* (4 etape): Planning, meeting, rework, follow-up.
  - *Avantaje*: Mai puţin formal, condus de autor, util pentru înțelegerea documentului și detectarea problemelor majore.

- **Technical Review**:
  - *Descriere*: Tip de review formal realizat de o echipă calificată tehnic pentru a examina conformitatea unui document cu scopul său și a identifica diferențe față de specificații și standarde. Activitate mai puţin riguroasă decât inspectarea Fagan.
  - *Rolurile membrilor echipei* (3-5 membri): Secretar, inspectori, moderator (conduce echipa) și autorul documentului (toți sunt peer reviewers). Autorul nu este și secretar.
  - *Activităţile asociate* (3-4 etape): Planning, preparation (obligatoriu), meeting (opţional), rework.
  - *Avantaje/Obiective*: Identificarea unui consens, posibilelor defecte, idei noi, motivarea autorului. Utilizarea checklists este opţională.

- **Informal Review**:
  - *Descriere*: Realizat fără o procedură formală sau documentată. Exemple: buddy check, pairing, pair review, over-the-shoulder, e-mail pass-around. Activitate de scurtă durată, puţin riguroasă.
  - *Rolurile membrilor echipei*: Realizare în pereche (autor, inspector) sau echipe (>2 persoane, autor, cel puţin un inspector peer reviewer). Inspectorul este un coleg.
  - *Activităţile asociate* (1-2 etape): Meeting (opţională), rework.
  - *Avantaje/Obiective*: Identificarea posibilelor defecte, idei noi, rezolvarea problemelor minore. Utilizarea checklists opţională, rezultatele se pot documenta. Utilitatea depinde de inspector. Frecvent utilizată în metodologiile Agile.

1.1.7 Pair-Programming

- **Caracteristici**: Metodă de elaborare a programelor în care două persoane lucrează împreună. Combină activităţile de inspectare a codului şi implementare. Programatorii alternează rolurile. Activitățile de inspectare nu sunt determinate de checklists, ci se bazează pe împărtăşirea aceloraşi principii și stil de programare. Nu există mediatori, responsabilitatea pentru atmosfera de lucru deschisă depinde de programatori.

- **Avantaje**: Calitate mai bună a codului, transfer de cunoștințe, rezolvare mai rapidă a problemelor, disciplină crescută. (Avantajele sunt deduse din caracteristici și scopul metodei).

---

2 Curs 02. Testare. Testare Black-Box

2.1 Testare

2.1.1 Definiţii ale testării (4)

1. Semnalează prezența defectelor unui program, fără a garanta absenţa acestora. (Dijkstra1969)

2. Procesul de execuţie al unui program cu scopul de a identifica erori. (Myers2004)

3. Observarea comportării unui program în mai multe execuții. (Frențiu2010)

4. Investigaţie tehnică și empirică realizată cu scopul de a oferi beneficiarilor testării informații referitoare la programul testat. (BBST2010)

Testarea este un proces distructiv; se poate finaliza cu succes (passed) sau eşec (failed).

2.1.2 Terminologie: program, program testat, caz de testare

- **Program (Computer program, software application, software product)**: Listă de instrucţiuni sau o mulţime de metode sau module care permit execuţia de către un calculator. O comunicare între persoane şi calculatoare, separate în timp şi spaţiu, ce conţine instrucţiuni executate de calculator.

- **Program Testat (Software Under Test - SUT)**: Asimilat unei funcţii matematice $P : D \to R$, unde D este mulţimea datelor de intrare și R mulţimea datelor de ieşire aşteptate.

- **Caz de Testare (Test Case)**: Mulțime de date de intrare, condiții de execuție și rezultate așteptate, proiectate cu un anumit scop (e.g., parcurgerea unui drum particular sau verificarea unei cerințe specifice). O interogare adresată de tester programului testat. Notație: (i, r), $i \in D, r \in R$; pentru intrarea i se așteaptă rezultatul r.

2.1.3 Tipuri de testare: exhaustivă, selectivă

- **Testare Exhaustivă (Testare completă, Exhaustive testing, Complete testing)**: Testare cu toate cazurile de testare posibile, folosind toate datele și scenariile de utilizare posibile. Dacă D (domeniul datelor de intrare) este finit, P se poate executa pentru fiecare $i \in D$. În majoritatea situaţiilor D nu este finit, deci testarea exhaustivă nu este posibilă și nici eficace.

- **Testare Selectivă (Selective testing)**: Testare cu o submulțime de cazuri de testare. Dacă D nu este finit, atunci se aleg o parte din elementele i, unde $i \in S, S \subset D$.

2.2 Testare Black-Box

2.2.1 Definiţie, caracteristici

- **Testare Black-Box (Black-box testing, data driven testing, input/output driven testing)**: Testare funcţională. Datele de intrare se aleg pe baza specificaţiei problemei, programul fiind văzut ca o cutie neagră. Nu se utilizează informaţii referitoare la structura internă a programului (codul sursă). Permite identificarea situaţiilor în care programul nu funcţionează conform specificaţiilor.

2.2.2 ECP, BVA, ECP vs. BVA

- **Partiţionarea în Clase de Echivalenţă (Equivalence Class Partitioning - ECP)**:

  - *Definiţie*: Tehnică eficientă pentru reducerea numărului de cazuri de testare. Presupune împărţirea domeniului datelor de intrare/ieşire în clase de echivalenţă (EC), astfel încât, dacă programul rulează corect pentru o valoare dintr-o EC, se presupune că va rula corect pentru orice valoare din acea EC. O clasă de echivalenţă este o mulţime de date de intrare/ieșire pentru care programul are comportament similar.
  - *Etape*:
    1. Identificarea claselor de echivalenţă disjuncte (pentru a evita redundanța). Se clasifică în valide și non-valide.
    2. Proiectarea cazurilor de testare: se alege un singur element din fiecare clasă de echivalenţă. Se scrie un caz de testare pentru cât mai multe EC valide neacoperite și câte un caz de testare pentru fiecare EC non-validă neacoperită.
  - *Reguli de identificare ECs* :
    * Interval de valori [a,b]: 1 EC validă, 2 EC non-valide.
    * Mulțime finită de valori: 1 EC validă pentru fiecare valoare, 1 EC non-validă.
    * Număr de valori: 1 EC validă, 2 EC non-valide.
    * Situație "must be": 1 EC validă, 1 EC non-validă.
    * Dacă programul nu tratează similar elementele dintr-o EC, ECs se împart în ECs mai mici.

- **Analiza Valorilor Limită (Boundary Value Analysis - BVA)**:
  - *Definiţie*: Testare realizată prin alegerea datelor de test pe baza limitelor claselor de echivalenţă de intrare/ieşire. Investighează posibilele bug-uri existente la limita dintre ECs. O valoare limită (boundary value) este o valoare a domeniului pentru care comportamentul programului se modifică.
  - *Etape*:
    1. Identificarea condiţiilor asociate valorilor limită (pentru limitele ECs valide). Se scriu condiții BVA pentru fiecare limită: valoare sub limită, pe limită, deasupra limitei.
    2. Clasificarea condițiilor BVA în valide și non-valide.
    3. Proiectarea cazurilor de testare: se aleg date de test pentru fiecare condiţie limită identificată. Se scrie un caz de testare pentru cât mai multe condiții BVA valide neacoperite și câte un caz de testare pentru fiecare condiție BVA non-validă neacoperită.
  - *Reguli de proiectare TCs pentru BVA* :
    * Interval [a,b]: TCs pentru $a, a+1, b-1, b$ (valide) și $a-1, b+1$ (non-valide).
    * Mulțime ordonată: TCs pentru primul/ultimul element (valide); valoarea imediat $<$ minimă și imediat $>$ maximă (non-valide).
    * Număr de valori (e.g., 1-5): TCs pentru min/max (valide, e.g., 1, 5); valoarea imediat $<$ minimă și imediat $>$ maximă (non-valide, e.g., 0, 6).
- **ECP vs. BVA** :
  - *ECP*: Presupune comportament similar în EC; se alege orice valoare reprezentativă din EC; ECs pentru condiții valide/non-valide; obiectiv: verificarea specificațiilor pentru valori uzuale (building confidence).
  - *BVA*: Valorile sunt prelucrate individual; valorile sunt la limitele ECs; se consideră valori pe limită, imediat inferioare/superioare; se consideră date de intrare/ieşire pentru fiecare EC validă; obiectiv: căutarea bug-urilor uzuale (bug hunting).

2.2.3 Aplicarea ECP şi BVA pentru probleme concrete

Aplicarea ECP și BVA se face urmând etapele și regulile definite mai sus. Exemple concrete sunt prezentate în Curs 02B (paginile 11-14 pentru ECP, paginile 23-26 pentru BVA), demonstrând identificarea claselor/limitelor și derivarea cazurilor de testare pentru câmpuri de tip dată calendaristică (lună) și valori numerice (depozit bancar) .

2.2.4 Avantaje şi dezavantaje BBT

- **Avantaje Testare Black-Box**:
  - Nu necesită informaţii despre implementare.
  - Activitatea testerului este independentă de cea a programatorului.
  - Reflectă punctul de vedere al utilizatorului.
  - Surprinde ambiguităţile sau inconsistenţele din specificaţii.

- Poate începe imediat după finalizarea specificaţiilor.

- **Dezavantaje Testare Black-Box**:
  - Dacă specificaţia nu este clară, este dificil de construit cazuri de testare.
  - Multe drumuri din graful de execuţie pot rămâne netestate, putând conţine bug-uri neidentificate.
  - Doar un număr foarte mic de date de intrare va fi efectiv testat.

---

3 Curs 03. Testare White-Box

3.1 Testare White-Box: definiţie, caracteristici, avantaje şi dezavantaje

- **Definiţie (Criteriul cutiei transparente, White-Box testing, Logic driven testing)**: Testare structurală. Datele de intrare se aleg pe baza instrucţiunilor care trebuie executate, programul fiind văzut ca o cutie transparentă. Presupune acces la structura internă a programului (codul sursă). Permite identificarea situaţiilor în care execuţia programului nu acoperă diferite structuri ale acestuia.

- **Caracteristici**: Se bazează pe analiza structurii interne a codului. Tehnicile includ acoperirea fluxului de control și a fluxului de date.

- **Avantaje**:
  - Cazurile de testare sunt proiectate pe baza structurii interne a codului.
  - Identifică disfuncţionalităţi în execuţia anumitor secvenţe de cod.
  - Permite acoperirea cu teste a codului scris.

- **Dezavantaje**:
  - Nu poate testa cerinţe neimplementate sau identifica bug-uri în codul lipsă.
  - Proiectarea cazurilor de testare poate începe doar după implementare.
  - Testerul trebuie să cunoască limbajul de programare.
  - Ineficientă pentru module de mari dimensiuni.

3.2 CFG (definiţie şi construire), drumuri independente (definiţie), CC (definiţie, 3 moduri de calcul)

- **Graful Fluxului de Control (Control Flow Graph - CFG)**:
  - *Definiţie*: Reprezentare grafică detaliată a unei unităţi de program, permițând vizualizarea tuturor drumurilor. Este un graf orientat cu vârfuri (noduri) ce indică structuri secvenţiale/condiţii și arce (muchii) ce indică sensul transmiterii controlului. Tipuri de vârfuri: decizie, instrucţiune/calcul, conector, intrare, ieşire.

- *Construire* :
  1. Se numerotează unic fiecare element de structură secvenţială și condiţională.
  2. Se începe de la vârful de intrare.
  3. Se adaugă celelalte vârfuri și arce, evidenţiind transmiterea controlului.
  4. Toate ieşirile posibile se unesc în vârful de ieşire.

- **Drumuri Independente (Independent Path)**:
  - *Definiţie*: Orice drum în CFG care introduce cel puţin o instrucţiune nouă sau o condiţie nouă, care este executată cel puţin o dată. Mulţimea drumurilor independente formează mulţimea drumurilor de bază (basis path set). Indică numărul minim de cazuri de testare pentru ca fiecare instrucţiune să fie executată cel puţin o dată.

- **Complexitatea Ciclomatică (McCabe's Cyclomatic Complexity - CC)**:
  - *Definiţie*: Metrică software pentru măsurarea cantitativă a complexităţii logice a unui program. Permite determinarea numărului de drumuri independente din mulţimea de bază a unui CFG.
  - *3 Moduri de Calcul*:
    1. CC = numărul de regiuni din CFG. (O regiune este o zonă a CFG mărginită parţial sau total de arce și vârfuri).
    2. $CC = E - N + 2$, unde E = numărul de arce, N = numărul de vârfuri.
    3. $CC = P + 1$, unde P = numărul de vârfuri condiţie (vârfuri de decizie).

3.3 Construirea CFG, determinarea drumurilor independente şi calculul CC (3 moduri) pentru metode concrete

Aceste activități se realizează urmând definițiile și pașii descriși anterior.

- **Construirea CFG**: Se urmează pașii de la 1 la 4 din secțiunea de construire CFG . (Exemple grafice în Curs 03, paginile 16-18).

- **Determinarea drumurilor independente**: Se poate utiliza Algoritmul lui McCabe :

  1. Se alege un drum iniţial (D1), preferabil cu cât mai multe decizii.
  2. Pentru D2, se modifică rezultatul primei decizii de pe D1, păstrând numărul de decizii.
  3. Pentru D3, se modifică rezultatul celei de-a doua decizii de pe D1.
  4. Se repetă până toate deciziile de pe D1 au fost inversate.
  5. Se reiau pașii considerând D2 ca drum inițial, etc., până se obțin toate drumurile independente.

  (Exemple în Curs 03, paginile 20-21).

- **Calculul CC (3 moduri)**: Se aplică formulele CC = regiuni, $CC = E - N + 2$, sau $CC = P + 1$ pe CFG-ul construit. (Exemple în Curs 03, paginile 24-25).

3.4 Criteriile de acoperire apc, sc, dc, cc, dcc, mcc şi Ic

- **Acoperirea Tuturor Drumurilor (All Path Coverage - APC)**: Testarea tuturor drumurilor programului. Dificil de realizat pentru programe cu structuri repetitive.
- **Acoperirea Instrucţiunilor (Statement/Line/Node Coverage - SC)**: Proiectarea cazurilor de testare astfel încât toate instrucţiunile (fiecare vârf al CFG) sunt executate cel puţin o dată. Cel mai slab criteriu de acoperire.
- **Acoperirea Deciziilor (Branch/Edge/Decision Coverage - DC)**: Proiectarea cazurilor de testare astfel încât fiecare arc de decizie (fiecare rezultat posibil al unei decizii - true/false) să fie parcurs cel puţin o dată. $DC \implies SC$.
- **Acoperirea Condiţiilor (Condition Coverage - CC)**: Proiectarea cazurilor de testare astfel încât fiecare condiţie atomică din fiecare decizie ia fiecare dintre valorile logice posibile (true/false) cel puţin o dată.
- **Acoperirea Deciziilor şi Condiţiilor (Decision and Condition Coverage - DCC)**: Proiectarea cazurilor de testare astfel încât fiecare condiţie atomică din fiecare decizie ia toate valorile posibile cel puţin o dată, ŞI fiecare decizie ia toate valorile posibile cel puţin o dată. $DCC \implies CC$ și $DCC \implies DC$.
- **Acoperirea Condiţiilor Multiple (Multiple Condition Coverage - MCC)**: Proiectarea cazurilor de testare astfel încât toate combinaţiile posibile ale valorilor de ieşire ale condiţiilor atomice dintr-o decizie să fie parcurse cel puţin o dată. $MCC \implies DCC$.
- **Acoperirea Buclelor (Loop Coverage - LC)**: Proiectarea cazurilor de testare astfel încât structurile repetitive să fie iterate de un număr variabil de ori. Se consideră diferite scenarii: omiterea buclei (0 parcurgeri), 1 parcurgere, 2 parcurgeri, $m < n$ parcurgeri, $n - 1$ parcurgeri, n parcurgeri, $n + 1$ parcurgeri (pentru bucle simple). Strategii specifice pentru bucle imbricate, concatenate și nestructurate .
- **Relații între criterii** (ierarhie, de la cel mai puternic la cel mai slab, deși nu e strict liniară în toate cazurile): APC (cel mai puternic, adesea impracticabil) $\supseteq$ MCC $\supseteq$ DCC. DCC $\supseteq$ DC și DCC $\supseteq$ CC. DC $\supseteq$ SC (cel mai slab dintre acestea). Relația dintre CC și DC nu este de incluziune directă în ambele sensuri fără DCC.

3.5 Testare Black-Box vs. Testare White-Box

- **Testare Black-Box** :
  - Testare funcţională/comportamentală.
  - Cazuri de testare bazate pe specificaţii, fără acces la cod.
  - Surprinde ambiguităţi în specificaţii.
  - Activitate independentă de programator; testele pot fi proiectate înainte de finalizarea codului.
  - Eficientă pentru module mari.

- **Testare White-Box** :
  - Testare structurală.
  - Cazuri de testare bazate pe structura internă a codului.
  - Nu poate testa cerinţe neimplementate.
  - Proiectarea testelor începe după implementare.
  - Ineficientă pentru module mari (costisitoare construcția CFG, CC).

---

4 Curs 04. Niveluri de testare

4.1 Niveluri de testare. Definiţii şi caracteristici

Un nivel de testare reprezintă o serie de activităţi de testare asociate unei etape din procesul de dezvoltare a produsului soft. Clasificare :

- **Testare Unitară (Unit Testing / Module Testing)**:
  - *Definiţie*: Testarea individuală a unor unități separate dintr-un sistem software (funcție, procedură, clasă, metodă). Se realizează în etapa de implementare/codificare.
  - *Caracteristici/Motivație*: Gestionare eficientă a modulelor, proces de depanare eficient la nivel de modul, permite paralelizarea testării. Se folosesc obiecte de tip Test Doubles (dummy, fake, stub, spy, mock) . Contextul de testare include module *driver* (apelant) și *stub* (apelat simulat).
- **Testare de Integrare (Integration Testing)**:
  - *Definiţie*: Nivel de testare în care modulele individuale se combină şi se testează ca un grup. Permite construirea structurii programului pe măsură ce este testat pentru a identifica erorile la nivelul interfeţei dintre module. Se realizează în etapa de proiectare.
  - *Caracteristici/Motivație*: Module diferite sunt implementate de programatori diferiţi; testarea unitară e izolată; unele module pot genera mai multe defecţiuni.
- **Testare de Sistem (System Testing)**:
  - *Definiţie*: Verifică dacă produsul dezvoltat respectă obiectivele iniţiale. Se realizează în etapa de specificare a cerințelor sistemului.
  - *Caracteristici*: Elaborarea cazurilor de testare se bazează pe analiza obiectivelor și documentația de utilizare. Include testare funcţională și non-funcţională.
- **Testare Funcţională (Functional Testing - ca parte a Testării de Sistem)**:
  - *Definiţie*: Testarea cerinţelor descrise în specificaţiile sistemului. Identifică neconcordanţele între comportamentul programului şi specificaţia acestuia din punctul de vedere al utilizatorului.

  - *Caracteristici*: Elaborarea cazurilor de testare se bazează pe criteriul black-box și folosește specificația sistemului. Cazurile de testare sunt derivate din cazuri de utilizare.

- **Testare de Acceptare (User Acceptance Testing - UAT)**:
  - *Definiţie*: Proces de testare prin care se verifică dacă programul îndeplineşte cerinţele iniţiale şi nevoile curente ale utilizatorului final. Nu este responsabilitatea dezvoltatorului; testerul este clientul/beneficiarul. Se realizează în etapa de descriere a cerințelor utilizatorului.
  - *Caracteristici*: Realizată efectiv de client, aplicând tehnici black-box.

- **Alpha Testing şi Beta Testing** (tipuri de testare de acceptare):
  - *Alpha Testing*: Realizată înainte de livrare, de către client sau persoane desemnate, pe o platformă la dezvoltator, într-un mediu controlat unde programatorul poate interveni imediat. Deficiențele majore pot fi rezolvate imediat.
  - *Beta Testing*: Realizată înainte de livrare, de către client, pe platforma sa, în mediul real pentru care a fost dezvoltat, fără intervenția directă a dezvoltatorului. Furnizează feedback autentic; deficiențele sunt investigate ulterior.

4.2 Strategii de integrare (big-bang, top-down, bottom-up, sandwich), descriere, comparare

- **Integrarea Big-Bang**: Toate modulele sunt testate unitar, apoi combinate simultan pentru a construi programul.
  - *Descriere*: Simplu conceptual, dar depanare dificilă, volum mare de muncă pentru sisteme complexe.

- **Integrarea Incrementală Top-Down**: Construirea și testarea programului adăugând module noi de sus în jos, de la modulul principal. Defectele sunt uşor de izolat. Necesită stub-uri.
  - *Descriere*: Variante: în adâncime (depth-first) - integrează componentele pe o ramificație majoră; pe niveluri (breadth-first) - integrează componentele pe orizontală, pe nivelul imediat subordonat.

- **Integrarea Incrementală Bottom-Up**: Construirea și testarea programului pornind de la modulele atomice (terminale). Defectele modulelor inferioare sunt uşor de izolat. Necesită drivere pentru modulele terminale/clusteri.
  - *Descriere*: Modulele terminale se pot organiza în grupuri (clusteri). Driverele sunt înlocuite cu modulele de nivel superior pe măsură ce integrarea avansează.

- **Integrarea Sandwich (Mixtă)**: Combină abordările top-down și bottom-up.
  - *Descriere*: Modulele terminale (bottom-layer) sunt integrate bottom-up; modulul principal (top-layer) este integrat top-down; modulele intermediare (middle-layer) sunt integrate prin big-bang (sau incremental).

- **Compararea Strategiilor de Integrare**:

Criteriu	**Big-bang**	**Top-down**	**Bottom-up**	**Sandwich**
Integrare	Târzie	Timpurie	Timpurie	Timpurie
Programul final	Târziu	Devreme	Târziu	Devreme
Driver	Da	Nu	Da	Da
Stub	Da	Da	Nu	Da
Paralelizare	Mare	Mică	Medie	Medie

4.3 5 tipuri de testare non-funcţională (volume, stress, load, usability, security)

Testarea non-funcţională verifică modul în care sistemul îndeplineşte cerinţele non-funcţionale.

- **Testare de Volum (Volume Testing)**: Evaluează comportamentul sistemului atunci când se gestionează volume mari de date. Obiectiv: obţinerea de informaţii legate de funcţionarea aplicaţiei cu un volum de date ridicat (e.g., creşterea dimensiunii fişierelor). (Parte din testarea de performanță).

- **Testare de Stres (Stress Testing)**: Investighează dacă comportamentul softului se degradează în condiţii extreme de utilizare și când nu are acces la resursele necesare. Obiectiv: obţinerea de informaţii despre modul în care sistemul funcţionează în condiţii extreme, dincolo de limitele normale (e.g., creșterea numărului de fișiere până la eșec). (Parte din testarea de performanță).

- **Testare de Încărcare (Load Testing)**: Evaluează capacitatea sistemului de a opera cu volume de date normale sau în condiţii de solicitare (vârf). Obiectiv: obţinerea de informaţii referitoare la comportamentul sistemului în anumite condiţii de utilizare (e.g., creșterea numărului de fișiere folosite). (Parte din testarea de performanță).

- **Testare de Utilizabilitate (Usability Testing)**: Verifică ușurința cu care utilizatorii pot folosi aplicația, eficiența și satisfacția utilizatorului.

- **Testare de Securitate (Security Testing)**: Presupune simularea unor atacuri la nivel de securitate, având scopul de a identifica vulnerabilităţile softului. Verifică protecția datelor și a sistemului împotriva accesului neautorizat, confidențialitatea, integritatea, disponibilitatea.

4.4 Tip de testare vs. Nivel de testare. Definiţii şi caracteristici

- **Nivel de Testare**: O serie de activităţi de testare asociate unei etape din procesul de dezvoltare. Răspunde la întrebarea "Ce testez?".

- **Tip de Testare**: Mijlocul prin care un obiectiv al testării, stabilit anterior pentru un nivel de testare, poate fi realizat. Răspunde la întrebarea "Cum testez?".

- **Re-testare (Re-testing, Confirmation Testing)**:

  – *Definiţie*: Re-execuţia testelor care au pus în evidenţă anterior un bug ce se presupune că a fost eliminat.

  - *Scop*: Confirmarea că defectul a fost eliminat. Cazurile de testare sunt identice cu cele rulate anterior.

- **Testare de Regresie (Regression Testing)**:
  - *Definiţie*: Re-execuţia unor teste care au fost rulate anterior cu succes.
  - *Scop*: Identificarea efectelor secundare (bug-uri) care pot apărea în urma modificării unor module. Testele pot viza toate funcționalitățile, cele cu probabilitate mare de a fi afectate, sau componentele modificate.

- Re-testarea este diferită de testarea de regresie.

---

5 Curs 07. Instrumente utilizate in testare (Mockito) - Aspecte despre Test Doubles

Informațiile despre Test Doubles sunt prezentate în Curs 04, paginile 7-8, fiind relevante pentru testarea unitară și utilizarea Mockito.

5.1 Tipuri de obiecte utilizate în testare (Test Doubles)

Tipuri de obiecte utilizate de tool-uri, denumite generic Test Doubles:

- **Dummy Object**: Obiecte transmise ca parametri dar care nu sunt folosite de metodele apelate. Utilizate pentru a respecta signatura metodei.
- **Fake Object**: Obiecte cu implementări funcţionale/utilizabile, dar simpliste, neadecvate pentru livrabilul către client. Exemplu: o colecţie de date in-memory.
- **Stub**: Obiecte sau metode ale unor obiecte care furnizează rezultate prestabilite atunci când sunt apelate în cadrul unui test. Nu au altă utilitate în afara contextului testării.
- **Mock Object**: Obiecte pentru care s-a stabilit un anumit comportament (behavior expectations) şi sunt utilizate pentru a observa interacţiunea cu obiectul supus testării.
- **Spy**: Obiecte stub care pot păstra/reţine informaţii referitoare la modul în care au fost folosite. Exemplu: un serviciu de e-mail care reţine numărul de mesaje transmise.

(Diagrama tipurilor de Test Doubles se găsește în Curs 04, pagina 7 și în PDF-ul de tematică la pagina 7).

5.2 Diferenţe între obiectele Test Doubles

Diferențele principale constau în scopul și modul lor de interacțiune în cadrul testelor:

- **Dummy objects** sunt doar umpluturi pentru lista de parametri și nu sunt utilizate activ.
- **Fake objects** au implementări funcționale simplificate, utile pentru a rula teste fără a depinde de componente complexe (ex. baze de date reale).
- **Stubs** furnizează răspunsuri predefinite la apeluri, permițând testarea unei unități în izolare prin controlul dependențelor indirecte. Sunt folosite pentru a controla starea și fluxul execuției.
- **Spies** sunt stub-uri care înregistrează informații despre cum au fost apelate (ex. ce metode, cu ce parametri, de câte ori), permițând verificarea interacțiunilor indirecte.
- **Mocks** sunt obiecte care specifică așteptări despre cum ar trebui să fie apelate. Testul va eșua dacă aceste așteptări nu sunt îndeplinite. Sunt folosite pentru a verifica comportamentul și interacțiunile dintre obiecte.

---

6 Curs 08. Corectitudine

6.1 Metoda lui Floyd (metoda aserțiunilor inductive)

6.1.1 Elementele necesare construirii condiţiilor de verificare şi condiţiilor de terminare

Metoda aserțiunilor inductive (Floyd) se aplică pentru a demonstra parțial corectitudinea, terminarea și total corectitudinea unui program. Utilizează precondiții, postcondiții și algoritmul în sine. Etapele generale includ identificarea punctelor de tăietură, a aserțiunilor inductive și construirea/demonstrarea condițiilor de verificare/terminare.

Pentru Parțial Corectitudine :

1. **Puncte de tăietură**: Se aleg în cadrul algoritmului, incluzând un punct la început, unul la sfârșit, și cel puțin un punct în fiecare buclă.
2. **Predicate invariante (Aserțiuni inductive)** $P_i(X, Y)$: Se alege câte un predicat pentru fiecare punct de tăietură. Pentru punctul de intrare $\varphi(X)$, pentru cel de ieșire $\psi(X, Z)$, și pentru punctele intermediare (în bucle) $\eta(X, Y)$. X sunt variabile de intrare, Y variabile intermediare, Z variabile de rezultat.
3. **Drumuri între punctele de tăietură** ($\alpha_{i,j}$ sau d_{ij}): Secvențe de instrucțiuni între două puncte de tăietură i și j care nu conțin alte puncte de tăietură.
4. **Condiţii de parcurgere a drumurilor** $R_{\alpha_{i,j}}(X, Y)$: Predicate care specifică condiția ca un drum $\alpha_{i,j}$ să fie executat.
5. **Funcţii de transformare a variabilelor** $r_{\alpha_{i,j}}(X, Y)$: Funcții care indică transformările variabilelor Y de-a lungul drumului $\alpha_{i,j}$.

6. **Condiţii de verificare**: Pentru fiecare drum $\alpha_{i,j}$ de la punctul de tăietură i (cu predicatul P_i) la punctul de tăietură j (cu predicatul P_j), se construiește și demonstrează condiția: $\forall X \forall Y (P_i(X,Y) \wedge R_{\alpha_{i,j}}(X,Y) \implies P_j(X, r_{\alpha_{i,j}}(X,Y)))$. Dacă toate condițiile de verificare sunt adevărate, programul este parțial corect.

Pentru Terminare :

1. Se aleg punctele de tăietură și predicatele invariante (ca la parțial corectitudine).
2. Se alege o mulțime M parțial ordonată, fără șiruri descrescătoare infinite (de obicei, numerele naturale $\mathbb{N}$).
3. **Funcţii descrescătoare (funcții de terminare)** $u_i(X,Y)$: Pentru fiecare punct de tăietură i din interiorul unei bucle, se asociază o funcție $u_i : D_X \times D_Y \to M$. Valoarea acestei funcții trebuie să fie non-negativă și să scadă strict la fiecare iterație a buclei.
4. **Condiţii de terminare**: Pentru fiecare drum $\alpha_{i,j}$ care formează o buclă (sau duce la progres în buclă), se demonstrează că valoarea funcției u descrește: $\forall X \forall Y (P_i(X,Y) \wedge R_{\alpha_{i,j}}(X,Y) \implies u_i(X,Y) > u_j(X, r_{\alpha_{i,j}}(X,Y)) \wedge u_j(X, r_{\alpha_{i,j}}(X,Y)) \geq 0)$ (Dacă parțial corectitudinea a fost demonstrată, P_i poate fi folosit. Altfel, se folosește precondiția programului $\varphi(X)$). Dacă toate condițiile de terminare sunt adevărate, programul se termină.

Total Corectitudine = Parțial Corectitudine + Terminare.

6.1.2 Demonstrarea parţial corectitudinii, terminării și a total corectitudinii pentru probleme concrete

Metoda lui Floyd se aplică pentru a demonstra corectitudinea unor algoritmi precum:

- Determinarea celui mai mare divizor comun a două numere naturale (problema menționată în Seminar 05).
- Căutarea unei valori într-un şir ordonat (problema menționată în Seminar 05).

(Un exemplu detaliat pentru $z = x^y$ este prezentat în Curs08.CorectitudineFloydHoare.pdf, pagina 8).

6.2 Teoria lui E. Dijkstra

6.2.1 Rafinare: definirea regulilor

Dezvoltarea algoritmilor corecți din specificații se poate face prin metoda rafinării. Un program abstract $Z : [\varphi, \psi]$ (unde φ este precondiția și ψ este postcondiția) este rescris succesiv $Z = P_0 < P_1 < \cdots < P_n$ folosind reguli de rafinare. Reguli de rafinare (conform Curs08.CorrectnessDijkstra.pdf, pagina 12):

- **Regula atribuirii**: $[\varphi(v/e), \psi] < v := e$ (unde $\varphi(v/e)$ este φ cu v înlocuit de e)
- **Regula compunerii secvenţiale**: $[\eta_1, \eta_2] < [\eta_1, \gamma]; [\gamma, \eta_2]$ (unde γ este un predicat auxiliar)

- **Regula alternanţei** (if $c_1 \to S_1 \square \ldots \square c_n \to S_n$ fi, unde $cond = c_1 \vee \cdots \vee c_n$): $[\eta_1, \eta_2] <$ if $c_1 \to [\eta_1 \wedge c_1, \eta_2] \square \ldots \square c_n \to [\eta_1 \wedge c_n, \eta_2]$ fi
- **Regula iteraţiei** (do $c_1 \to S_1 \square \ldots \square c_n \to S_n$ od, unde $cond = c_1 \vee \cdots \vee c_n$ și η este invariantul buclei, TC este condiția de terminare): $[\eta, \eta \wedge \neg cond] <$ do $c_1 \to [\eta \wedge c_1, \eta \wedge TC] \square \ldots \square c_n \to [\eta \wedge c_n, \eta \wedge TC]$ od

6.2.2 Rafinare algoritmi din specificații (2 probleme)

Metoda rafinării se aplică pentru dezvoltarea algoritmilor din specificații pentru probleme precum:

- **Împărţire întreagă (cât și rest)**:
  - *Specificare*: $\varphi : (x \geq 0) \wedge (y > 0)$, $\psi : (x = q \cdot y + r) \wedge (0 \leq r < y)$
  - *Rafinare ($A_0 : [\varphi, \psi]$)*:
  - A_1: Folosind predicatul intermediar $\eta : (x = q \cdot y + r) \wedge (0 \leq r)$ și regula compunerii secvențiale: $[\varphi, \eta]$ urmat de $[\eta, \psi]$.
  - A_2: Predicatul η devine adevărat prin atribuirea $(q, r) := (0, x)$: $(q, r) \leftarrow (0, x)$ urmat de $[\eta, \eta \wedge r < y]$.
  - A_3: Aplicarea regulii iterației pe $[\eta, \eta \wedge r < y]$ (deoarece η este invariant): $(q, r) \leftarrow (0, x)$ DO $r \geq y \to [\eta \wedge r \geq y, \eta \wedge TC]$ OD
  - A_4: Pentru terminare, r trebuie să scadă. Deoarece $r \geq y$, putem reduce r cu y, i.e., $r \leftarrow r - y$. Pentru ca η să rămână adevărat, $q \cdot y + r = (q+1) \cdot y + (r - y)$. Deci, și q se modifică: $q \leftarrow q + 1$. Algoritmul final: $(q, r) \leftarrow (0, x)$ DO $r \geq y \to (q, r) \leftarrow (q + 1, r - y)$ OD
- **Rădăcină pătrată**:
  - *Specificare*: $\varphi : n > 1$, $\psi : r^2 \leq n < (r + 1)^2$ (unde $r = \lfloor \sqrt{n} \rfloor$)
  - *Rafinare ($A_0 : [\varphi, \psi]$)*:
  - Se rescrie ψ ca $(r^2 \leq n < q^2) \wedge (q = r + 1)$.
  - A_1: Folosind predicatul intermediar $\eta : (r^2 \leq n < q^2)$ și regula compunerii secvențiale: $[\varphi, \eta]$ urmat de $[\eta, \psi]$.
  - A_2: η devine adevărat pentru $(q, r) := (n + 1, 0)$: $(q, r) \leftarrow (n + 1, 0)$ urmat de $[\eta, \eta \wedge (q = r + 1)]$. (Nota: ψ este $\eta \wedge (q = r + 1)$)
  - A_3: Aplicarea regulii iterației pe $[\eta, \eta \wedge (q = r + 1)]$ (deoarece η este invariant): $(q, r) \leftarrow (n + 1, 0)$ DO $q \neq r + 1 \to [\eta \wedge q \neq r + 1, \eta \wedge TC]$ OD
  - A_4: Pentru terminare, $q - r$ trebuie să devină 1. Se introduce $p = (q + r)/2$. Intervalul $[r, q]$ se actualizează la $[r, p]$ sau $[p, q]$. Dacă $p^2 \leq n$, atunci $r \leftarrow p$ menține invariantul η. Dacă $p^2 > n$, atunci $q \leftarrow p$ menține invariantul η. Algoritmul final: $(q, r) \leftarrow (n + 1, 0)$ DO $q \neq r + 1 \to p \leftarrow (q + r)/2$ IF $p^2 \leq n \to r \leftarrow p \; \square p^2 > n \to q \leftarrow p$ FI OD

---

7 Curs 09. Raportarea bug-urilor

7.1 Ciclul de viaţă al unui bug

7.1.1 Cele două variante discutate la curs (simplu şi detaliat)

- **Varianta Simplă** (Bug Found, Open, Resolved, Closed, Deferred):

  1. Bug Found: Testerul găsește și înregistrează bug-ul.
  2. Open: Raportul de bug este atribuit programatorului.
  3. Resolved: Programatorul repară bug-ul. Raportul este atribuit testerului.
  4. Closed: Testerul confirmă că bug-ul este reparat și închide raportul.
  5. Deferred: Repararea bug-ului este amânată.

  (Diagrama în Curs09-Raportarea bug-urilor.pdf, pagina 4)

- **Varianta Detaliată** (cu mai multe stări, conform ISTQB):

  – Stări stabilite de tester: New, Pending Testing, Retest, Reopened, Verified, Closed.

  – Stări stabilite de programator: Assigned, Open (cu sub-stări posibile: Duplicate, Rejected, Deferred, Not a Bug), Fixed.

  (Diagrama în Curs09-Raportarea bug-urilor.pdf, pagina 5)

7.2 RIMGEA

RIMGEA este un grup de reguli utilizat pentru investigarea şi îmbunătăţirea descrierii unui bug.

7.2.1 Descrierea semnificaţiei acronimului

- **R - Replicate (Reproducerea bug-ului)**: Activitate de testare prin care se identifică ce este necesar pentru ca bug-ul să apară de fiecare dată când se doreşte manifestarea lui. Se descriu și cazurile în care bug-ul nu poate fi reprodus la cerere și factorii posibili.

- **I - Isolate (Izolarea bug-ului)**: Activitate de testare prin care se identifică cea mai scurtă secvenţă de paşi necesară pentru ca bug-ul să fie reprodus şi raportat într-o formă clară, evidențiind defecţiunea. Un raport de bug prezintă o singură defecţiune.

- **M - Maximize (Maximizarea bug-ului)**: Activitate de testare prin care se identifică cele mai importante (grave) consecinţe ale existenţei bug-ului. Se prezintă consecinţele frecvente în configurații uzuale, cu date reale.

- **G - Generalize (Generalizarea bug-ului)**: Activitate de testare prin care se identifică şi se clasifică situaţiile în care acest bug va cauza o defecţiune.

- **E - Externalize (Externalizarea bug-ului)**: Activitate de testare prin care se identifică consecinţele prezenţei bug-ului din perspective diferite: utilizator, client/beneficiar, dezvoltator/furnizor, terț/competitor.

- **A - And say it clearly and dispassionately (Comunicare clară și neutră)**: Raportul unui bug conţine date concrete sau speculative într-o manieră constructivă, imparţială şi corectă. Nu se critică persoane, nu se includ aspecte irelevante decât dacă sunt esențiale.

7.3 Tipuri de bug-uri

- **Coding Bug (Bug de implementare)**: Programul funcţionează într-o manieră pe care proiectantul şi programatorul o consideră nepotrivită. Pune în evidenţă o deficienţă de scriere sau implementare a soluţiei. Comportamentul este considerat incorect de către designer, programator sau tester.

- **Design Bug (Bug de proiectare)**: Programul funcţionează conform proiectării şi implementării intenționate de designer și programator. Pune în evidenţă o deficienţă în abordarea modului de rezolvare, care reduce nejustificat calitatea softului.

- **Coding Bug vs. Design Bug** (din perspectiva aplicării RIMGEA):

RIMGEA	**Bug de implementare**	**Bug de proiectare**
Replicate	Esenţial	Rareori important
Isolate	Important	Important
Maximize	Uneori esenţial	Util
Generalize	Important	Util
Externalize	Uneori util	Esenţial
And say it clearly and dispassionately	Esenţial	Esenţial