

CURS 01A.

VERIFICARE ȘI VALIDARE

Verificarea și validarea sistemelor soft

[28 Februarie 2023]

Lector dr. Camelia Chisăliță-Crețu

Universitatea Babeș-Bolyai

Conținut

- Calitatea produselor soft
 - Stakeholders
 - Definiții ale calității produselor soft
 - Activități asociate calității
- Verificare și validare
- Defect software
 - Terminologie
 - Costul unui bug software
- Bug-uri software celebre
- Bibliografie

CALITATEA PRODUSELOR SOFT

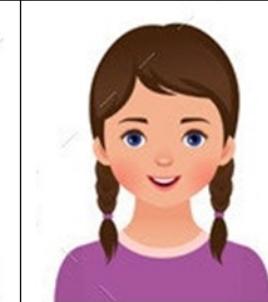
Stakeholders

Definiții ale calității produselor soft

Activități asociate controlului calității unui produs soft

Managementul publicațiilor la BJ

- O aplicație pentru gestionarea publicațiilor la Biblioteca Județeană.

				
Emil (47), Director General BJ	Clara (44), Bibliotecar	Anna (23), Bibliotecar	Alex (37), Programator, Manager firma IT	George (29), Programator Software

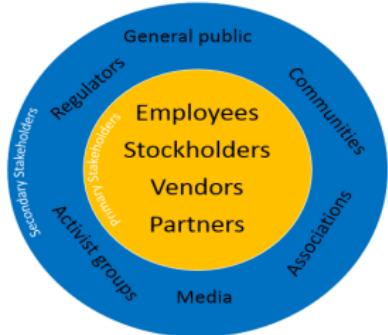
Stakeholders

- **stakeholder** (*rom. beneficiar, utilizator*)
 - o persoană care manifestă un interes particular pentru **succesul** sau **eșecul** unui produs soft [\[BBST2010\]](#).



[\[CFI2022\]](#)

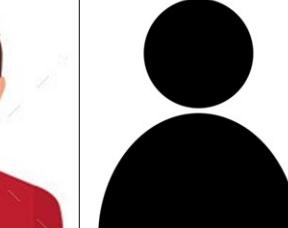
Tipuri de Stakeholders



- primar/secundar (engl., primary, secondary) [\[StakeholderMap2019\]](#):
 - **beneficiar primar** – direct afectat de succesul sau eşecul produsului;
 - **beneficiar secundar** – nu este afectat direct de succesul sau eşecul produşului.
- preferat/nedorit (engl., favored, disfavored) [\[GauseWeinberg2011\]](#), [\[KanerBach2005\]](#):
 - **beneficiar preferat (avantajat)** – produsul este proiectat pentru a fi utilizat de acesta;
 - **beneficiar nedorit (dezavantajat)** - produsul este proiectat sa creeze dificultăţi în utilizare;
 - **beneficiar neutru** – produsul nu este proiectat pentru acesta și nu îl poate influența;
 - **beneficiar ignorat (neglijat)** – produsul nu este proiectat pentru fi utilizat de acest tip de utilizator.

Exercițiu

- Clasificați următorii beneficiari pe baza categoriilor:
 - A. beneficiar primar/secundar;
 - B. beneficiar preferat/ nedorit/ neutră/ ignorat.

					
Emil (47), Director General BJ	Clara (44), Bibliotecar	Anna (23), Bibliotecar	Alex (37), Programator, Manager firma IT	George (29), Programator Software	???

Calitatea produselor soft. Definiții (1)

- ***“produsul soft este conform cu cerințele documentate”*** [[Pressman2000](#)]:
 - conformitatea cu cerințele funcționale și de performanță precizate și documentate explicit în standarde de dezvoltare și caracteristicile implicate pe care un produs soft dezvoltat le are;
- ***“produsul soft este conform cu cerințele reale ale utilizatorului”*** [[Crosby1980](#)]:
 - conformitatea cu cerințele reale ale utilizatorului care pot fi incluse sau nu în specificațiile scrise;
 - **conformitate cu cerințele (nevoile) reale, nu doar cu cerințele documentate;**



Calitatea produselor soft. Definiții (2)

- ***“produsul soft este adecvat pentru a fi utilizat”*** [[Juran1998](#)]:
 - *satisfiers* – orice aspect care îl mulțumește pe beneficiar;
 - *dissatisfiers* – orice aspect care îl nemulțumește pe beneficiar;
- ***“produsul soft este relevant/important pentru o persoană”*** [[Weinberg1992](#)]:
 - calitatea este subiectivă;
 - un aspect care are relevanță/importanță însemnată pentru un utilizator poate fi mai puțin important pentru un alt utilizator din aceeași categorie de utilizatori.



Activități asociate calității

- *în procesul de dezvoltare, calitatea este abordată din perspectiva:*
 - **procesului** ==> **asigurarea calității** (engl. quality assurance):
 - **Obiectiv:** asigură respectarea standardelor, planurilor și etapelor proceselor de dezvoltare necesare elaborării adecvate a produsului cerut;
 - **Întrebare:** Cum se asigură calitatea activităților desfășurate în procesul dezvoltare?
 - **produsului** ==> **controlul calității** (engl. quality control):
 - **Obiectiv:** identifică deficiențele în produsul obținut;
 - **Întrebare:** Cum se controlează calitatea rezultatelor obținute (e.g., work products) în urma activităților desfășurate?

Asigurarea calității

- **Prevenție** bug-uri
- Orientare pe **proces**
- Planificarea și monitorizarea activităților

Controlul calității

- **Detectie** bug-uri
- Orientare pe **produs**
- Căutare și eliminare bug-uri

Activități asociate controlului calității

Analiza statică

- examinarea unor documente (specificații, modele conceptuale, diagrame de clase, cod sursă, planuri de testare, documentații de utilizare);
- **exemple:** activități de inspectare a codului, analiza algoritmului, demonstrarea corectitudinii;
- **NU presupune execuția propriu-zisă a programului dezvoltat;**

- metode de analiză complementare;
- dezvoltatorii aplică metode hibride, care folosesc avantajele celor două abordări.

Analiza dinamică

- examinarea comportamentului programului cu scopul de a evidenția defecțiuni posibile;
- **exemple: tipuri de testare** (de regresie, funcțională, non-funcțională), **niveluri de testare** (testare unitară, testare de integrare, testare de sistem, testare funcțională, testare de acceptare);
- **include activitatea de execuție propriu-zisă a programului (testare);**

VERIFICARE ȘI VALIDARE

Verificare

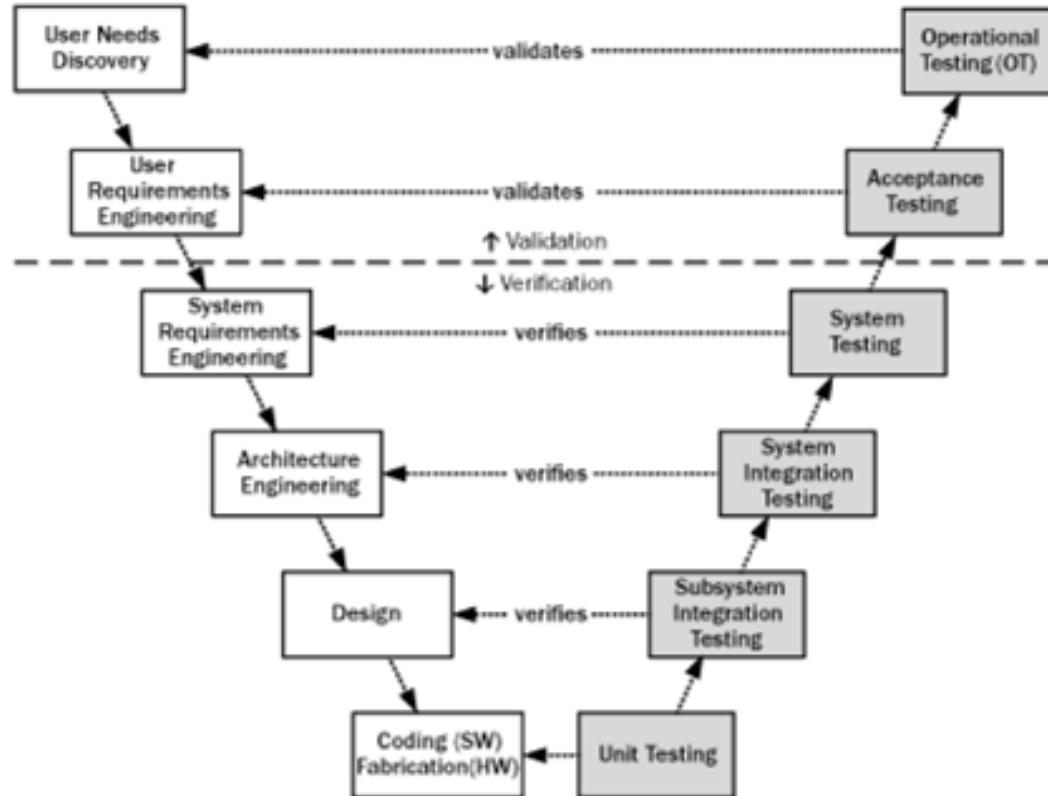
Validare

Verificare vs. Validare

Verificare și Validare. Definiție SEI

- SEI (Software Engineering Institute) [[NT2005](#)]
- **Verificare**
 - procesul prin care se asigură că produsul este dezvoltat conform cerințelor, specificațiilor și standardelor;
 - întrebare asociată: **Dezvoltăm corect produsul?** (*Are we building the product right?*)
- **Validare**
 - procesul prin care se asigură că produsul dezvoltat satisfac cerințele utilizatorului;
 - întrebare asociată: **Dezvoltăm produsul corect (de care are nevoie clientul)?**
(Are we building the right product?)

Verificare și Validare în modelul V



• sursa: [\[Firesmith2015\]](#)

Verificare vs. Validare

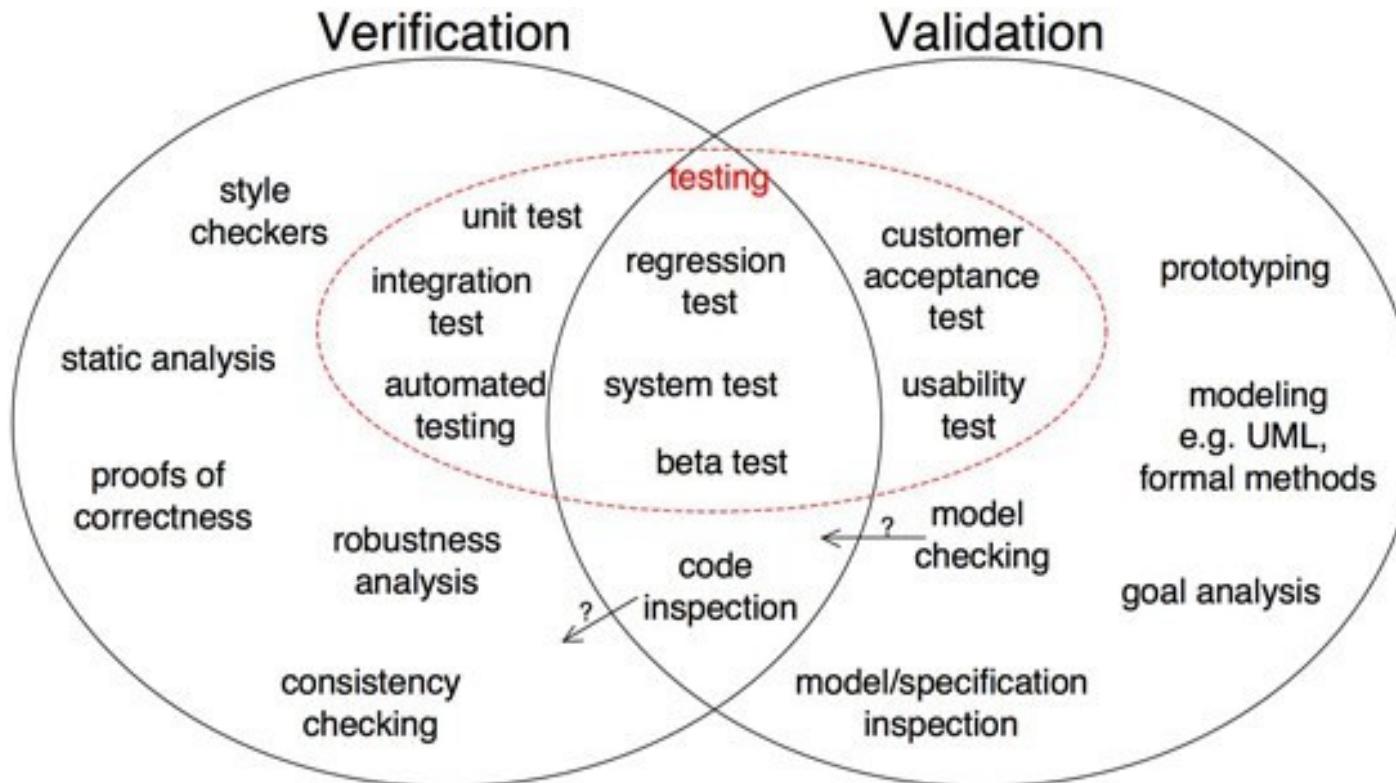
Verificare

- stabilește dacă rezultatul unei etape de dezvoltare satisface cerințele acelei etape;
- asigurare a consistenței, completitudinii, corectitudinii;
- **aplică metode de control al calității;**

Validare

- **confirmă că produsul satisface cerințele de utilizare;**
- se desfășoară spre sfârșitul procesului de dezvoltare, cu scopul de a demonstra că întregul sistem satisface nevoile și așteptările;
- se aplică asupra întregului sistem, în contextul real în care va funcționa, folosind diferite tipuri de testare.

Activități de Verificare și Validare



• sursa: [[Easterbrook2010](#)]

DEFECT SOFTWARE

Terminologie

Când apare un bug într-un produs soft?

De ce apare un bug în procesul de dezvoltare software?

Costul unui bug software

Defecte/Buguri software celebre

Terminologie (1)

- **eroare** (*engl. error, mistake*; greșeală):
 - o acțiune umană care are ca rezultat un defect în produsul software [[Patton2005](#)];
- **defect** (*engl. fault*, i.e., **bug**):
 - consecință a unei erori [[Patton2005](#)];
 - 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;
- **defecțiune** (*engl. failure*):
 - devierea de la comportamentul obișnuit 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 defecțiune [[Patton2005](#)];

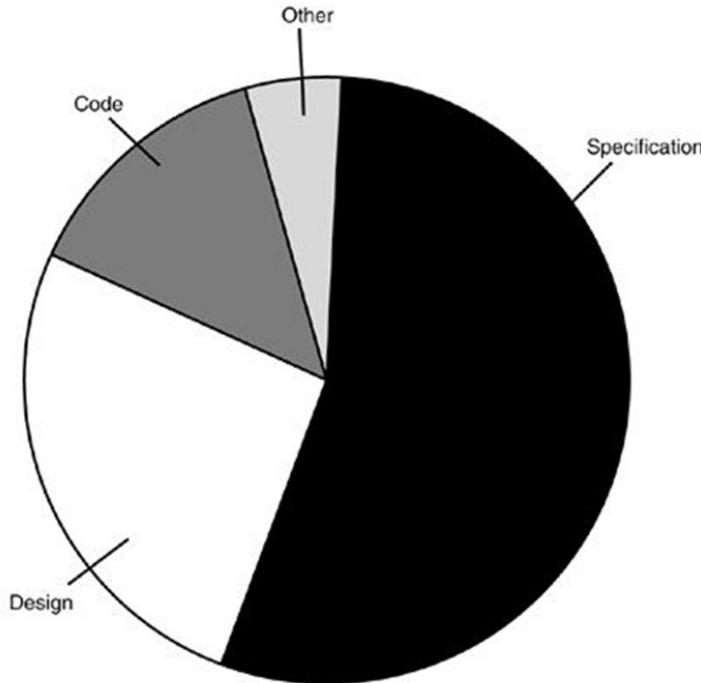
Terminologie (2)

- **defect** (*engl. bug, software error*)
 - orice aspect al unui produs soft care
 - cauzează reducerea inutilă și inadecvată a calității produsului soft [\[BBST2008\]](#);
 - constituie o amenințare asupra imaginii produsului [\[BBST2008\]](#);
 - exemple: deficiențe de proiectare, greșeli în documentații, utilizare cu dificultate a programului;
 - totuși, anumite aspecte ale produsului pot limita calitatea acestuia, dar nu pot fi considerate defecte!
 - exemplu: constrângerile de utilizare precizate sau nu în specificații;
 - În cadrul acestui curs, orice deficiență sau problemă a produsului soft este denumită **bug (defect)**.
 - sinonime pentru bug: *engl. variance, problem, inconsistency, error, incident, anomaly* [\[Patton2005\]](#).

De ce apare un bug într-un produs soft?

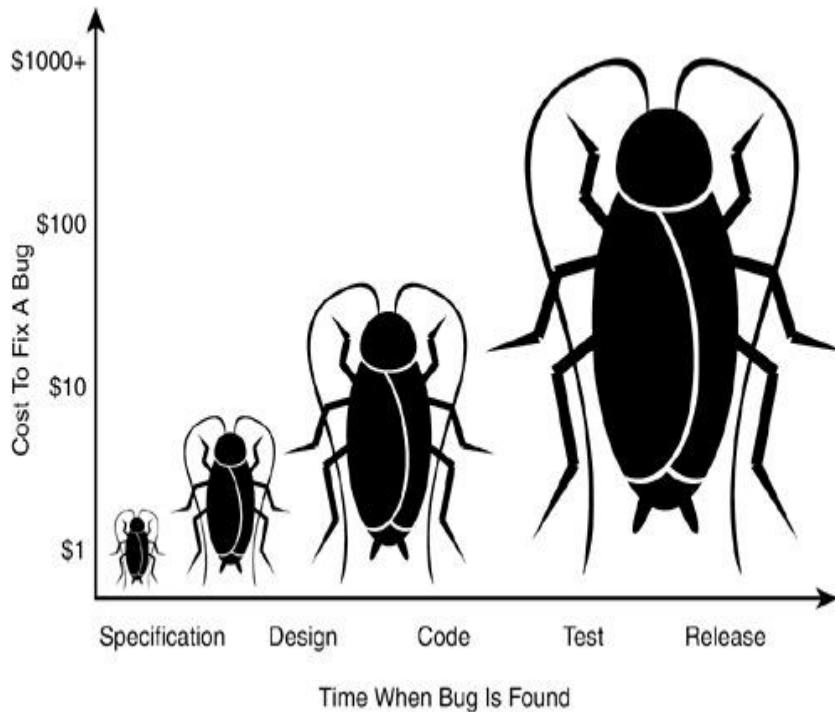
- Un bug software apare atunci când cel puțin una din următoarele situații are loc [\[Patton2005\]](#):
 - Produsul soft nu face ce este precizat în specificația lui.
 - Produsul soft face ce nu este precizat în specificație.
 - Produsul soft face ce specificația precizează că **nu** trebuie făcut.
 - Produsul soft nu face ceea ce specificația ar trebui să precizeze.
 - Produsul soft este dificil de înțeles, greu de utilizat, lent. Testerul pune în evidență perspectiva utilizatorului final asupra produsului soft, adică produsul nu funcționează conform așteptărilor lui.

În ce etapă a procesului de dezvoltare software apar bug-urile?



- **specificarea cerințelor:**
 - nu se scriu specificațiile, sunt superficiale, se schimbă continuu, nu sunt comunicate corespunzător întregii echipe de dezvoltare;
- **proiectare:**
 - sunt superficiale, nu se comunică eficient, se modifică;
- **implementare:**
 - complexitatea produsului soft, lipsa documentației (pentru codul sursă îmbunătățit), erori de redactare, presiunea termenului limită.
- **Care este etapa de dezvoltare în care se introduc cele mai multe defecte?**

Cât costă eliminarea unui bug?



- **Care sunt costurile de eliminare a unui bug software?**
- costul eliminării bug-urilor crește pe măsură ce produsul soft este dezvoltat.

BUG-URI SOFTWARE CELEBRE

Activitate de seminar

9+ bug-uri software celebre

Activitate de seminar. Bug Poster

- **Bug Poster**
 - **CE?** Descrieți un bug faimos (celebru) într-un poster (**1 pagină A4, portret/landscape, Ro/En**);
 - **CUM?** Elementele posterului: denumirea bug-ului, anul apariției bug-ului, descrierea contextului (a aplicației) în care a apărut bug-ul, descrierea bug-ului (pe scurt), consecințele (impactul) apariției bug-ului din diferite perspective (costuri de depanare, scăderea credibilității, etc.), o imagine sugestivă a bug-ului;
 - **CINE?** Perechi de 2 studenți; înscrierea se face în fișierul de la acest [link](#); **la completarea datelor, studenții sunt rugați să se asigure că bug-ul propus nu este ales deja de alți colegi care apar anterior în listă;**
 - **CÂND?** Posterul va fi prezentat în timpul orelor de seminar la grupa din care fac parte membrii echipei; **1 poster/seminar**;
 - **CÂT?** **temp alocat: max. 5 minute/poster**; după prezentare, poster-ul va fi încărcat în MS Teams, în channel-ul **BugPosters**, secțiunea **Files**;
 - **DE CE?** Studenții primesc **max. 2 puncte de activitate** pentru seminarul în cadrul căruia are loc prezentarea posterului.

Buguri software celebre (1)

- **Naveta spațială Mariner 1 – 1962**
 - naveta spațială Mariner 1 a deviat de la traекторia ei la scurt timp după lansarea spre planeta Venus; a fost distrusă la 293 secunde după lansare;
 - **cauza:** eroare la scrierea unei instrucțiuni în limbajul FORTRAN, determinând calculul eronat al traiectoriei;
 - **cost:** 18.5 milioane \$

DO 10 I=1.10

.....

- compilatorul Fortran ignoră spațiile, iar instrucțiunea a fost considerată corectă; astfel:

DO10I = 1.10 ---> se inițializează o variabilă nedeclarată

- intenția programatorului a fost:

DO 10 I = 1, 10

.....



Buguri software celebre (2)

- **Tratamente împotriva cancerului – 1985**
 - dispozitivul Therac-25 fost folosit în terapia prin radiații;
 - **cauza:** programul a calculat greșit doza de radiații pe baza datelor de intrare, unii pacienți primind o doză de câteva ori mai mare decât cea normală;
 - **cost:** 3 pacienți decedați, 3 răniți prin iradiere.



Buguri software celebre (3)

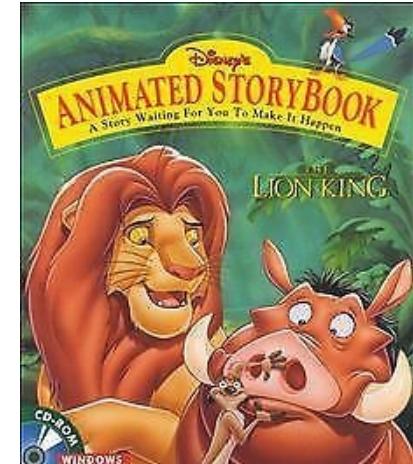
- **Sistemul de apărare american anti-rachetă – 1991**

- sistemul american de apărare antirachetă MIM-104 Patriot situat în Arabia Saudită nu a reușit să detecteze atacuri cu rachete Scud irakiene;
- **cauza:** o eroare de rotunjire la ceasul sistemului (un sfert de secundă) s-a cumulat, astfel încât la 14 ore, sistemul de urmărire își pierdea acuratețea, devenind incapabil să localizeze și să intercepteze rachetele;
- **cost:** în atacul asupra unei cazarme din Dhahran au decedat 28 soldați americani;
- eroarea fusese deja remediată de expertii armatei americane, iar noua versiune a softului urma să ajungă cu o zi mai târziu.



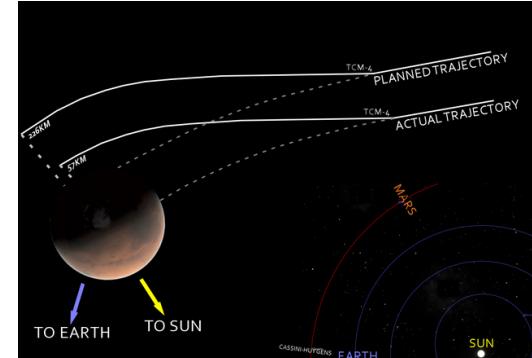
Buguri software celebre (4)

- **Jocul asociat desenului animat Disney Lion King – 1995**
 - la prima apariție pe piață a companiei Disney cu un joc pentru copii - *The Lion King Animated Storybook* - unii utilizatori nu au reușit să folosească produsul soft achiziționat;
 - **cauza:** compania Disney nu a testat produsul pe diferite modele de calculatoare personale existente pe piață;
 - **cost:** credibilitatea companiei, schimbarea unităților CD-ROM.



Buguri software celebre (5)

- **Naveta spațială Mars Climate Orbiter – 1998**
 - **obiectiv:** orbitarea planetei Marte și transmiterea informațiilor despre condițiile meteo;
 - **eveniment:** după o călătorie de 286 zile de pe Pământ, la intrarea în atmosfera planetei Marte, motoarele au deviat traекторia navetei;
 - **rezultat:** dezintegrarea navetei în atmosferă;
 - **cauza:** două dintre echipele implicate în dezvoltarea aplicației foloseau sisteme de măsurare a distanței diferite, imperial (**inch, feet**) și cel metric (**m, km**).



Buguri software celebre (6)

- **Naveta spațială Mars Polar Lander – 1998**

- **obiectiv:** studierea solului și a climei din regiunea Planum Australe de pe Marte;
- pentru mecanismul de identificare a momentului când mototarele trebuie să fie opriate, NASA nu a folosit radare costisitoare, ci un senzor pe talpa picioarelor navetei, care determina oprirea alimentării cu combustibil;
- **eveniment:** la intrarea în atmosfera planetei Marte, programul a interpretat vibrațiile navetei – cauzate de turbulențele din atmosferă – că aceasta ar fi aterizat și a oprit motoarele navetei;
- **rezultat:** prăbușirea navetei de înălțimea de 40m față de suprafața planetei Marte;
- **cauza:** testare incompletă – procedura de aterizare a fost împărțită în două etape, care au fost testate independent; nu s-a realizat testarea de integrare.



Buguri software celebre (7)

- **Knight Capital Group – 2012**
 - casa de brokeraj Knight Capital Group a suferit o pierdere consistentă la bursa din New York;
 - **cauza:** sistemul a introdus pe bursa de la New York tranzacții care au provocat fluctuații violente ale prețurilor multor acțiuni;
 - **cost:** pierderi de 440 milioane \$ în doar 45 minute.



PHOTO: STAN HONDA/AFP/GTETTY IMAGES

Buguri software celebre (8)

- **Termostatul Nest – 2016**

- termostatul Nest Learning Thermostat (achiziționat de Google în 2014 pentru 3.2 mld \$) nu a permis controlul temperaturii în locuințele în care a fost instalat – imposibilitatea de a-l utiliza pentru încălzire sau prepararea apei calde în timpul unui weekend friguros;
- **cauza:** update-ul de firmware pentru device împreună cu existența unor filtre necurățate și centrale termice incompatibile; acești factori au dus la descărcarea bateriei device-ului.



Buguri software celebre (9)

- **Beresheet („In the beginning...”)** – 2019
 - în 11 aprilie 2019 a avut loc tentativa eşuată a Israelului de a trimite pe Lună o naveta spațială fără oameni a bord;
 - **cauza:** un bug la sistemul de control al motorului care l-a împiedicat să reducă viteza în timpul aselenizării;
 - inginerii au încercat să corecteze bug-ul de la distanță prin restartarea motorului, dar la preluarea controlului asupra motorului era prea târziu pentru ca Beresheet să poată fi încetinită și să se dezintegreze la prăbușire.



Referințe bibliografice

- [Firesmith2015] Donald Firesmith, *Four Types of Shift Left Testing*, https://insights.sei.cmu.edu/sei_blog/2015/03/four-types-of-shift-left-testing.html
- [NT2005] K. Naik and P. Tripathy. *Software Testing and Quality Assurance*, Wiley Publishing, 2005.
- [NASA] NASA, <https://www.grc.nasa.gov/www/wind/valid/tutorial/glossary.html>.
- [Crosby1980] Philip B. Crosby, *Quality Is Free*, Signet Shakespeare, 1980.
- [Juran1998] A. Blanton Godfrey, Joseph Juran, *JURANS QUALITY HANDBOOK*, McGraw-Hill, 1998.
- [Weinberg1992] Gerald Weinberg, *Quality Software Management , Vol. 1: Systems Thinking*, Dorset House Publishing, 1992.
- [Pressman2000] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, Inc., 2000.
- [BBST] BBST – Bug Advocacy Course,
[http://testingeducation.org/BBST/\(http://testingeducation.org/BBST/bugadvocacy/BugAdvocacy2008.pdf](http://testingeducation.org/BBST/(http://testingeducation.org/BBST/bugadvocacy/BugAdvocacy2008.pdf).
- [Patton2005] R. Patton, *Software Testing*, Sams Publishing, 2005.
- [Easterbrook2010] S. Easterbrook, *Software Testing*,<http://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation/>.
- [CFI2022] Stakeholders, <https://corporatefinanceinstitute.com/resources/knowledge/finance/stakeholder/>.
- [StakeholderMap2019] Stakeholders, <https://www.stakeholdermap.com/primary-stakeholders.html>.
- [GauseWeinberg2011] Donald C. Gause, Gerald M. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House, 2011.
- [KanerBach2005] Kaner, C., Bach, J., Requirements Analysis for Test Documentation,
<http://www.testingeducation.org/BBST/extras/BBSTTestDocs2005.pdf>.

CURS 01B.

INSPECTARE

Verificarea și validarea sistemelor soft
[28 Februarie 2023]

Lector dr. Camelia Chisăliță-Crețu
Universitatea Babeș-Bolyai

Conținut

- Calitatea produselor soft
 - Activități asociate controlului calității
 - Analiza statică. Clasificare
- Metode bazate pe factorul uman
 - Definiție. Motivație. Caracteristici
 - Inspectare Fagan
 - Walkthroughs
 - Technical Review
 - Pair-Programming
- Pentru examen...
- Bibliografie

CALITATEA PRODUSELOR SOFT

Activități asociate controlului calității unui produs soft

Analiza statică. Clasificare

Activități asociate calității

- *în procesul de dezvoltare, calitatea este abordată din perspectiva:*
 - **procesului** ==> **asigurarea calității** (engl. quality assurance):
 - **Obiectiv:** asigură respectarea standardelor, planurilor și etapelor proceselor de dezvoltare necesare elaborării adecvate a produsului cerut;
 - **Întrebare:** Cum se asigură calitatea activităților desfășurate în procesul dezvoltare?
 - **produsului** ==> **controlul calității** (engl. quality control):
 - **Obiectiv:** identifică deficiențele în produsul obținut;
 - **Întrebare:** Cum se controlează calitatea rezultatelor obținute (e.g., work products) în urma activităților desfășurate?

Asigurarea calității

- **Prevenție** bug-uri
- Orientare pe **proces**
- Planificarea și monitorizarea activităților

Controlul calității

- **Detectie** bug-uri
- Orientare pe **produs**
- Căutare și eliminare bug-uri

Controlul calității. Activități asociate (1)

Analiză statică (static testing)

- examinarea unor documente (specificații, modele conceptuale, diagrame de clase, cod sursă, planuri de testare, documentații de utilizare);
- **exemple:** activități de inspectare a codului, analiza algoritmului, demonstrarea corectitudinii;
- se pot baza pe factorul uman (reviews) sau utilizarea tool-urilor (analiza statică).

Analiză dinamică (dynamic testing)

- examinarea comportamentului programului cu scopul de a evidenția defecțiuni posibile;
- **exemple:** *tipuri de testare* (de regresie, funcțională, non-funcțională), *niveluri de testare* (testare unitară, testare de integrare, testare de sistem, testare funcțională, testare de acceptare);
- se bazează întotdeauna pe execuția programului.

Controlul calității. Activități asociate (2)

Analiză statică (static testing)

- permit identificarea mai multor erori (greșeli) care pot fi corectate simultan;
- **NU** presupune execuția propriu-zisă a programului dezvoltat;

Analiză dinamică (dynamic testing)

- sugerează doar un simptom, fiecare eroare identificată fiind eliminată individual;
- include activitatea de execuție propriu-zisă a programului (testare);
- poate să evidențieze o defecțiune doar în anumite situații.

- metode de analiză complementare;
- dezvoltatorii aplică metode hibride, care folosesc avantajele celor două abordări.

Analiză statică

- preconcepție (anii '60) –
 - „*singura modalitate de a verificare a unui program este execuția pe calculator*” [[Myers2004](#), Cap.3];
 - se presupunea că un program este scris doar pentru execuția de către calculator și nu este util și necesar să fie citit și înțeles de o persoană, e.g., programator, tester;
- metode de analiză statică bazate pe:
 - *factorul uman* (*engl. human-based testing*, HbT);
 - *instrumente specializate* (*engl. computer-based testing*, CbT).

Analiză statică. Clasificare

- metode de analiză statică bazate pe:
 - *factorul uman* (*engl. human-based testing*, HbT), i.e., **reviews**:
 - **formale**: inspectare Fagan, technical review, walkthroughs;
 - **informale**: buddy check, pairing, pair review, over-the-shoulder, e-mail pass-around;
 - *instrumente specialize* (*engl. computer-based testing*, CbT), i.e., **static analysis**:
 - **tool-assisted**: style checker, corecteness checker;
 - metode hibride: **pair-programming**.

METODE BAZATE PE FACTORUL UMAN

Definiție. Motivație. Obiective
Avantaje și dezavantaje. HbT vs CbT
Inspectare
Walkthroughs
Pair-Programming

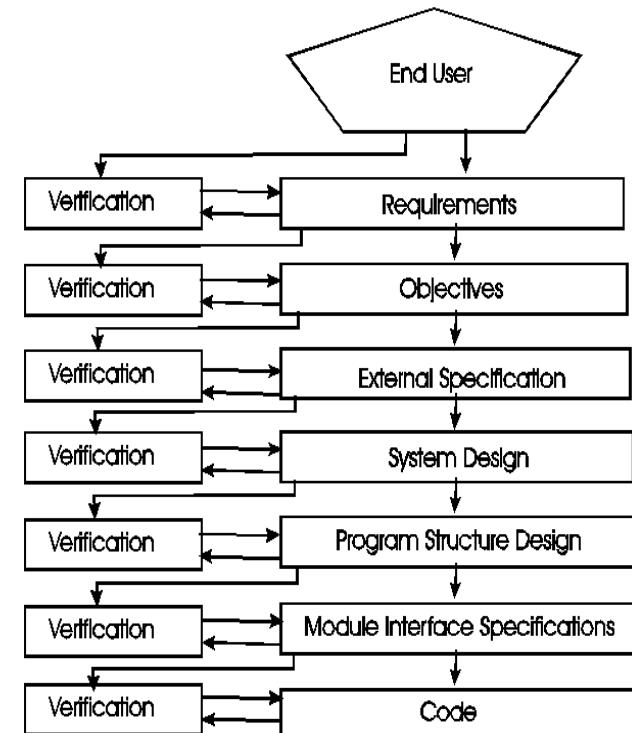
Metode bazate pe factorul uman. Definiție

- **metodă HbT** [[Young2008](#), [Frentiu2010](#)]
 - verificare efectuată de o persoană sau un grup persoane la sfârșitul unei etape a procesului de dezvoltare și înainte de a demara următoarea fază de dezvoltare;
- **exemplu:**
 - activitate: **inspectarea codului sursă**;
 - se efectuează *după* etapa de implementare și *înainte* de începerea testării.



Metode HbT. Obiective. Motivație

- **obiective**
 - *identificarea defectelor;*
- **motivație**
 - *utilizarea metodelor HbT contribuie la creșterea productivității și a gradului de încredere că rezultatul obținut îndeplinește cerințele:*
 - costul de corectare (eliminare) al defectelor crește odată cu parcurgerea etapelor de dezvoltare a softului;
 - modificarea comportamentului programatorilor la demararea analizei dinamice, i.e., la depanare se introduc mai multe bug-uri.



Metode HbT. Avantaje și dezavantaje

Avantaje

- sunt implicate în proces și **alte persoane** pe lângă autorul documentului verificat;
- permite **localizarea** defectelor;
- **identifică** între 30% și 70% din **bugurile de proiectare și implementare ale produselor soft.**

Dezavantaje

- nu sunt eficiente la identificarea **erorilor majore** de proiectare;
- nu pot evidenția situații excepționale care apar în utilizarea propriu-zisă a softului.

INSPECTARE FAGAN

Definiție. Caracteristici

Echipa de inspectare. Atribuțiile membrilor

Activități de inspectare

Checklists. Definiție. Motivație. Tipuri de checklists

Avantaje

Inspectare Fagan. Definiție. Caracteristici

- 1976 – Fagan [[Fagan1976](#)] introduce la IBM procesul de inspectare;
- **Inspectare**
 - proces structurat prin care se încearcă **identificarea defectelor** din **documentele elaborate** pe parcursul etapelor de dezvoltare a softului, pe baza unor **criterii prestabilite**;
- **Caracteristici**
 - **echipa de inspectare (4 membri):**
 - moderator, autor, secretar, prezentator;
 - **activități de inspectare (6 etape):**
 - planificarea, prezentarea, pregătirea, ședința de analiză, corectarea, reinspectarea;
 - **tipuri de erori căutate:** *checklists*, adaptate tipului de document inspectat;
 - **timp de desfășurare:** 90-120 minute.

Echipa de inspectare. Atribuțiile membrilor

- **moderator**
 - distribuie materialele și planifică sesiunile de inspectare; conduce sesiunea de inspectare;
 - urmărește modul în care sunt corectate erorile;
- **autorul documentului inspectat** (analist, proiectant, programator, tester);
 - răspunde la întrebările adresate de membrii echipei, clarifică nelămuririle semnalate de către aceștia;
 - participă la discuțiile purtate în timpul ședinței de analiză; remediază defecțiunile constatate;
- **secretar**
 - redactează concluziile ședinței de analiză;
 - înregistrează defectele semnalate și problemele discutate într-un document (raport de inspectare);
- **prezentator (reader)**
 - citește în cadrul ședinței de analiză părți ale documentului inspectat;
- **inspectori** – cu excepția autorului, toți ceilalți sunt considerați inspectori;
 - analizează documentul primit cu scopul de a identifica cât mai multe defecte (bug-uri).

Activități de inspectare (1)

1. **planificarea** (*engl. planning*)
 - moderatorul alege membrii echipei de inspectare;
 - distribuie materialele tuturor membrilor echipei și atribuie sarcini de inspectare;
 - verifică dacă documentul care trebuie inspectat este complet și acceptabil pentru a fi inspectat;
2. **prezentarea** (*engl. overview*) – nu este obligatorie
 - se prezintă detaliile materialului inspectat tuturor membrilor echipei de inspectare;
 - moderatorul poate decide dacă este necesară etapa de prezentare sau se trece direct la pregătirea individuală;
3. **pregătirea individuală** (*engl. preparation*)
 - citirea atentă și înțelegerea documentului primit pentru inspectare;
 - inspectorii rețin toate observațiile critice și formulează întrebări referitoare la aspectele care nu sunt clare.

Activități de inspectare (2)

4. **ședința de inspectare** (*engl. inspection meeting*)
 - se discută observațiile critice ale fiecărui inspector;
 - secretarul notează observațiile considerate prin consens ca fiind defecte și ulterior redactează concluziile inspectării;
 - concluziile inspectării sunt predate autorului documentului inspectat pentru a corecta greșelile;
5. **corectarea** (*engl. rework*)
 - autorul efectuează modificările necesare și corectează erorile;
6. **reinspectarea** (*engl. follow-up*)
 - se verifică dacă modificările efectuate au eliminat erorile;
 - se poate reduce la o întâlnire între autor și moderator.

Checklists. Definiție. Motivație. Tipuri de checklists

- **checklist** = listă cu defecte frecvent întâlnite într-un anumit tip de document;
- **motivație**
 - obiectivul inspectării: identificarea defectelor;
 - în raport cu documentul analizat, se urmărește identificarea unor bug-uri specifice;
- **tipuri de checklists** pentru inspectarea:
 - documentației de specificare;
 - documentației de analiză;
 - documentației de codificare;
 - documentației de testare.
- **Fiecare checklist conține aspecte particulare documentelor inspectate și sunt rezultatul experienței acumulate în identificarea greșelilor întâlnite frecvent în desfășurarea unor etape de dezvoltare software.**

Tipuri de checklists (1)

- checklist utilizat la inspectarea **documentației de specificare**:
 1. specificația respectă cerințele beneficiarului?
 2. există ambiguități în specificare?
 3. datele de intrare și de ieșire, cât și condițiile de intrare și de ieșire asociate sunt specificate corect?
 4. există cerințe care nu sunt specificate în document?
 5. există cerințe de precizie a datelor? sunt clar exprimate?
 6. există cerințe de performanță?
- checklist utilizat la inspectarea **documentației de analiză**:
 1. se respectă specificația?
 2. toate funcționalitățile din specificare au fost descrise?

Tipuri de checklists (2)

- checklist utilizat la inspectarea **documentației de codificare**:
 1. codul sursă respectă cerințele de proiectare, specificațiile și cerințele utilizatorului?
 2. sunt apelate toate metodele?
 3. sunt inițializate toate variabilele?
 4. aspecte analizate în mod special: cicluri infinite, accesarea unui index non-valid, alocarea și accesarea memoriei.
- checklist utilizat la inspectarea **documentației de testare**:
 1. toate cazurile de testare au fost documentate complet?
 2. cazurile de testare sunt relevante?
 3. datele de testare satisfac criteriul de acoperire ales?
 4. la testarea de integrare este clară ordinea de integrare?

Inspectare Fagan. Avantaje

- **avantaje**
 - permite descoperirea defectelor devreme;
 - reducere costul și timpul de dezvoltare;
 - metodă de grup – membrii echipei conlucrează;
 - modalitate de învățare la nivelului echipei;
 - **stabilește sursa defecțiunii, nu oferă doar indicii** referitoare la existența lor, e.g., testarea;
 - elimină stresul depanării într-un timp foarte scurt.
- **Inspectare vs. Testare** [[Collard2003](#)]
 - identificarea, localizarea și eliminarea defectului;
 - abordare aplicată în două etape (individual și apoi în grup);
 - checklists se focalizează pe anumite părți ale documentului care sunt **predispuse la introducerea de defecte** pe parcursul dezvoltării softului.

WALKTHROUGHS

Definiție. Caracteristici

Walkthroughs vs Inspectare

Walkthroughs. Definiție. Caracteristici

- **walkthroughs** [[Yourdon1979](#)]
 - procesul prin care se încearcă identificarea defectelor din documentele elaborate pe parcursul etapelor de dezvoltare a softului sub îndrumarea autorului documentului;
- **caracteristici** [[Yourdon1979](#) , [Collard2003](#)]
 - **echipa de realizare (3-5 membri) :**
 - secretar, inspector și moderator (autorul documentului inspectat, i.e., analist, proiectant, programator, tester);
 - **activitățile de walkthrough (4 etape):**
 - planning, meeting, rework, follow-up;
 - aplică tehnici de identificare a erorilor diferite de inspectarea Fagan, i.e., **nu se folosesc checklists**;
 - **temp de realizare:** 90-120 minute.

Walkthroughs vs Inspectare Fagan

Walkthroughs

- activitate mai puțin riguroasă;
- echipa este formată din 3-5 membri;
- se desfășoară în 4 etape;
- nu are pretenția identificării tuturor defectelor;
- autorul conduce echipa de walkthrough;
- se folosesc scenarii prestabilite.

Inspectare Fagan

- activitate riguroasă;
- echipa este formată din 4 membri;
- se desfășoară în 6 etape;
- identifică defectele des întâlnite;
- moderatorul conduce echipa de inspectare;
- folosește checklists pentru identificarea defectelor.

TECHNICAL REVIEW

Definiție. Caracteristici

Technical review vs Inspectare

Technical Review. Definiție. Caracteristici

- **technical review** [[TechReview2019](#)]
 - tip de review formal realizat de o echipă formată din **personal calificat tehnic** care examinează conformitatea unui document (work product) cu scopul pentru care este utilizat și identifică diferențele față de specificații și standarde;
- **caracteristici**
 - **echipa de realizare (3-5 membri)** :
 - secretar, inspectori, moderator (conduce echipa) și autorul documentului inspectat (toți fiind persoane calificate în același domeniu și fiind considerate *peer reviewers* față de autor);
 - autorul nu este și secretar;
 - **activitățile de technical review (3-4 etape)**:
 - planning, preparation (obligatoriu), meeting (optional), rework;
 - **timp de realizare**: 60-90 minute.

Technical Review vs Inspectare Fagan

Technical Review

- activitate mai puțin riguroasă;
- echipa este formată din 3-5 membri;
- se desfășoară în 4 etape;
- **obiective:** identificarea unui consens, identificarea posibilelor defecte, identificarea de idei noi și motivarea autorului să îmbunătățească documentele elaborate folosind implementări alternative;
- utilizarea checklists este optională.

Inspectare Fagan

- activitate riguroasă;
- echipa este formată din 4 membri;
- se desfășoară în 6 etape;
- **obiective:** evaluarea calității, identifică defectele des întâlnite;
- folosește checklists pentru identificarea defectelor.

INFORMAL REVIEW

Definiție. Caracteristici
Informal Review vs Inspectare

Informal Review. Definiție. Caracteristici

- informal review [[TechReview2019](#)]
 - se realizează fără o procedură formală sau documentată;
- exemple: **buddy check, pairing, pair review, over-the-shoulder, e-mail pass-around.**
- caracteristici
 - realizare în pereche (2 persoane) sau echipe (>2 persoane):
 - autorul, inspectori (cel puțin unul, toți sunt *peer reviewers* față de autor);
 - activitățile asociate unui review informal (1-2 etape):
 - meeting (optională), rework;
 - timp de realizare: 15-60 minute.

Informal Review vs Inspectare Fagan

Informal Review

- activitate se scurtă durată, puțin riguroasă;
- perechi (autor, inspector) sau echipe de membri;
- se desfășoară în 1-2 etape;
- **obiective:** identificarea posibilelor defecte, identificarea de **idei noi**, rezolvarea unor probleme minore;
- inspectorul este un coleg;
- utilizarea checklists este opțională, rezultatele se pot documenta;
- gradul de utilitate depinde de inspector;
- utilizată frecvent metodologiile Agile.

Inspectare Fagan

- activitate riguroasă;
- echipa este formată din 4 membri;
- se desfășoară în 6 etape;
- **obiective:** evaluarea calității, identifică defectele des întâlnite;
- moderatorul conduce echipa de inspectare;
- folosește checklists pentru identificarea defectelor.

PAIR-PROGRAMMING

Definiție. Caracteristici

Pair-Programming. Definiție. Caracteristici

- **pair-programming**
 - metodă de elaborare a programelor, în care două persoane lucrează împreună;
- **caracteristici**
 - combină activitățile: inspectarea codului și implementarea (codificarea);
 - programatorii alternează rolurile;
 - **activități de inspectare:**
 - nu sunt determinate de checklists;
 - **se bazează pe împărtășirea acelorași principii de programare și a unui stil de programare asemănător;**
 - **temp de desfășurare:** durata unei zile normale de muncă, fără exces de ore suplimentare sau presiunea unui program de lucru strict;
 - nu există mediatori, iar responsabilitatea pentru atmosfera de lucru deschisă și non-agresivă depinde de programatori.

PENTRU EXAMEN...

Pentru examen...

- **concepțe, caracteristici, asemănări și diferențe:**

- verificare, validare; verificare vs. validare;
- eroare, defect/bug, defecțiune; eroare vs. defect/bug vs. defecțiune;
- stakeholders, calitate, QA, QC;
- analiza statică vs analiza dinamică;
- HbT, motivație;
- inspectare Fagan, walkthroughs, technical review, informal review:
 - descriere, rolurile membrilor echipei, activitățile asociate și descrierea lor, avantaje;
- pair-programming:
 - caracteristici, avantaje.

Cursul următor...

- **Testare**
 - modele folosite în testare;
 - planuri de testare;
 - cazuri de testare;
 - raportarea testării;
- **Technici de testare Black-box**
 - împărțirea în clase de echivalentă;
 - analiza valorilor limită;
- **Testing Management Tool – TestLink**
 - prezentare tool.

Referințe bibliografice

- [Crosby1980] Philip B. Crosby, *Quality Is Free*, Signet Shakespeare, 1980.
- [Juran1998] A. Blanton Godfrey, Joseph Juran, *JURANS QUALITY HANDBOOK*, McGraw-Hill, 1998.
- [Weinberg1992] Gerald Weinberg, *Quality Software Management , Vol. 1: Systems Thinking*, Dorset House Publishing, 1992.
- [Pressman2000] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, Inc., 2000.
- [Pal2013] Kaushik Pal, *Software Testing: Verification and Validation*, <http://mrbool.com/software-testing-verification-and-validation/296091>
- [Fagan1976] M. E. Fagan, *Design and code inspections to reduce errors in program development*, IBM Systems Journal, pages 182–211, 1976.
- [Collard2003] J. F. Collard, I. Burnstein. *Practical Software Testing*. Springer-Verlag New York, Inc., 2003.
- [Yourdon1979] E. Yourdon, *Structured Walkthroughs*, Prentice-Hall, Englewood Cliffs, NJ, 1979.
- [Myers2004] Glenford J. Myers, *The Art of Software Testing*, John Wiley & Sons, Inc., 2004
- [Young2008] M. Pezzand, M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley and Sons, 2008.
- [Frentiu2010] M. Frentiu, *Verificarea si validarea sistemelor soft*, Presa Universitara Clujeana, 2010.
- [TechReview2019] Cania Consulting, Informal and formal testing reviews, <https://cania-consulting.com/2019/10/12/test-manager-guide-to-reviews/>

CURS 02A. TESTARE

Verificarea și validarea sistemelor soft
[07 Martie 2023]

Lector dr. Camelia Chisăliță-Crețu
Universitatea Babeș-Bolyai

Conținut

- Evaluarea calității unui produs soft
 - Activități asociate calității
 - Controlul calității. Activități asociate
 - Metode de verificare și validare
- Testare
 - Program. Program testat
 - Definiții ale testării
 - Caz de testare. Definiții. Caracteristici
 - Tipuri de testare
 - Principii de testare. Axiome de testare
- Procesul de testare
 - Întrebări fundamentale
 - Activități ale procesului de testare
- Bibliografie

EVALUAREA CALITĂȚII UNUI PRODUS SOFT

Activități asociate calității

Controlul calității. Activități asociate

Metode de verificare și validare

Activități asociate calității

- În procesul de dezvoltare, calitatea este abordată din perspectiva:
 - procesului ==> **asigurarea calității** (engl. quality assurance):
 - **Obiectiv:** asigură respectarea standardelor, planurilor și etapelor proceselor de dezvoltare necesare elaborării adecvate a produsului cerut;
 - **Întrebare:** Cum se asigură calitatea activităților desfășurate în procesul dezvoltare?
 - produsului ==> **controlul calității** (engl. quality control):
 - **Obiectiv:** identifică deficiențele în produsul obținut;
 - **Întrebare:** Cum se controlează calitatea rezultatelor obținute (e.g., work products) în urma activităților desfășurate?

Asigurarea calității

- **Prevenție** bug-uri
- Orientare pe **proces**
- Planificarea și monitorizarea activităților

Controlul calității

- **Detectie** bug-uri
- Orientare pe **produs**
- Căutare și eliminare bug-uri

Controlul calității. Activități asociate (1)

Analiză statică (static testing)

- examinarea unor documente (specificații, modele conceptuale, diagrame de clase, cod sursă, planuri de testare, documentații de utilizare);
- **exemple:** activități de inspectare a codului, analiza algoritmului, demonstrarea corectitudinii;
- se pot baza pe factorul uman (reviews) sau utilizarea tool-urilor (analiza statică).

Analiză dinamică (dynamic testing)

- examinarea comportamentului programului cu scopul de a evidenția defecțiuni posibile;
- **exemple:** *tipuri de testare* (de regresie, funcțională, non-funcțională), *niveluri de testare* (testare unitară, testare de integrare, testare de sistem, testare funcțională, testare de acceptare);
- se bazează întotdeauna pe execuția programului.

Controlul calității. Activități asociate (2)

Analiză statică (static testing)

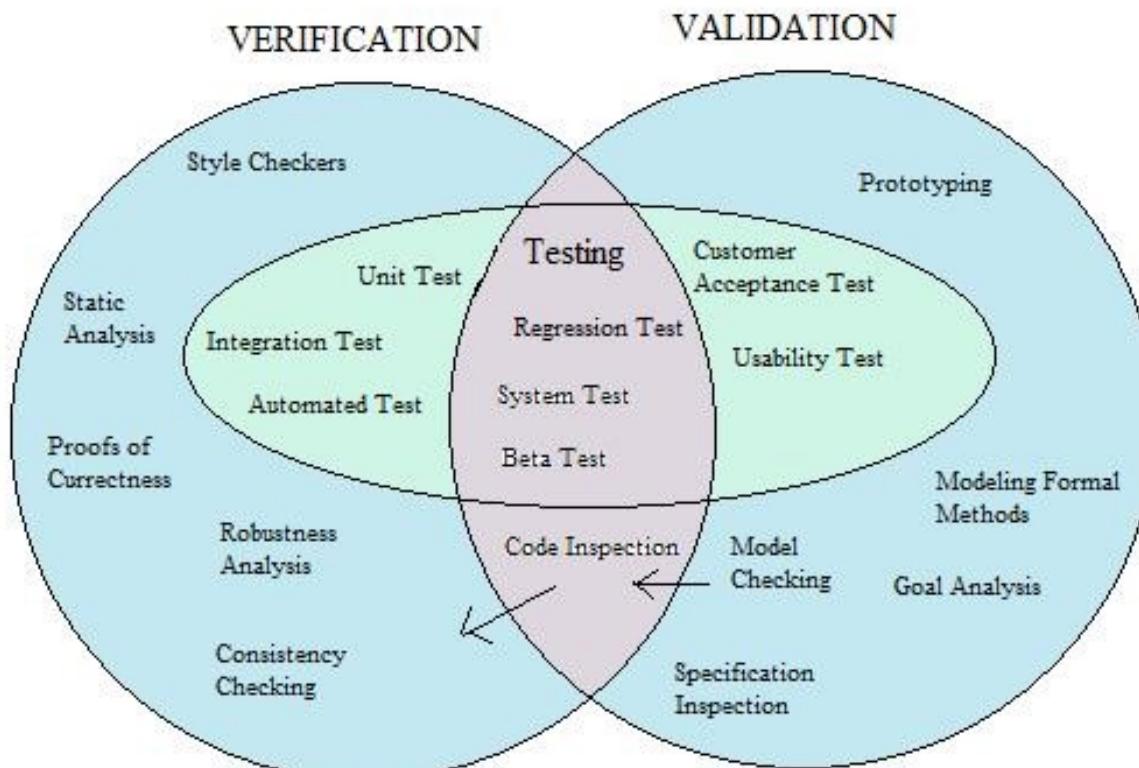
- permit identificarea mai multor erori (greșeli) care pot fi corectate simultan;
- **NU** presupune execuția propriu-zisă a programului dezvoltat;

Analiză dinamică (dynamic testing)

- sugerează doar un simptom, fiecare eroare identificată fiind eliminată individual;
- include activitatea de execuție propriu-zisă a programului (testare);
- poate să evidențieze o defecțiune doar în anumite situații.

- metode de analiză complementare;
- dezvoltatorii aplică metode hibride, care folosesc avantajele celor două abordări.

Metode de Verificare și Validare



sursa: [Pal2013]

TESTARE

Program. Program testat

Testare. Definiții

Caz de testare. Definiții. Caracteristici

Tipuri de testare

Principii de testare. Axiome de testare

Program. Definiție

- **program** (engl. **computer program**, **software application**, **software product**):
 - listă de instrucțiuni sau o mulțime de metode sau module care permit execuția de către un calculator;
- **un program** este
 - **o comunicare**
 - **între persoane și calculatoare**
 - care sunt **separate** în timp și spațiu
 - și conține **instrucțiuni** care sunt **executate** de către calculator. [BBST2010](#)

Program testat. Definiție

- **program testat** (*engl. software under test, SUT*):
 - \approx funcție matematică;
 - $P : D \rightarrow R$, unde
 - D – mulțimea datelor de intrare;
 - R – mulțimea datelor de ieșire așteptate.

Testare. Definiții. Caracteristici

1. *semnalează prezența defectelor unui program, fără a garanta absența acestora* [[Dijkstra1969](#)].
2. *procesul de execuție al unui program cu scopul de a identifica erori* [[Myers2004](#)].
3. *observarea comportării unui program în mai multe execuții* [[Frentiu2010](#)].
4. *investigație tehnică și empirică realizată cu scopul de a oferi beneficiarilor testării informații referitoare la programul testat* [[BBST2010](#)].

- Testarea este un proces
 - destructiv;
 - se poate finaliza cu succes (**passed**) sau eșec (**failed**).

Caz de testare. Definiție

- **caz de testare** (*engl. test case*) –
 - mulțime de **date de intrare, condiții de execuție și rezultate așteptate**, proiectate cu un anumit scop (e.g., cum ar fi parcurgerea unui drum particular în execuția programului sau pentru a verifica respectarea unei cerințe specifice) [[IEEE1990](#)];
 - **o interogare** adresată de tester programului testat [[BBST2010](#)];
 - este relevant obiectivul informațional, i.e., **informația pe care o descoperim prin testare**, e.g., testul este *passed* sau *failed*, timpul de execuție asociat testului este foarte mare.
 - **notătie:** (i, r) , $i \in D$, $r \in R$;
 - pentru intrarea i se așteaptă să se obțină rezultatul r .

Caz de testare. Caracteristici

- **caracteristici ale unui caz de testare care are efectul așteptat [BBST2011], i.e., caz de testare eficace:**

- probabilitate mare de a identifica bug-uri;
- nu este redundant;
- relevant în cadrul categoriei din care face parte;
- nu este prea simplu;
- nu este prea complex.

power	representative	performable	easy to evaluate	affordable
valid	non-redundant	maintainable	support troubleshooting	opportunity cost
value	motivating	information value	appropriately complex	
credible	reusable	coverage	accountable	

Tipuri de testare. Definiții

- **testare exhaustivă (testare completă, engl. exhaustive testing, complete testing):**
 - testare cu toate cazurile de testare posibile, folosind toate datele și scenariile de utilizare posibile;
 - dacă D este finit atunci P se poate executa pentru fiecare $i \in D$;
 - în majoritatea situațiilor D nu este finit, deci testarea exhaustivă nu este posibilă și nici eficace;
- **testare selectivă (engl. selective testing):**
 - testare cu o submulțime de cazuri de testare;
 - dacă D nu este finit, atunci se aleg o parte din elementele i , unde $i \in S$, $S \subset D$.
- **depanare (engl. debugging, bug fixing):**
 - proces de localizare și eliminare al unui bug care a fost evidențiat prin testarea programului;
 - se formulează ipoteze asupra comportamentului programului, se corectează defectele și apoi se reia procesul de testare.

Principii de testare

[[Myers2004](#)] [[Cap2. sectiunea Software Testing Principles](#)]

1. Definește rezultatele așteptate în urma testării.
2. Evită să testezi programelor proprii.
3. Analizează riguros rezultatele fiecărui test.
4. Scrie cazuri de testare atât pentru condiții de intrare valide cât și pentru cele non-valide.
5. Testează dacă programul **nu** face ceea ce se precizează în specificație, dar și dacă ceea ce face programul **nu** este descris în specificații.
6. Păstrează întotdeauna cazurile de testare.
7. Organizează și planifică procesul de testare, considerând că se vor identifica bug-uri.
8. Testarea este o activitate de stimulare a creativității.
 - *The goal of a software tester is to find bugs, find them as early as possible, and make sure they get fixed.*

Axiome ale testării [Patton2005] [Cap.3. sectiunea Testing Axioms]

1. Este imposibil ca un program să fie complet (exhaustiv) testat.
2. Testarea softului presupune asumarea unui risc.
3. Testarea nu poate demonstra că bug-urile nu există.
4. Numărul mare de bug-uri asociat unei funcționalități este un indicator al prezenței altor bug-uri – bug-urile pot fi grupate în anumite funcționalități, nu sunt izolate.
5. Paradoxul pesticidului (în testare): cu cât un program este testat mai mult folosind aceleași teste (tehnici de testare), imunitatea la testare crește (nu se descoperă bug-uri noi).
6. Nu orice bug identificat va fi eliminat.
7. Specificația produsului soft se schimbă în permanență.
8. Testerii nu sunt cei mai apreciați membri ai echipei de dezvoltare.

PROCESUL DE TESTARE

Întrebări fundamentale

Activități ale procesului de testare

Întrebări fundamentale (1)

- **De ce este necesar sa testăm un produs soft ? Care este scopul testării ?**
 - evaluarea unor caracteristici sau atribute care să reflecte calitatea produsului soft;
 - descoperirea unor **informații** referitoare la produsul soft [[BBST2010](#)];
 - **obiective ale testării** = aspectele de interes care vor fi investigate (evaluate) în procesul de testare;
- **Cum se organizează procesul de testare ?**
 - **contextul aplicației** = particularități de realizare a testării, e.g., componenta testată, beneficiarul testării (**stakeholder**), diverse **constrângeri**, etc;
 - **misiunea testării** = acțiunea desfășurată prin testare pentru a atinge **obiectivele testării**;
 - **strategie de testare** = cadru general prin care se determină care sunt cele mai potrivite teste care trebuie proiectate (i.e., ce **tehnici de testare** se aplică), astfel încât testarea să își atingă obiectivele informaționale, luând în considerare **contextul aplicației** în care se desfășoară testarea;
 - **tehnică de testare** = *metodă de proiectare, implementare și interpretare a rezultatelor unui test*;
 - **abordare a testării** = modalitate de aplicare a unei tehnici de testare, e.g., *black-box testing, white-box testing, grey-box testing, exploratory testing, scripted testing*.

Întrebări fundamentale (2)

- Cum determinăm momentul în care putem realiza testarea? *Care sunt condițiile care trebuie indeplinite pentru a demara procesul de testare?*
 - criterii de începere a testării (*engl. entry criteria*);
- Cum determinăm momentul în care testarea efectuată este suficientă? *Cât timp testăm, câte teste executăm?*
 - criterii de terminare a testării (*engl. exit criteria*).

Întrebări fundamentale. Exemplu

- **Exemplu. Criteriu de terminare a testării într-o strategie bazată pe risc:**

- Se analizează relația dintre numărul de teste executate, i.e., **amount of testing**, și numărul de bug-uri identificate, i.e., **quantity**;

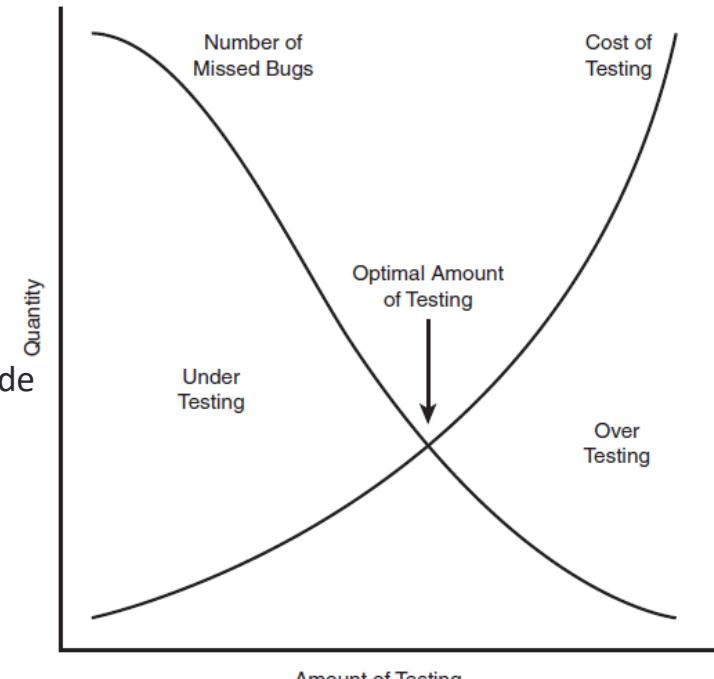
- **Over testing:**

- Dacă se testează tot/mult: costurile cresc, numărul de bug-uri scade
- raportul $\frac{\text{bugs found}}{\text{testing costs}}$ devine mic ==> **eficiența testării scade**;

- **Under testing:**

- Dacă se testează puțin sau se iau decizii nepotrivite legate de **CE** se va testa: costurile sunt mici, numărul de bug-uri rămâne ridicat
- raportul $\frac{\text{bugs found}}{\text{testing costs}}$ rămâne mare ==> **calitate redusă**;

- **Fiecare proiect soft are un cost de testare optim.**



Activități ale procesului de testare

1. planificare (engl. test planning):

- stabilirea obiectivelor (*de ce testăm*);
- stabilirea terminării testării (*cât testăm*);
- identificarea unității de program care trebuie testată (*ce testăm*);
- elaborarea strategiei de testare (*cum testăm, ce tehnici aplicăm, ce abordare folosim*);

2. proiectare (engl. test design):

- stabilirea datelor de intrare;
- stabilirea rezultatului așteptat;
- configurarea mediului de execuție pentru program;

3. testare (engl. test execution):

- execuția programului, i.e, rularea testelor;

4. analiza (engl. test result analysis):

- analiza rezultatului testului (*evaluarea rezultatului*);
- raportarea bug-urilor;

5. monitorizare (engl. test control and monitoring):

- supravegherea procesului de testare;
- evaluarea și îmbunătățirea procesului de testare.

• QA vs QC în procesul de testare:

- activități QC: 2 .. 4;
- activități QA: 1, 5;

Referințe bibliografice

- [Pal2013] Kaushik Pal, *Software Testing: Verification and Validation*, <http://mrbool.com/software-testing-verification-and-validation/29609>
- [Dijkstra1969] E.W. Dijkstra, *Software engineering techniques*, Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 October 1969.
- [Myers2004] Glenford J. Myers, *The Art of Software Testing*, John Wiley & Sons, Inc., 2004
- [Frentiu2010] M. Frentiu, *Verificarea si validarea sistemelor soft*, Presa Universitara Clujeana, 2010.
- [BBST2010] Black-Box Software Testing (BBST), Foundations,
<http://www.testingeducation.org/BBST/foundations/BBSTFoundationsNov2010.pdf>.
- [IEEE990] IEEE, IEEE STD 610, In IEEE Standard Glossary of Software Engineering Terminology, 1990.
- [Patton2005] R. Patton, *Software Testing*, Sams Publishing, 2005.
- [ISTQBCertification2020] ISTQB Exam Certification, <http://tryqa.com/what-is-a-defect-life-cycle/>.
- [BBST2011] BBST – Test Design, Cem Kaner,
<http://www.testingeducation.org/BBST/testdesign/BBSTTestDesign2011pfinal.pdf>

CURS 02B.

TESTARE BLACK-BOX

Verificarea și validarea sistemelor soft
[07 Martie 2023]

Lector dr. Camelia Chisăliță-Crețu
Universitatea Babeș-Bolyai

Conținut

- Abordări ale testării
- Testare Black-Box
 - Definiție. Caracteristici.
 - Clasificare. Tehnici de testare black-box
 - Partiționarea în clase de echivalentă. Exemple
 - Analiza valorilor limită. Exemple
 - Partiționarea în clase de echivalentă vs Analiza valorilor limită
 - Avantaje și dezavantaje
- Pentru examen...
- Bibliografie

ABORDĂRI ALE TESTĂRII

Abordări ale testării. Clasificare

Tehnici de testare asociate

Abordări ale testării. Clasificare

- abordare a testării
 - modalitate de realizare a testării în care se aplică una sau mai multe tehnici de testare în cadrul unei strategii de testare stabilită anterior;
- clasificare
 - testare Black-box (**criteriul cutiei negre**, *engl. Black-box testing*);
 - testare White-box (**criteriul cutiei transparente**, *engl. White-box testing*);
 - testare Grey-box (**criteriul cutiei gri**, *engl. Grey-box testing*);
 - testare exploratorie (*engl. Exploratory testing*);
 - testare bazată pe scripturi (*engl. Scripted testing*);

Abordări ale testării. Tehnici de testare asociate

- Testare Black-Box – testare funcțională:
 - Partiționarea în clase de echivalență;
 - Analiza valorilor limită;
 - Tabele de decizie, Cazuri de utilizare, Scenarii de utilizare, etc.;
- Testare White-box – testare structurală:
 - Acoperirea fluxului de control (e.g., instrucțiuni, ramificații, decizii, condiții, bucle, drumuri);
 - Acoperirea fluxului de date;
- Testare Grey-box – testare mixtă:
 - folosirea simultană a avantajelor abordărilor black-box și white-box pentru proiectarea cazurilor de testare.

TESTARE BLACK-BOX

Definiție. Caracteristici. Tehnici de testare black-box

Partiționarea în clase de echivalență. Exemple

Analiza valorilor limită. Exemple

Partiționarea în clase de echivalență vs Analiza valorilor limită

Avantaje și dezavantaje

Definiție. Caracteristici

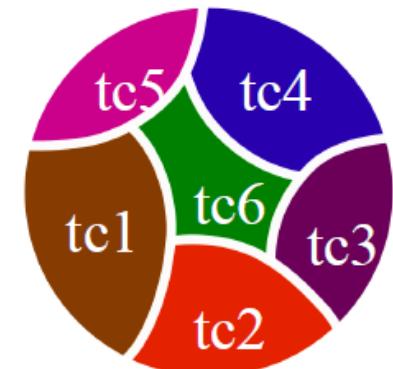
- **testare black-box (engl. black-box testing, data driven testing, input/output driven testing):**
 - testare funcțională;
 - datele de intrare se aleg pe baza **specificației problemei**, programul fiind văzut ca o cutie neagră;
 - nu se utilizează informații referitoare la structura internă a programului, i.e., codul sursă;
 - permite identificarea situațiilor în care programul nu funcționează conform specificațiilor.

Tehnici de testare black-box

- tehnici de proiectare a cazurilor de testare bazate pe criteriul black-box:
 1. **Partiționarea în clase de echivalență;**
 2. **Analiza valorilor limită;**
 3. Testarea domeniului de valori;
 4. Tabele de decizie;
 5. Cazuri de utilizare;
 6. Scenarii de utilizare;
 7. *alte tehnici.*

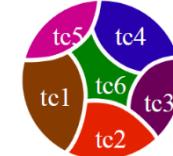
Partiționarea în clase de echivalență. Motivație

- În general, **testarea exhaustivă** nu este posibil de realizat, e.g.:
 - există un set consistent de date de intrare sau domeniul de valori testat este infinit;
 - există restricții, e.g., timp, buget, resursa umană.
- partiționarea în clase de echivalență (*engl. Equivalence Class Partitioning, ECP*) este **eficientă** pentru reducerea numărului de cazuri de testare care trebuie proiectate;
- **Etape:**
 - *identificarea claselor de echivalență disjuncte:*
 - se evită redundanța cazurilor de testare;
 - *proiectarea cazurilor de testare:*
 - se alege un singur element din fiecare clasă de echivalență;



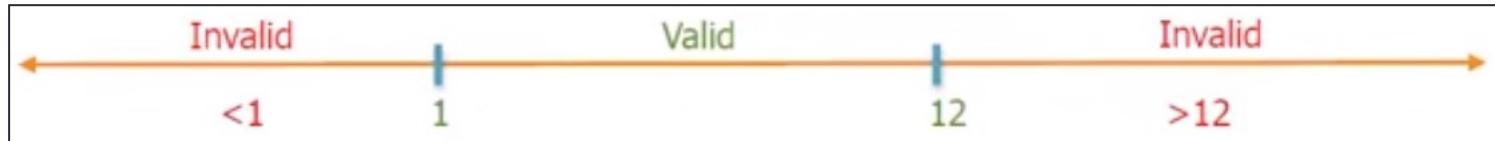
Partiționarea în clase de echivalență. Definiție

- **clasă de echivalență (engl. equivalence class, EC):**
 - mulțimea datelor de intrare/ieșire pentru care programul are comportament similar [[Myers2004](#)];
- procesul de **partiționare în clase de echivalență (engl. equivalence class partitioning, ECP)**:
 - împărțirea (divizarea) domeniului datelor de intrare/ieșire în EC, astfel încât, dacă programul va rula corect pentru o valoare dintr-o EC, atunci va rula corect pentru orice valoare din acea EC.



ECP. Exemplu 1. Identificarea ECs

- Se consideră un formular de înscriere la un concurs. Pentru data nașterii se introduce ziua, luna și anul.
- Identificați clasele de echivalență corespunzătoare câmpului lună calendaristică (pentru data nașterii). Domeniul de valori valide este [1, 12].



Abordare primară:

un număr ≥ 1 și ≤ 12 ;

1 EC validă:

$$\text{EC}_1: D_1 = [1, 12];$$

2 EC non-valide:

$$\text{EC}_2: D_2 = \{\text{luna} < 1\} = (-\infty, 1);$$

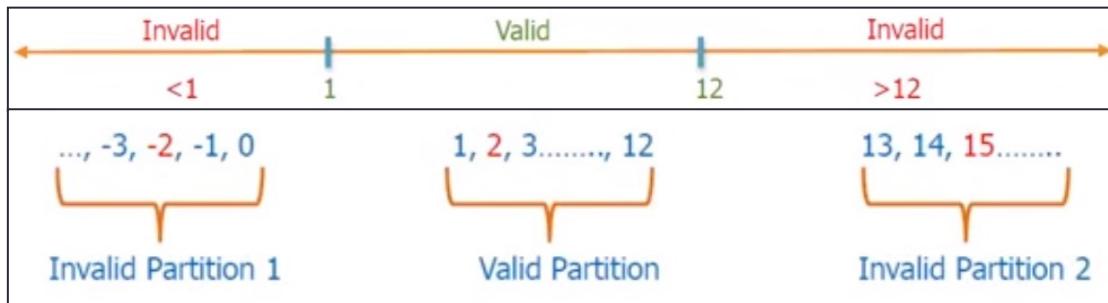
$$\text{EC}_3: D_3 = \{\text{luna} > 12\} = (12, +\infty);$$

$$\text{EC}_4: D_4 = \text{simboluri/litere din alfabet.}$$

Abordare secundară:

- Numărul de ordine al lunii în cadrul unui an: prima, a doua, a treia, etc.
- Numărul de cifre: 0 cifre, 1-2 cifre (1 .. 12), 3 cifre (non-valid);
- Numărul de spații înainte de/după cifră/e: 0 (cazul general), > 0 (caz excepțional);
- Numărul de spații între cifre: 0 (caz general), > 0 (unele programe (OOWriter) ignoră caracterele "non-valide" din interiorul numărului dat ca string);
- Codurile ASCII: cifre (48-57), non-cifre (58 to 127), etc.

ECP. Exemplu 1. Proiectarea cazurilor de testare



- ECs identificate:
 - 1 EC validă, $EC_1: D_1 = [1, 12]$;
 - 3 EC non-valide, $EC_2: D_2 = \{luna | luna < 1\} = (-\infty, 1)$, $EC_3: D_3 = \{luna | luna > 12\} = (12, +\infty)$, $EC_4: D_4 = \text{simboluri alfanumerice}$;
- Cazuri de testare proiectate:
 - 1 EC validă ==> 1 caz de testare valid, e.g., $TC_{01}: luna = 2$;
 - 3 EC non-valide ==> 3 cazuri de testare non-valide, e.g., $TC_{02}: luna = -2$, $TC_{03}: luna = 15$, $TC_{04}: luna = "%L10"$;
- Din fiecare EC de intrare identificată se alege o singură valoare. ECP consideră că fiecare EC tratează în manieră similară toate valorile din acea EC.

ECP. Exemplu 2. Identificarea ECs

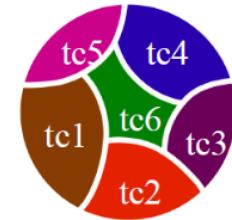
- Pentru constituirea unui depozit bancar se consideră următoarea ofertă de dobânzi:
 - 0,50% dacă valoarea depozitului este până la 1000,00 Euro;
 - 1,00% dacă valoarea depozitului este până la 2000,00 Euro, dar mai mult de 1000,00 Euro;
 - 1,50% dacă valoarea depozitului este peste 2000,00 Euro;
- **Care sunt clasele de echivalență valide și non-valide pentru valoarea depozitului constituit?**
 - Clase de echivalență valide:
 - **EC₁**: 0,00 Euro – 1000,00 Euro;
 - **EC₂**: 1000,01 Euro – 2000,00 Euro;
 - **EC₃**: >= 2000,01 Euro.
 - Clase de echivalență non-valide:
 - **EC₄**: < 0,00 Euro;
 - **EC₅**: > valoarea maximă admisă pentru un depozit.
 - **EC₆**: caractere din alfabet.

ECP. Exemplu 2. Proiectarea cazurilor de testare

- ECs identificate:
 - **3 ECs valide:**
 - EC_1 : 0,00 Euro – 1000,00 Euro;
 - EC_2 : 1000,01 Euro – 2000,00 Euro;
 - EC_3 : \geq 2000,01 Euro.
 - **3 ECs non-valide:**
 - EC_4 : < 0,00 Euro;
 - EC_5 : > valoarea maximă admisă pentru un depozit;
 - EC_6 : caractere din alfabet.
-
- **Cazuri de testare proiectate:**
 - **3 ECs valide ==> 3 cazuri de testare valide, e.g.:**
 - TC_{01} : amount = 678,99;
 - TC_{02} : amount = 1742,81;
 - TC_{03} : amount = 5213,00;
 - **3 ECs non-valide ==> 3 cazuri de testare non-valide, i.e., câte un TC care corespunde fiecărei EC non-valide identificate, e.g.:**
 - TC_{04} : amount = -0,79;
 - TC_{05} : amount = 9876543210,123;
 - TC_{06} : amount = #12a.

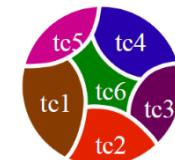
ECP. Algoritm

- Algoritm de aplicare a ECP (*identificarea ECs și proiectarea TCs*):
 1. se identifică clasele de echivalență pe baza condițiilor de intrare/ieșire;
 2. se clasifică clasele de echivalență în:
 - **valide** – formate din datele de intrare/ieșire valide pentru program;
 - **non-valide** – formate din datele de intrare/ieșire eronate, corespunzătoare tuturor celorlalte stări ale condiției de intrare/ieșire.
 3. se asociază un identificator unic fiecărei clase de echivalență (e.g., EC_1 , EC_2 , etc.);
 4. cât timp (*nu au fost descrise cazuri de testare pentru toate clasele de echivalență valide/non-valide*):
 - scrie (*un nou caz de testare care corespunde la cât mai multe clase de echivalență valide încă neacoperite*);
 - scrie (*un nou caz de testare care corespunde doar uneia dintre clasele de echivalență de non-valide încă neacoperite*).



ECP. Proiectarea cazurilor de testare. Reguli (1)

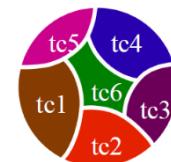
1. dacă o condiție de intrare precizează apartenența la un interval de valori $[a,b]$:
 - ==> 1 EC validă, 2 EC non-valide;
 - E.g.: luna, o valoare intervalul $[1, 12]$;
2. dacă o condiție de intrare precizează o mulțime finită de valori de intrare:
 - ==> 1 EC validă pentru fiecare valoare, 1 EC non-validă;
 - E.g.: tip curs \in CourseType = {optional, obligatoriu, facultativ};
 - 1 EC validă pentru fiecare element din CourseType:
 - **EC₁**: {optional},
 - **EC₂**: {obligatoriu},
 - **EC₃**: {facultativ} ==> 3 ECs valide;
 - 1 EC non-validă:
 - **EC₄**: $M = \{e \mid e \notin \text{CourseType}\}$;



ECP. Proiectarea cazurilor de testare. Reguli (2)

3. dacă o condiție de intrare precizează numărul de valori:

- ==> 1 EC validă, 2 EC non-valide;
 - E.g.: “de la 1 până la 5 studenți”;
 - 1 EC validă:
 - **EC₁**: D=[1,5];
 - 2 EC non-valide:
 - **EC₂**: nici un student;
 - **EC₃**: mai mult de 5 studenți;

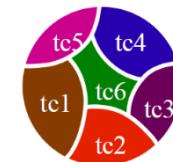


ECP. Proiectarea cazurilor de testare. Reguli (3)

4. dacă o condiție de intrare precizează o situație de tipul “must be”:

- ==> 1 EC validă, 1 EC non-validă.
 - E.g.,: “primul caracter din parolă trebuie să fie un simbol numeric”;
 - 1 EC validă:
 - **EC₁**: primul caracter este un simbol numeric;
 - 1 EC non-validă:
 - **EC₂**: primul caracter nu este un simbol numeric.

Dacă există argumente că programul nu tratează similar toate elementele dintr-o EC, atunci ECs se împart în ECs mai mici.

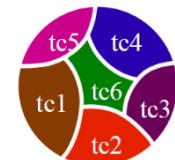


ECP. Acoperirea testării ECs

- **calculul acoperirii (engl. coverage) testării ECs pentru tehnica de testare ECP:**

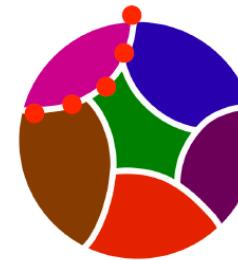
$$\text{Acoperirea ECs} = \frac{\text{numărul de ECs testate}}{\text{numărul de ECs identificate}} \times 100$$

- E.g.:
 - pe baza specificațiilor au fost identificate 18 ECs (pentru datele de intrare și ieșire);
 - pentru 15 ECs s-au proiectat, implementat și executat teste;
 - **Acoperirea ECs= $(15/18)*100 = 83,33\%$.**
- **Acoperirea ECs poate fi folosită ca și criteriu de terminare a testării, i.e., exit criteria.**



Este ECP eficientă la limita dintre ECs ?

- ECP presupune că programul are un comportament similar pentru toate valorile dintr-o EC;
- ECP nu garantează că programul este testat și la limitele ECs identificate;



- există greșeli de programare tipice care apar la limita ECs identificate;
 - e.g., pentru $x \geq 3$

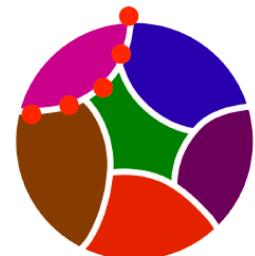
```
if (x>3) y++; //bug
```

```
if (x>=3) y++;
```
 - [ECP]: pentru $EC_1: [3, MaxInt]$ se alege $TC_{01}: x=4$, dar TC_{01} nu surprinde bug-ul de implementare.

Analiza valorilor limită. Motivație

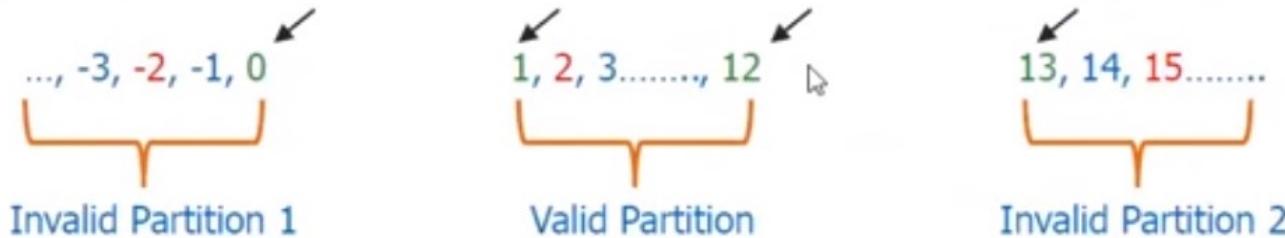
- analiza valorilor limită investighează posibilele bug-uri existente la limita dintre ECs identificate;
 - E.g.: pentru $x \geq 3$

```
if (x>3) y++; //bug
if (x>=3) y++;
```
 - [ECP]: pentru EC₁: [3, MaxInt] se alege TC₀₁: x=4, dar TC₀₁ nu surprinde bug-ul de implementare;
 - [BVA]: pentru EC₁: [3, MaxInt] se alege TC₀₂: x=3;
- **Etape:**
 - *identificarea condițiilor asociate valorilor limită*:
 - *proiectarea cazurilor de testare*:
 - se aleg date test pentru fiecare condiție limită identificată;

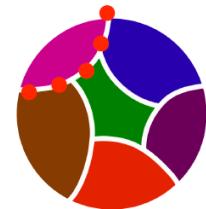


Analiza valorilor limită. Definiție

- analiza valorilor limită (*engl. boundary value analysis, BVA*) [[Myers2004](#)]:
 - testarea realizată prin alegerea datelor de test pe baza limitelor ECs de intrare/ieșire;



- valoare limită (*engl. boundary value, BV*):
 - valoare a domeniului pentru care comportamentul programului se modifică.



BVA. Exemplu 1. Condiții BVA

- Limitele unei EC valide indică situațiile în care comportamentul programului se schimbă!



- ECs identificate:
 - 1 EC validă: $EC_1: D_1 = [1, 12]$;
 - 3 EC non-valide: $D_2 = \{\text{luna} \mid \text{luna} < 1\} = (-\infty, 1)$, $D_3 = \{\text{luna} \mid \text{luna} > 12\} = (12, +\infty)$, $D_4 = \text{simboluri alfanumerice}$;
- Condiții BVA, construite pentru limitele ECs valide:

<ul style="list-style-type: none">• Limita inferioară a EC_1:<ul style="list-style-type: none">• 1. luna = 0; (non-validă)• 2. luna = 1;• 3. luna = 2;	<ul style="list-style-type: none">• Limita superioară a EC_1:<ul style="list-style-type: none">• 4. luna = 11;• 5. luna = 12;• 6. luna = 13; (non-validă)
---	--

BVA. Exemplu 1. Proiectarea cazurilor de testare

- ECs valide identificate:
 - 1 EC validă:
 - $EC_1: D_1 = [1, 12]$;
- Cazuri de testare proiectate pe baza condițiilor BVA identificate:
 - Limita inferioară a EC_1 :
 - 1. luna = 0 $\implies TC_{01}$: luna = 0; (non-valid)
 - 2. luna = 1 $\implies TC_{02}$: luna = 1; (valid)
 - 3. luna = 2 $\implies TC_{03}$: luna = 2; (valid)
 - Limita superioară a EC_1 :
 - 4. luna = 11 $\implies TC_{04}$: luna = 11; (valid)
 - 5. luna = 12 $\implies TC_{05}$: luna = 12; (valid)
 - 6. luna = 13 $\implies TC_{06}$: luna = 13; (non-valid)

BVA. Exemplu 2. Condiții BVA

- ECs valide identificate:
 - **EC₁**: 0,00 Euro – 1000,00 Euro;
 - **EC₂**: 1000,01 Euro – 2000,00 Euro;
 - **EC₃**: >= 2000,01 Euro.
- Condiții BVA identificate:
 - Limita inferioară a EC₁:
 - 1. amount = -0,01; (non-validă)
 - 2. amount = 0,00;
 - 3. amount = 0,01;
 - Limita superioară a EC₁:
 - 4. amount = 999,99;
 - 5. amount = 1000,00;
 - 6. amount = 1000,01; (non-validă)
 - Limita inferioară a EC₂:
 - 1. amount = 1000,00; (non-validă)
 - 2. amount = 1000,01;
 - 3. amount = 1000,02;
 - Limita superioară a EC₂:
 - 4. amount = 1999,99;
 - 5. amount = 2000,00;
 - 6. amount = 2000,01; (non-validă)
 - Limita inferioară a EC₃:
 - 1. amount = 2000,00; (non-validă)
 - 2. amount = 2000,01;
 - 3. amount = 2000,02;
 - Limita superioară a EC₃, **MAX_VALUE** (float):
 - 4. amount = MAX_VALUE-0,01;
 - 5. amount = MAX_VALUE;
 - 6. amount = MAX_VALUE+0,01; (non-validă)

BVA. Exemplu 2. Proiectarea cazurilor de testare

- ECs valide identificate:
 - **EC₁**: 0,00 Euro – 1000,00 Euro;
 - **EC₂**: 1000,01 Euro – 2000,00 Euro;
 - **EC₃**: >= 2000,01 Euro.
- Cazuri de testare proiectate pe baza condițiilor BVA identificate:
 - Limita inferioară a EC₁:
 - 1. amount = -0,01; **TC₀₁**: amount = -0,01; (non-valid)
 - 2. amount = 0,00; **TC₀₂**: amount = 0,00 (valid)
 - 3. amount = 0,01; **TC₀₃**: amount = 0,01; (valid)
 - Limita superioară a EC₁:
 - 4. amount = 999,99; **TC₀₄**: amount = 999,99; (valid)
 - 5. amount = 1000,00; **TC₀₅**: amount = 1000,00; (valid)
 - 6. amount = 1000,0; **TC₀₆**: amount = 1000,01; (non valid)
 - similar, se proiectează cazuri de testare valide și non-valide pentru limitele inferioare și superioare ale EC₂ și EC₃;

Condiții BVA. Excepții de identificare a condițiilor BVA

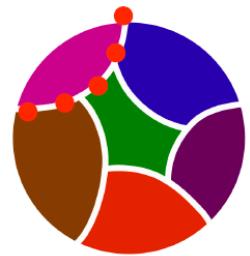
- **există ECs care nu au limite:**
 - E.g.: multimea {Dl, Dna, Dra, Dr.} sau CourseType = {optional, obligatoriu, facultativ};
- **există ECs (ordonate) care nu au două limite** (inferioară și superioară);
 - E.g.: valoarea unei depuneri într-un cont bancar;
- **variabile multiple dependente:**
 - E.g.: variabilele: număr card bancar, data eliberare, data expirare, nume titular;
 - toate variabilele au valori valide și toate constrângerile existente între acestea sunt satisfăcute <==> card valid;
 - dacă variabilele au valori valide dar constrângerile nu sunt satisfăcute ==> card non-valid;
 - dacă variabilele au valori non-valide ==> card non-valid;
- **ECs dependente** – valoarea unei variabile depinde de/ influențează valoarea alteia:
 - E.g.: în OpenOffice Writer există mai multe tipuri de pagină: format_pagină = {A2, A3, A4}; formatul A4 constrânge dimensiunea header-ului paginii (header height) maximă 20.56 cm.

Condiții BVA. Sumar

Tip ECs	Există limite
interval de valori	da
număr de valori	da
multime valori neordonate	nu
multime valori ordonate	da
valoare “must be”	nu
secvență	da
ECs dependente	da
variabile multiple dependente	nu

BVA. Algoritm

- Algoritm de aplicare a BVA (*identificarea condițiilor BVA și proiectarea TCs*):
 1. se identifică limitele tuturor ECs valide de intrare/ieșire;
 2. se scriu condiții BVA pentru fiecare limită a fiecărei EC identificate, astfel încât:
 - valoarea să fie sub limită (mai mică decât limita), e.g., $x < 2$;
 - valoarea să fie pe limită (egală cu limita) , e.g., $x = 2$;
 - valoarea să fie deasupra limitei (mai mare decât limita), e.g., $x > 2$;
 3. se clasifică condițiile BVA în
 - **valide** – corespund unor date de intrare/ieșire valide pentru program;
 - **non-valide** – corespund unor date de intrare/ieșire non-valide pentru program.
 4. se asociază un identificator unic fiecărei condiții BVA (e.g., c1, c2, etc.);
 5. cât timp (*nu au fost descrise cazuri de testare pentru toate condițiile BVA valide/non-valide*):
 - scrie (*un caz de testare nou, care corespunde la cât mai multe condiții BVA valide încă neacoperite*);
 - scrie (*un caz de testare nou, care corespunde doar uneia dintre condițiile BVA non-valide încă neacoperite*).



BVA. Proiectarea cazurilor de testare. Reguli

1. dacă o condiție de intrare/ieșire precizează apartenența la un interval de valori $[a,b]$:

- ==> cazuri de testare pentru:
 - (1) condiții BVA valide - limitele intervalului (e.g., $a, a+1; b-1, b$);
 - (2) condiții BVA non-valide - valori aflate în afara intervalului (e.g., $a-1, b+1$);



2. dacă o condiție de intrare/ieșire precizează o mulțime de valori ordonată:

- ==> cazuri de testare pentru:
 - (1) condiții BVA valide - primul și ultimul element din mulțime;
 - (2) condiții BVA non-valide – valoarea imediat mai mică decât cea mai mică valoare din mulțime și valoarea imediat mai mare decât cea mai mare valoare în mulțime;

3. dacă o condiție de intrare/ieșire precizează numărul de valori (e.g., “de la 1 până la 5 studenți”):

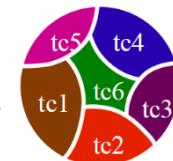
- ==> cazuri de testare pentru:
 - (1) condiții BVA valide – numărul minim și maxim de valori, i.e., 1 și 5;
 - (2) condiții BVA non-valide – valoarea imediat mai mică și imediat mai mare, i.e. 0 și 6;

BVA. Acoperirea testării condițiilor BVA

- **calculul acoperirii (engl. coverage) testării condițiilor BVA:**

$$\text{Acoperirea BVAs} = \frac{\text{numărul de condiții BVA testate}}{\text{numărul de condiții BVA identificate}} \times 100$$

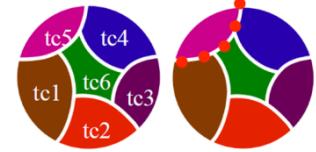
- E.g.:
 - pe baza specificațiilor au fost identificate 64 BVAs (pentru datele de intrare și ieșire, corespunzător ECs valide);
 - 48 BVAs au fost testate prin cazurile de testare proiectate;
 - **Acoperirea BVAs= $(48/64)*100 = 75\%$.**
- **Acoperirea BVAs** poate fi folosită ca și criteriu de terminare a testării, i.e., **exit criteria**.



ECP vs BVA

ECP

- **presupune că programul tratează similar toate valorile din aceeași EC;**
- se poate selecta orice valoare din EC;
- se alege **o singură valoare** din EC, considerată **reprezentativă** pentru a acoperi testarea acelei EC;
- ECs se construiesc pentru condiții de intrare/ieșire valide și non-valide;
- *obiectiv al testării* = verificarea respectării specificațiilor pentru valori uzuale, i.e., **building confidence in software**;



BVA

- **valorile identificate de condițiile BVA sunt prelucrate individual, nu în grup;**
- valorile se găsesc la limitele dintre ECs, acolo unde programul își schimbă comportamentul;
- se iau în considerare valori egale cu limita, valori imediat inferioare și valori imediat superioare limitei;
- sunt luate în considerare atât datele de intrare cât și cele de ieșire, corespunzătoare fiecărei EC valide;
- *obiectiv al testării* = căutarea bug-urilor uzuale, i.e., **bug hunting**;

Testarea Black-box

Avantaje

- nu se există informații despre implementare;
- activitatea testerului este independentă de cea a programatorului;
- reflectă punctul de vedere al utilizatorului;
- surprinde ambiguitățile sau inconsistențele din specificații;
- începe imediat după finalizarea specificațiilor.

Dezavantaje

- dacă specificația *nu* este clară ==> dificultate de construire a cazurilor de testare;
- la execuția programului, multe drumurile din graful de execuție asociat codului rămân netestate ==> secvențele de cod sursă corespunzătoare pot conține bug-uri care nu sunt identificate;
- doar un număr foarte mic de date de intrare va fi efectiv testat.

PENTRU EXAMEN...

Pentru examen...

- **testare:**
 - definiții ale testării (4);
 - terminologie: program, program testat, caz de testare;
 - tipuri de testare: exhaustivă, selectivă;
- **testare black-box:**
 - definiție, caracteristici;
 - ECP, BVA, ECP vs. BVA;
 - aplicarea ECP și BVA pentru probleme concrete;
 - avantaje și dezavantajele BBT.

Cursul următor...

- **Testare White-Box**
 - Tehnici de testare white-box
 - Testare bazată pe fluxul de control. Componente
 - Graful fluxului de control. Drumuri în CFG. Complexitatea ciclometrică
 - Testare bazată pe acoperirea drumurilor
 - Testare bazată pe acoperirea codului sursă
 - Acoperirea instrucțiunilor, deciziilor, condițiilor, deciziilor și condițiilor, condițiilor multiple, drumurilor, buclelor
- **Testare White-box vs Testare Black-box**

Referințe bibliografice

- [Pal2013] Kaushik Pal, *Software Testing: Verification and Validation*,
<http://mrbool.com/software-testing-verification-and-validation/29609>
- [Myers2004] Glenford J. Myers, *The Art of Software Testing*, John Wiley & Sons, Inc., 2004
- [Frentiu2010] M. Frentiu, *Verificarea si validarea sistemelor soft*, Presa Universitara Clujeana, 2010.
- [Patton2005] R. Patton, *Software Testing*, Sams Publishing, 2005.
- [NT2005] K. Naik and P. Tripathy. *Software Testing and Quality Assurance*, Wiley Publishing, 2005.
- [BBST2010] Black-Box Software Testing (BBST), Foundations,
<http://www.testingeducation.org/BBST/foundations/BBSTFoundationsNov2010.pdf>.

CURS 03.

TESTARE WHITE-BOX

Verificarea și validarea sistemelor soft

[14 Martie 2023]

Lector dr. Camelia Chisăliță-Crețu

Universitatea Babeș-Bolyai

Conținut

- **Abordări ale testării**
- **Testare White-Box**
 - Definiție. Caracteristici
 - Tehnici de testare white-box
- **Testare bazată pe fluxul de control. Componente**
 - Definiție. Caracteristici. Avantaje și dezavantaje
 - Graful fluxului de control. Exemple
 - Drumuri în CFG. Exemple
 - Complexitatea ciclomatică. Exemple
- **Testare bazată pe acoperirea drumurilor**
 - Definiție. Algoritm. Exemplu
- **Testare bazată pe acoperirea codului sursă**
 - Definiție. Criterii de acoperire
 - Acoperirea instrucțiunilor, deciziilor, condițiilor, deciziilor și condițiilor, condițiilor multiple, buclelor
- **Testare White-box vs Testare Black-box**
 - Avantaje și dezavantaje ale Testării White-Box
 - Testare Black-box vs. Testare White-box
- **Bibliografie**

ABORDĂRI ALE TESTĂRII

Abordări ale testării. Clasificare

Tehnici de testare asociate

Abordări ale testării. Clasificare

- abordare a testării
 - modalitate de realizare a testării în care se aplică una sau mai multe tehnici de testare în cadrul unei strategii de testare stabilită anterior;
- clasificare
 - testare Black-box (**criteriul cutiei negre**, *engl. Black-box testing*);
 - testare White-box (**criteriul cutiei transparente**, *engl. White-box testing*);
 - testare Grey-box (**criteriul cutiei gri**, *engl. Grey-box testing*);
 - testare bazată pe experiență (*engl. Experienced-based testing*);
 - testare bazată pe scripturi (*engl. Scripted-based testing*);

Abordări ale testării. Tehnici de testare asociate

- Testare Black-Box – testare funcțională:
 - Partiționarea în clase de echivalență;
 - Analiza valorilor limită;
 - Tabele de decizie, Cazuri de utilizare, Scenarii de utilizare, etc.;
- Testare White-box – testare structurală:
 - Acoperirea fluxului de control (e.g., instrucțiuni, ramificații, decizii, condiții, bucle, drumuri);
 - Acoperirea fluxului de date;
- Testare Grey-box – testare mixtă:
 - folosirea simultană a avantajelor abordărilor black-box și white-box pentru proiectarea cazurilor de testare.

TESTARE WHITE-BOX

Definiție. Caracteristici

Tehnici de testare white-box

Testare White-Box. Definiție. Caracteristici

- **criteriul cutiei transparente (engl. white-box testing, logic driven testing):**
 - testare structurală;
 - datele de intrare se aleg pe baza instrucțiunilor care trebuie executate, programul este văzut ca o cutie transparentă;
 - avem acces la structura internă a programului (codul sursă);
 - permite identificarea situațiilor în care execuția programului nu acoperă diferite structuri ale acestuia.

Tehnici de testare white-box

- tehnici de proiectare a cazurilor de testare white-box bazate pe:
 1. fluxul de control:
 - acoperirea drumurilor [[NT2005](#)];
 - acoperirea codului sursă:
 - instrucțiunilor, ramificațiilor, deciziilor, condițiilor, deciziilor și condițiilor multiple [[Myers2004](#)], condițiilor/deciziilor modificate;
 - buclelor [[Beizer1990](#)];
 - acoperirea predicatelor (*engl. predicate complete coverage*);
 - acoperirea prin mutații;
 2. fluxul de date.

TESTARE BAZATĂ PE FLUXUL DE CONTROL. COMPOUNTE

Definiție. Caracteristici

Graful fluxului de control. Exemple

Drumuri în CFG. Exemple

Complexitatea ciclomatică. Exemple

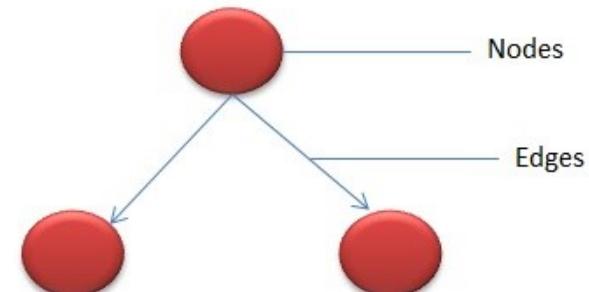
Testare bazată pe fluxul de control – Avantaje și dezavantaje

Testarea bazată pe fluxul de control

- **testarea bazată pe fluxul de control**
 - utilizează **structurile de control** pentru proiectarea cazurilor de testare;
 - **scop:** acoperirea prin cazuri de testare la un nivel satisfăcator a structurilor de control din programul testat;
- **componente:**
 - graful fluxului de control;
 - complexitatea ciclomatică.

Graful fluxului de control. Definiție

- **graful fluxului de control (engl. Control Flow Graph, CFG):**
 - reprezentare grafică detaliată a unei unități de program;
 - permite vizualizarea tuturor drumurilor din unitatea de program;
- **graf orientat:**
 - **vârf (engl. node):**
 - indică structuri secvențiale și condițiile din structurile alternative sau repetitive;
 - **arc (engl. edge):**
 - indică sensul transmiterii controlului logic în cadrul programului.



CFG. Caracteristici

- permite reprezentarea grafică a structurilor de programare;
- tipuri de vârfuri:

- **decizie:**

- are o condiție prin care se permite ramificarea execuției prin cel puțin două căi;
 - e.g., instrucțiunile `if`, `while`, `repeat/until`, `case`;

- **instrucțiune/calcul:**

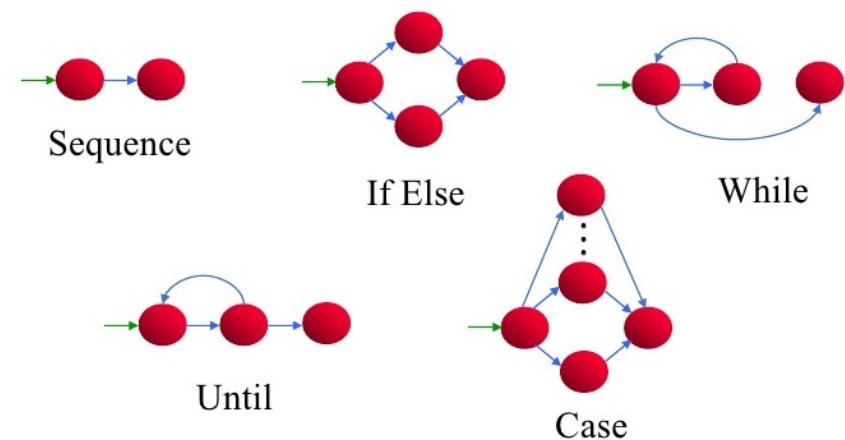
- conține o secvență de instrucțiuni;

- **conector:**

- nu conține o instrucțiune și reprezintă un punct al programului care unește mai multe ramificații;

- **intrare, ieșire:**

- există un singur vârf intrare și un singur vârf ieșire;
 - în vârful de intrare nu intră nici un arc;
 - din vârful de ieșire nu ieșe nici un arc.



CFG. Construire

- pași de elaborare a unui CFG:
 1. se numerotează unic fiecare element de structură secvențială (calcul) și condițională (decizie);
 2. se începe pornind de la vârful de intrare, care are (de obicei) numărul 1;
 3. se adaugă celelalte vârfuri corespunzătoare structurilor numerotate și se unesc prin arce, evidențiind transmiterea controlului în cadrul programului;
 4. la final, toate ieșirile posibile din program se unesc în vârful de ieșire;
- condiții complexe care conțin atribuiră ==> CFG are o descriere complexă [[NT2005](#)];
 - e.g., if (((fptr1 = fopen("file1", "r")) != NULL) && (i++) && (0)) { ... }.

CFG. Complexitatea construirii

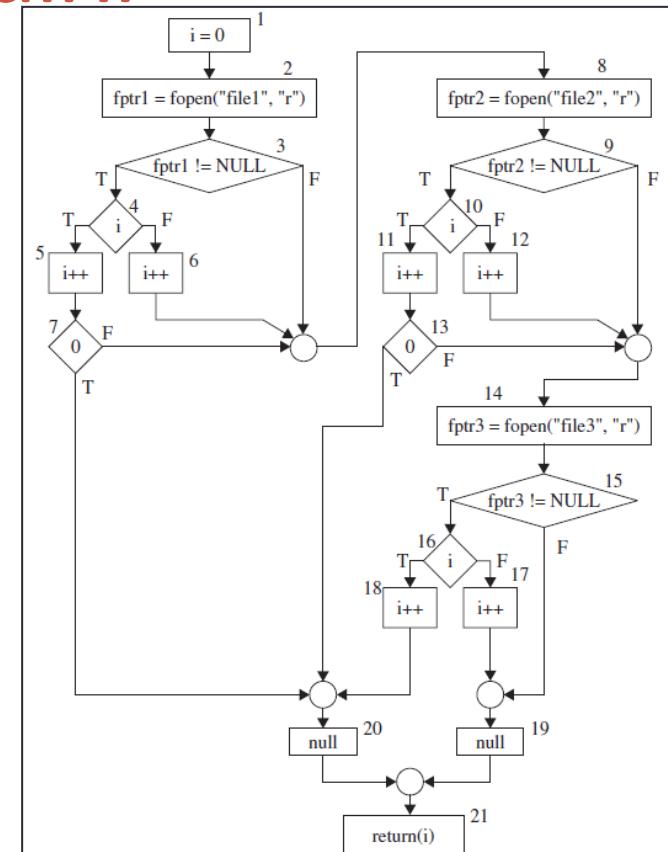
- pentru metoda `openfiles()`, avem CFG alăturat;

```
FILE *fptr1, *fptr2, *fptr3; /* These are global variables. */

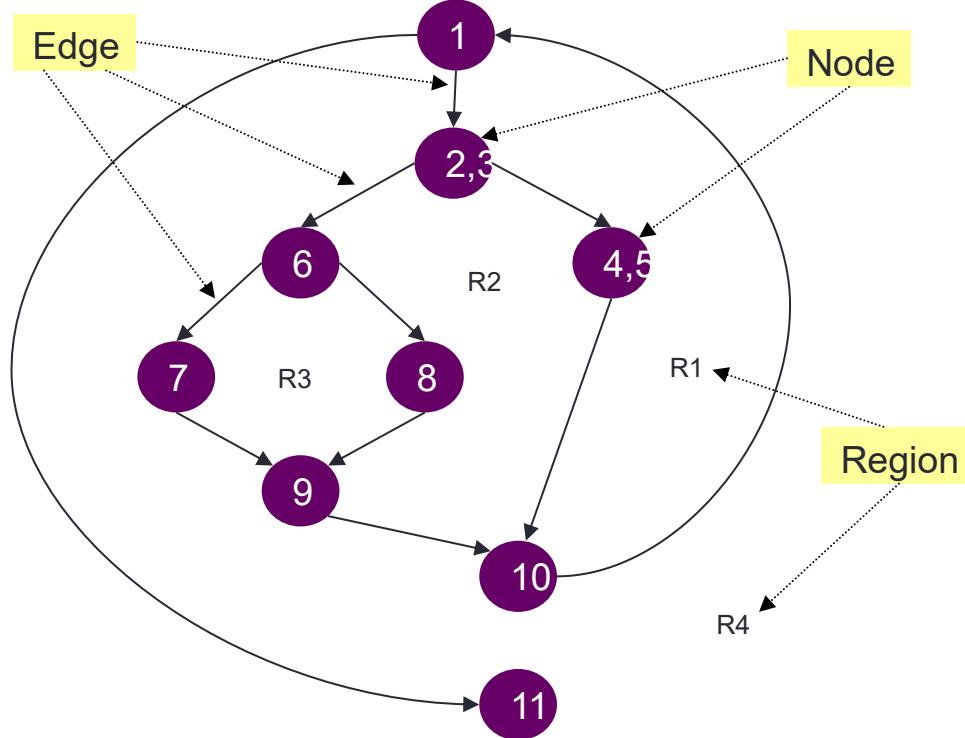
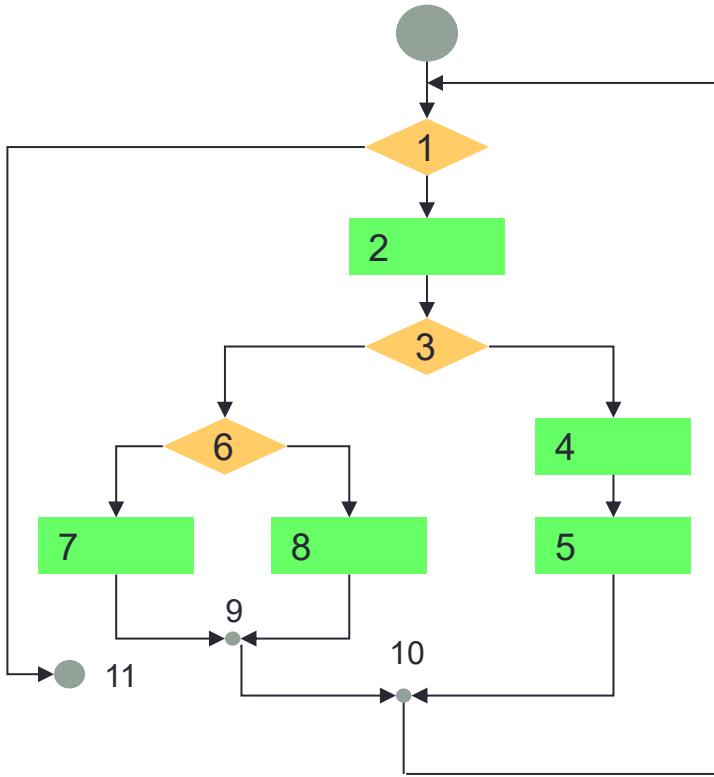
int openfiles(){
/*
    This function tries to open files "file1", "file2", and
    "file3" for read access, and returns the number of files
    successfully opened. The file pointers of the opened files
    are put in the global variables.
*/
    int i = 0;
    if(
        ((( fptr1 = fopen("file1", "r")) != NULL) && (i++)  

                     && (0)) ||
        ((( fptr2 = fopen("file2", "r")) != NULL) && (i++)  

                     && (0)) ||
        ((( fptr3 = fopen("file3", "r")) != NULL) && (i++)))
    );
    return(i);
}
```

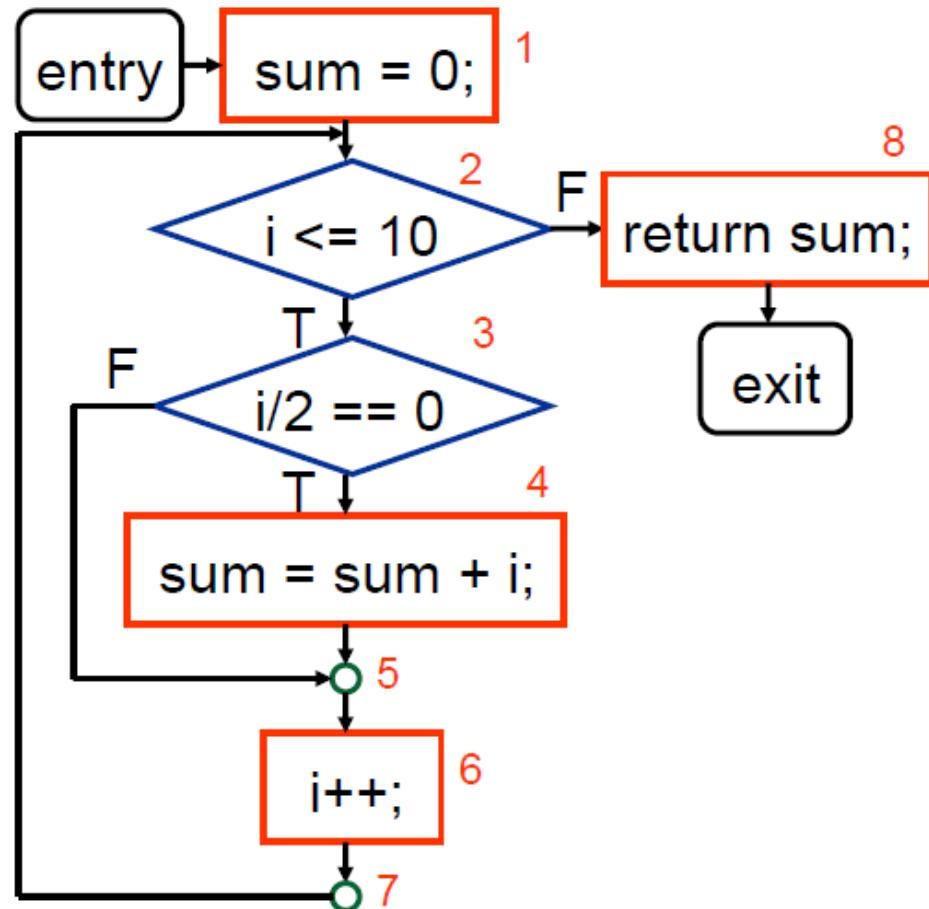


CFG. Exemple de notății

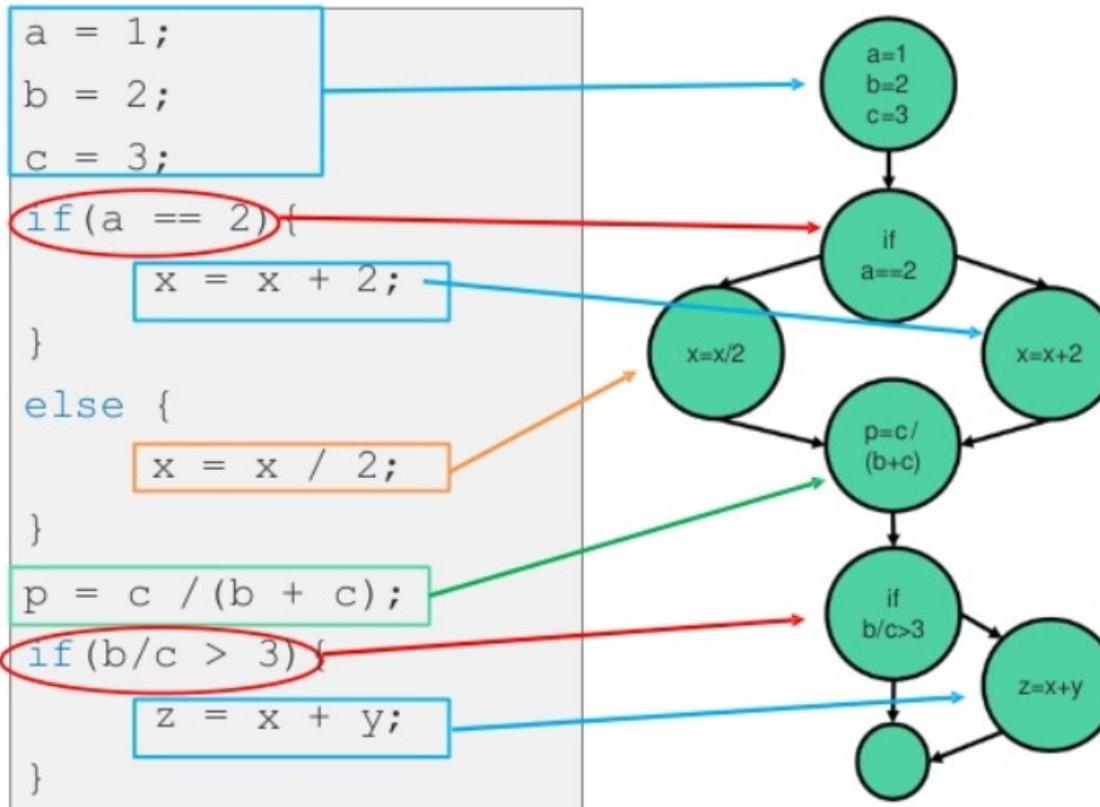


CFG. Exemple (1)

```
* int evenSum(int i) {  
1     int sum = 0;  
2     while (i <= 10) {  
3         if (i/2 == 0) {  
4             sum = sum + i;  
5         }  
6         i++;  
7     }  
8     return sum;  
* }
```



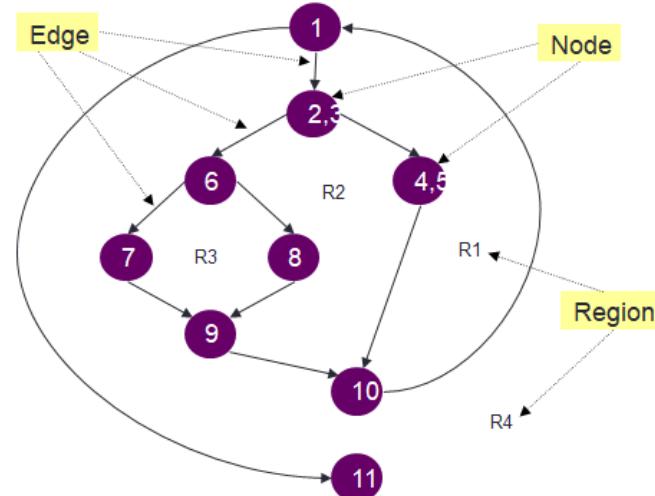
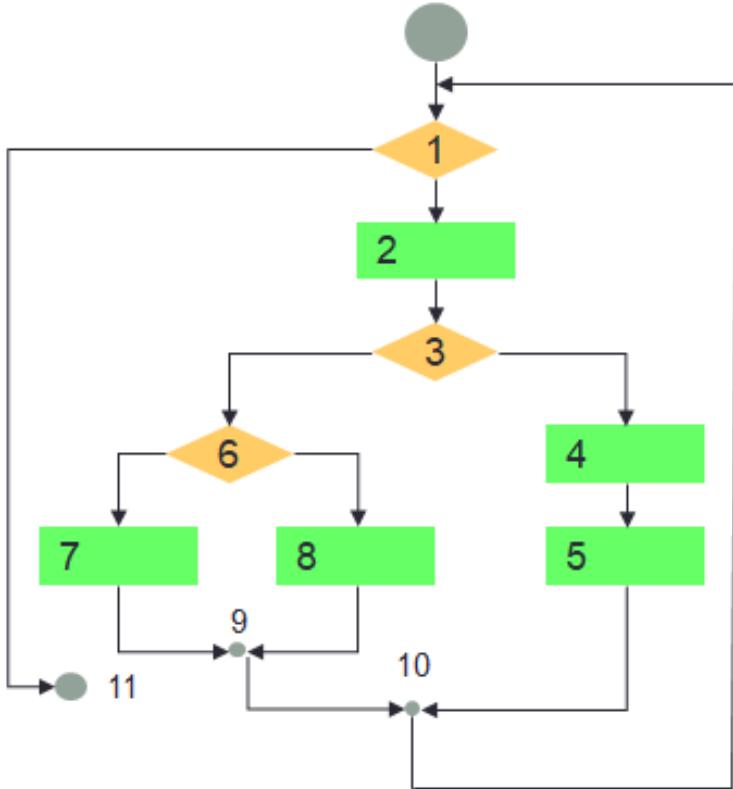
CFG. Exemple (2)



CFG. Drumuri în CFG

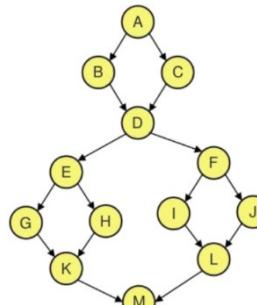
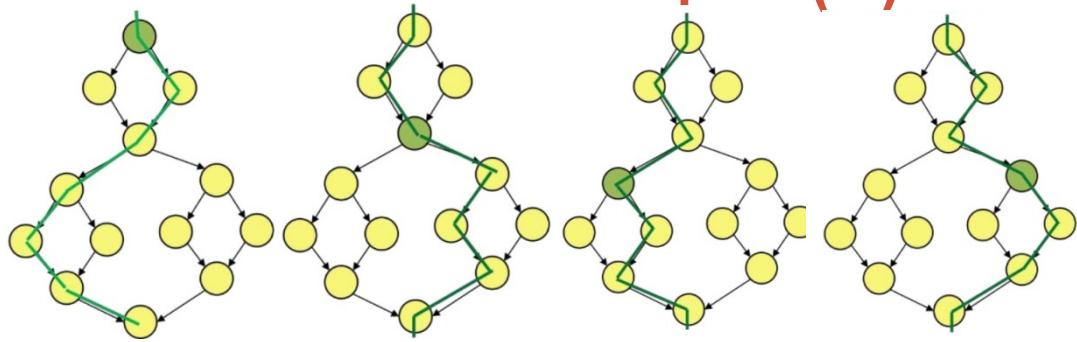
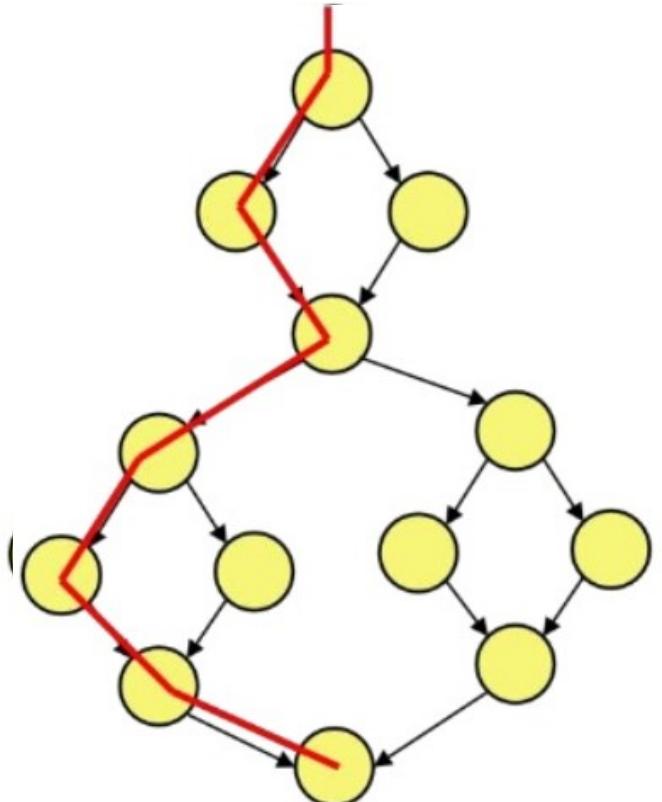
- **drum:**
 - execuția unei secvențe de instrucțiuni **de la punctul de intrare până la punctul de ieșire** al CFG asociat unei unități de program;
- **drum independent (engl. independent path):**
 - orice drum în CFG care introduce **cel puțin o instrucțiune nouă sau o condiție nouă**, care este executată cel puțin o dată;
- mulțimea drumurilor independente formează **mulțimea drumurilor de bază (engl. basis path set)** a unui CFG;
 - indică **numărul minim de cazuri de testare** care trebuie executate pentru ca fiecare instrucțiune să fie executată cel puțin o dată.

Drumuri independente în CFG. Exemple (1)



- drumuri independente:
 - **drum 1:** 1(F)-11.
 - **drum 2:** 1(T)-2-3(T)-4-5-10-1(F)-11.
 - **drum 3:** 1(T)-2-3(F)-6(T)-8-9-10-1(F)-11.
 - **drum 4:** 1(T)-2-3(F)-6(F)-7-9-10-1(F)-11.

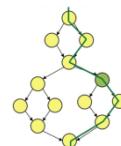
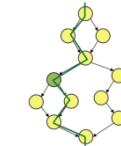
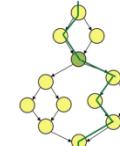
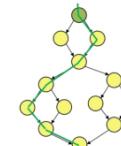
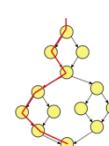
Drumuri independente în CFG. Exemple (2)



- drumuri independente:
 - **drum 1:** A-B-D-E-G-K-M;
 - **drum 2:** A-C-D-E-G-K-M;
 - **drum 3:** A-B-D-F-I-L-M;
 - **drum 4:** A-B-D-E-H-K-M;
 - **drum 5:** A-C-D-F-J-L-M.

CFG. Algoritm de construire a drumurilor independente

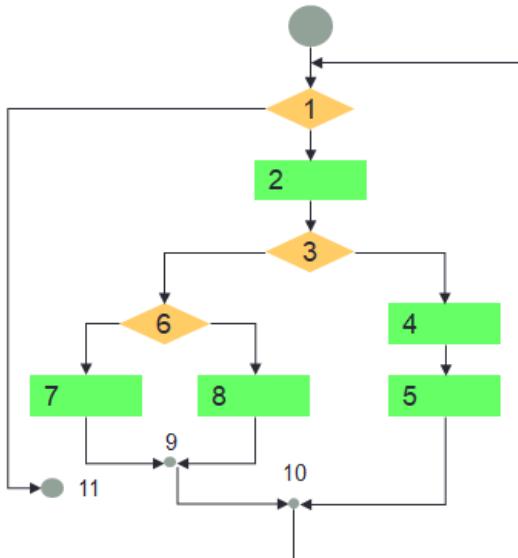
- **Algoritmul lui McCabe [McCabe1984, McCabe Baseline Method]**
 1. se alege un drum normal (numit **drum inițial, D1**); se recomandă ca acesta să aibă cât mai multe decizii este posibil;
 2. pentru generarea următorului drum (**D2**), se modifică rezultatul evaluării primei decizii de pe D1 și păstrând numărul de decizii ale drumului D1;
 3. pentru generarea următorului drum (**D3**), se modifică rezultatul evaluării celei de a doua decizii de pe D1;
 4. se repetă pasul 3 până când toate deciziile de pe D1 au fost modificate/inversate;
 5. se reiau pașii de la 1..4 considerând ca drum inițial pe D2, modificând/inversând deciziile, până când toate se obțin toate drumurile independente din mulțimea de bază a CFG.



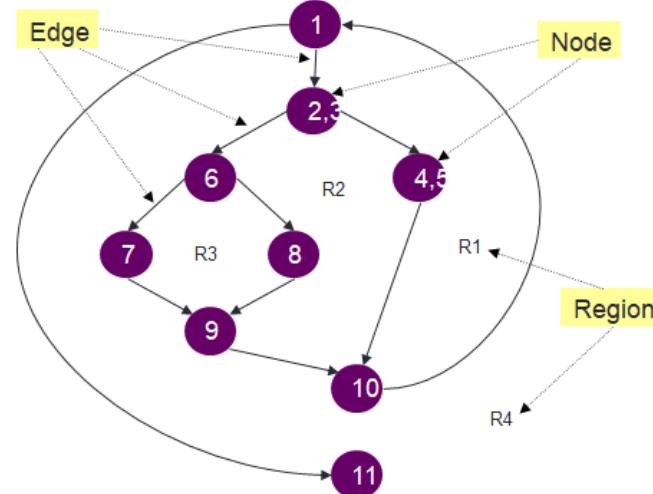
Complexitatea ciclometrică. Definiție

- **complexitatea ciclometrică** (engl. McCabe's cyclomatic complexity, CC):
 - *metrică software* aplicată pentru măsurarea cantitativă a complexității logice a unui program;
 - *permite determinarea numărului de drumuri independente din mulțimea de bază a unui CFG;*
- modalități de calcul a CC la nivelul CFG:
 - **CC = numărul de regiuni din CFG;**
 - **CC = E – N + 2, unde E - #arce, N - #vârfuri ;**
 - **CC = P + 1, unde P - #vârfuri condiție.**
- **regiune:**
 - zonă a CFG marginită parțial sau în totalitate de arce și vârfuri;

CC. Exemple (1)

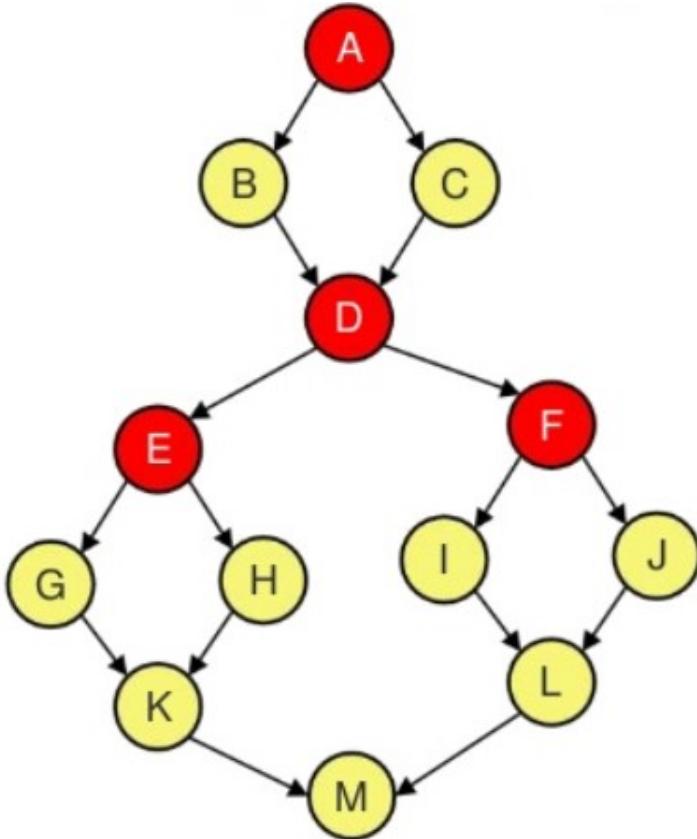


- CC pentru CFG:
 - $CC = \text{numărul de regiuni} = 4$ regiuni = 4.
 - $CC = E - N + 2 = 14 \text{ arce} - 12 \text{ vârfuri} + 2 = 4$.
 - $CC = P + 1 = 3 \text{ vârfuri condiție} + 1 = 4$.



- drumuri independente:
 - **drum 1:** 1(F)-11.
 - **drum 2:** 1(T)-2-3(T)-4-5-10-1(F)-11.
 - **drum 3:** 1(T)-2-3(F)-6(T)-8-9-10-1(F)-11.
 - **drum 4:** 1(T)-2-3(F)-6(F)-7-9-10-1(F)-11.

CC. Exemple (2)



- drumuri independente:
 - drum 1: A-B-D-E-G-K-M;
 - drum 2: A-C-D-E-G-K-M;
 - drum 3: A-B-D-F-I-L-M;
 - drum 4: A-B-D-E-H-K-M;
 - drum 5: A-C-D-F-J-L-M.
- CC pentru CFG:
 - $CC = \text{numărul de regiuni} = 5$ regiuni = 5;
 - $CC = E - N + 2 = 16$ arce - 13 vârfuri + 2 = 5;
 - $CC = P + 1 = 4$ vârfuri condiție + 1 = 5.

Utilizarea CFG în testare. Avantaje și dezavantaje

Avantaje

- Testarea de bază aplicată în **testarea unitară**, care sunt dezvoltate la momentul curent;
- Se aplică pentru modulele pentru care **prin inspectare nu pot fi suficient verificate**;
- Limbajele orientate-obiect reduc numărul de bug-uri la nivelul fluxului de control.

Dezavantaje

- Dacă este aplicată de tester și nu de programator, este necesar ca testerul să aibă abilități de programare pentru a înțelege codul sursă și modul de execuție al acestuia;
- **Tehnică de testare consumatoare de timp**, deoarece mai întâi se elaborează CFG, CC, drumuri independente și ulterior se proiectează cazurile de testare;
- Consideră că:
 - Specificațiile sunt corecte;
 - Datele sunt definite și accesate corespunzător;
 - Nu există alte bug-uri pe lângă cele determinate de fluxul de control.

TESTARE BAZATĂ PE ACOPERIREA DRUMURILOR

Definiție

Algoritm

Exemplu

Testare bazată pe acoperirea drumurilor. Definiție

- **acoperirea tuturor drumurilor** (*engl. all path coverage, apc*):
 - testarea tuturor drumurilor programului;
- avantaje și dezavantaje:
 - permite identificarea tuturor defectelor, dar **nu și de pe drumurile care lipsesc**;
 - **dificil** de realizat în practică pentru **programe cu structuri repetitive** ==> se alege un număr redus de drumuri;

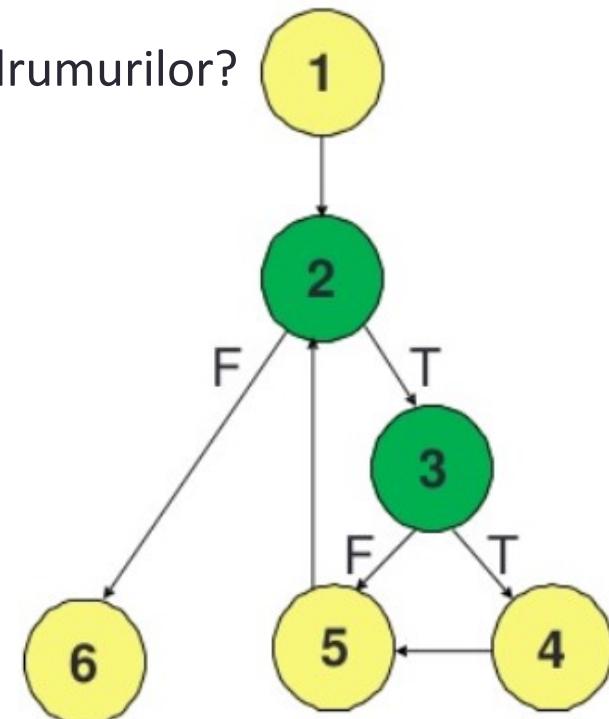
Testare bazată pe acoperirea drumurilor. Algoritm

- Algoritmul de proiectare a cazurilor de testare bazat pe drumuri este:
 1. Se elaborează CFG;
 2. Se calculează CC pe baza CFG;
 3. Se determină mulțimea de bază a CFG (cu drumuri independente, liniare);
 4. Se proiectează câte un caz de testare pentru fiecare drum independent identificat.
- ordinea de selectare a drumurilor:
 - drumuri scurte;
 - drumuri de lungime crescândă;
 - drumuri lungi, complexe, alese arbitrar.

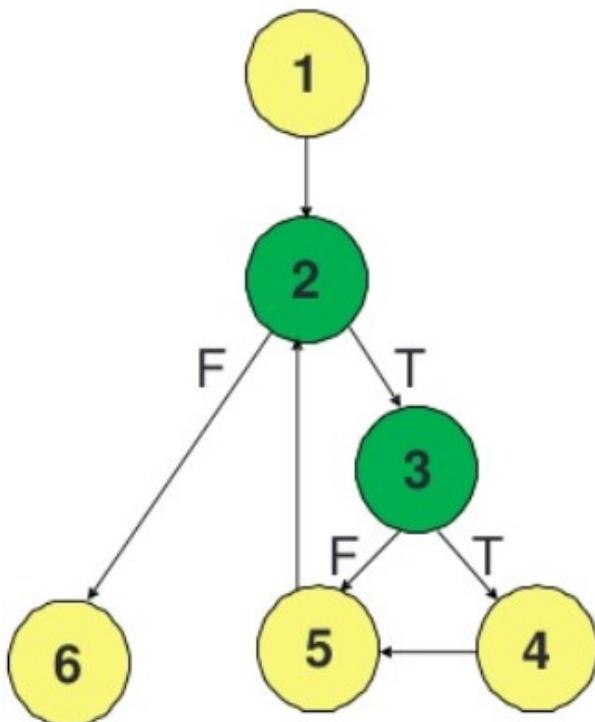
Testare bazată pe acoperirea drumurilor. Exemplu

Câte cazuri de testare sunt necesare pentru acoperirea drumurilor?

```
const int SIZE = 10;  
int i;  
int array[SIZE] = {52, 88, 90, 21, 62, 10, 16, 39, 45, 80};  
int min = array[0];  
while(i < 10) {  
    if(array[i] < min) {  
        min = array[i];  
    }  
    i = i + 1;  
}  
cout << min;
```



Testare bazată pe acoperirea drumurilor. Exemplu (cont)



- CC pentru CFG:
 - $CC = \text{numărul de regiuni} = 3 \text{ regiuni} = 3;$
 - $CC = E - N + 2 = 7 \text{ arce} - 6 \text{ vârfuri} + 2 = 3;$
 - $CC = P + 1 = 2 \text{ vârfuri condiție} + 1 = 3.$
- drumuri independente:
 - **drum 1:** 1-2(F)-6.
 - **drum 2:** 1-2(T)-3(F)-5-2(F)-6.
 - **drum 3:** 1-2(T)-3(T)-4-5-2(F)-6.

TESTARE BAZATĂ PE ACOPERIREA CODULUI SURSĂ

Definiție. Criterii de acoperire

Acoperirea instrucțiunilor. Definiție. Exemplu

Acoperirea deciziilor. Definiție. Exemplu

Acoperirea condițiilor. Definiție. Exemplu

Acoperirea deciziilor și condițiilor. Definiție. Exemplu

Acoperirea condițiilor multiple. Definiție. Exemplu

Acoperirea buclelor. Definiție. Exemplu

Testare bazată pe acoperirea codului sursă. Definiție

- **acoperirea codului sursă:**
 - testarea tuturor structurilor de control folosind un număr minim de teste, astfel încât să fie satisfăcute criteriile:
 - **acoperirea instrucțiunilor** (*engl. statement/line/node coverage*);
 - acoperirea ramificațiilor:
 - **acoperirea deciziilor** (arcelor) (*engl. decision/branch/edge coverage*);
 - **acoperirea condițiilor** (*engl. condition coverage*);
 - **acoperirea deciziilor și condițiilor** (*engl. decision-condition coverage*);
 - **acoperirea condițiilor multiple** (*engl. multiple condition coverage*);
 - acoperirea structurilor repetitive:
 - **acoperirea buclelor** (*engl. loop coverage*).

Acoperirea instrucțiunilor. Definiție

- **acoperirea instrucțiunilor (engl. statement/line/node coverage, sc):**
 - proiectarea cazurilor de testare astfel încât toate instrucțiunile sunt executate cel puțin o dată, adică fiecare vârf al CFG este vizitat;
 - **cel mai slab criteriu de acoperire în testare;**
 - o mulțime de teste care nu realizează acoperire 100% a vârfurilor nu este considerată acceptabilă.

SC. Exemplu

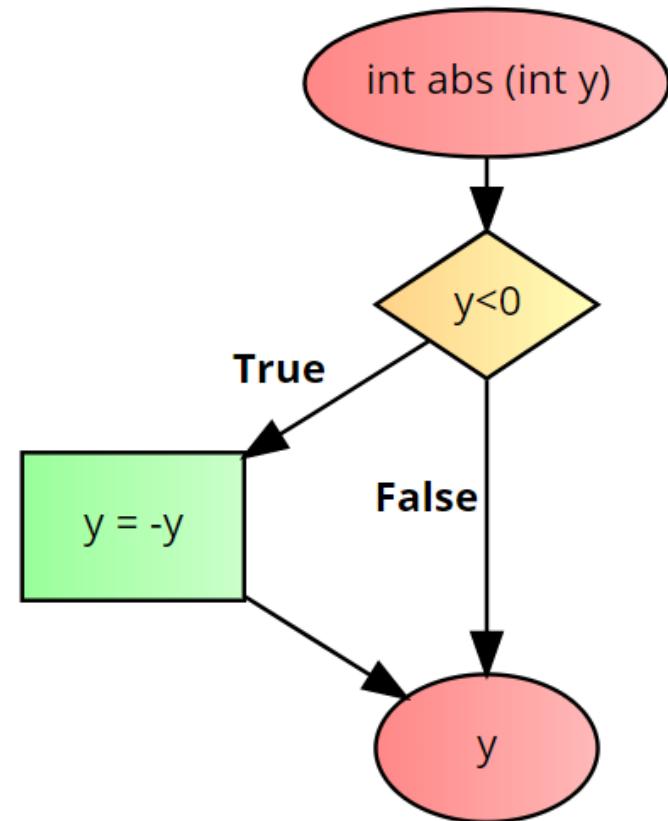
Fiecare intrucțiune trebuie să fie executată cel puțin o dată.

```
// returnează valoarea absolută a lui y  
int abs (int y) {  
    if (y<0)  
        y = -y;  
    return y;  
}
```

Care este numărul minim de cazuri de testare necesar?

TC	Input	Expected result	Actual result
1	-2	2	2

Criteriul de acoperire a instrucțiunilor este îndeplinit 100%.

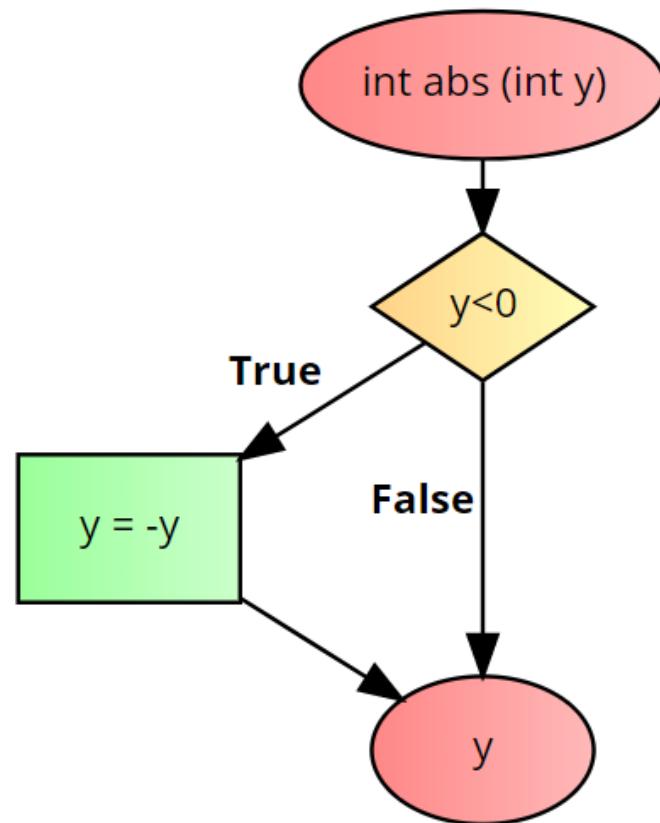


SC. Observații

- cel mai slab criteriu de acoperire deoarece:
 - nu acoperă ramificația `else` pentru instrucțiunile `if` care nu descriu explicit această ramificație; nu evidențiază **implicit** prezența posibilelor bug-uri de pe aceste ramificații;
- SC se recomandă doar atunci când nu există alte criterii de acoperire care se pot aplica.

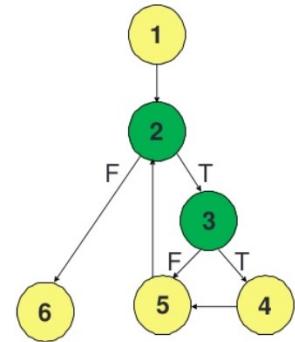
```
// returnează valoarea absolută a lui y
int abs (int y) {
    if (y<0)
        y = -y;
    return y;
}
```

Este necesară acoperirea deciziilor.



Acoperirea deciziilor. Definiție

- **ramificație (engl. branch/edge):**
 - arc care pornește dintr-un vârf;
 - din fiecare vârf pornește cel mult un arc, mai puțin din vârful de ieșire al CFG;
 - din vârfurile de decizie pornesc două arce, etichetate cu true și false;
- **acoperirea deciziilor (engl. branch/edge/decision coverage, dc):**
 - acoperirea unui arc $a =$ drum care parcurge arcul a ;
 - proiectarea cazurilor de testare se face astfel încât *fiecare arc de decizie* să fie parcurs cel puțin o dată;
- **regulă de selectare:**
 - fiecare decizie selectată, evaluată la true sau false, trebuie să se găsească pe cel puțin un drum.



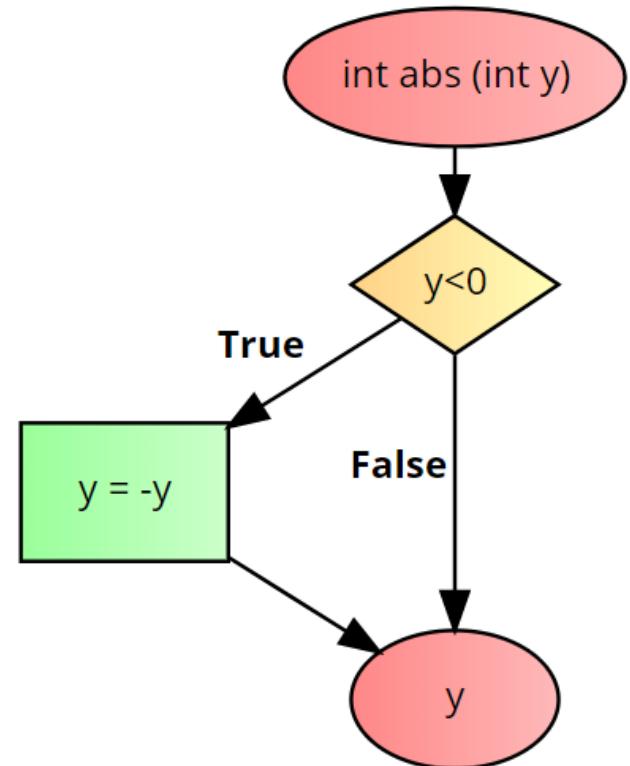
DC. Exemplu

Pentru fiecare decizie, fiecare ramificație (true, false) trebuie să fie executată cel puțin o dată.

```
// returnează valoarea absolută a lui y  
int abs (int y) {  
    if (y<0)  
        y = -y;  
    return y;  
}
```

Care este numărul minim de cazuri de testare necesar ?

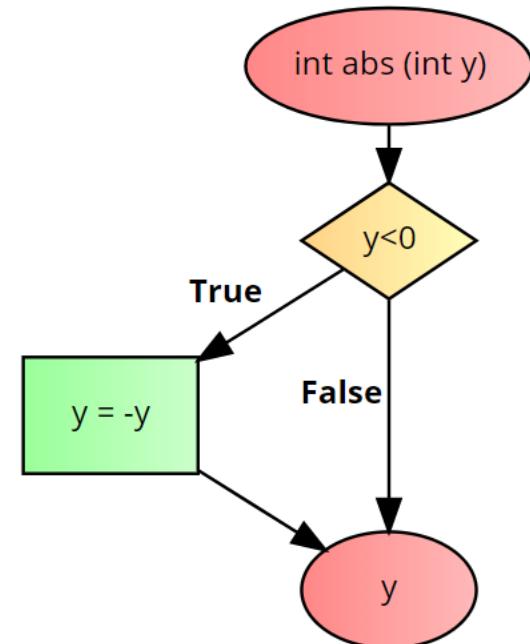
TC	Input	Decision (y<0)	Expected result	Actual result
1	-2	true	2	2
2	3	false	3	3



Criteriul de acoperire a deciziilor este îndeplinit 100%. Ambele ramificații ale deciziei au fost explorate.

DC vs. SC

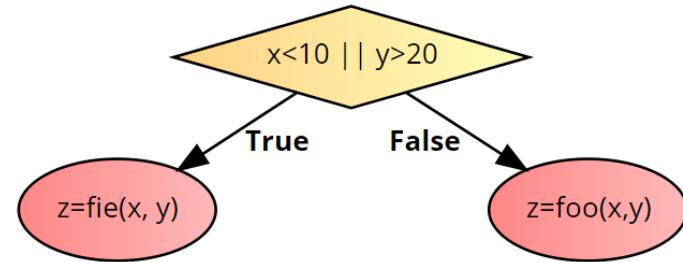
- **dc ==> sc;**
 - instrucțiunile se află pe arce; dacă se parcurge fiecare arc atunci se execută și instrucțiunile asociate;



DC. Observații

- e.g.,

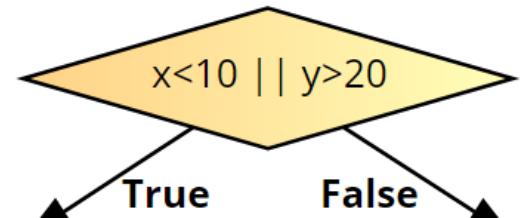
```
y = fou(x);  
if (x<10 || y>20)  
    { z=fie(x, y); }  
else { z=foo(x, y); }
```
- pentru $x=1$ și $y=2$ nu mai este relevantă evaluarea condiției $y>20$;
- în deciziile formate din mai multe condiții, unele condiții pot să rămână neacoperite fiind **irrelevant**e pentru rezultatul final al deciziei;
- dacă condiția este scrisă greșit, e.g., $y<20$ în loc de $y>20$, cazuri de testare ca $x=1$, $y=2$ nu evidențiază defectul.



Este necesară acoperirea condițiilor.

Acoperirea condițiilor. Definiție

- **condiție:**
 - expresie logică dintr-un vârf de decizie;
 - o decizie este formată din una sau mai multe condiții;
- **acoperirea condițiilor (engl. condition coverage, cc):**
 - proiectarea cazurilor de testare se realizează astfel încât fiecare condiție din fiecare decizie ia fiecare dintre valorile posibile, cel puțin o dată;
- **regulă de selectare:**
 - pentru fiecare decizie care conține mai multe condiții, fiecare condiție selectată va fi evaluată la true sau false și se va găsi pe cel puțin un drum.

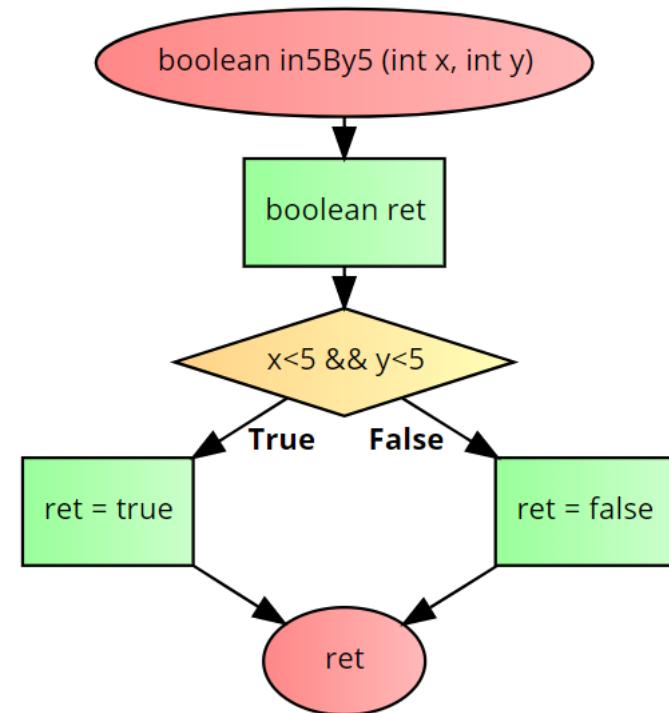


CC. Exemplu

Fiecare condiție din fiecare decizie trebuie să fie executată cel puțin o dată cu fiecare din valorile posibile (e.g., true, false)

```
// returnează true dacă (x,y) este în cadranul (5,5).  
boolean in5By5 (int x, int y) {  
    boolean ret;  
    if (x<5 && y<5)  
        ret = true;  
    else ret = false;  
    return ret;  
}
```

Care este numărul minim de cazuri de testare necesar ?



CC. Exemplu (cont.)

// returnează true dacă (x,y) este în cadranul (5,5).

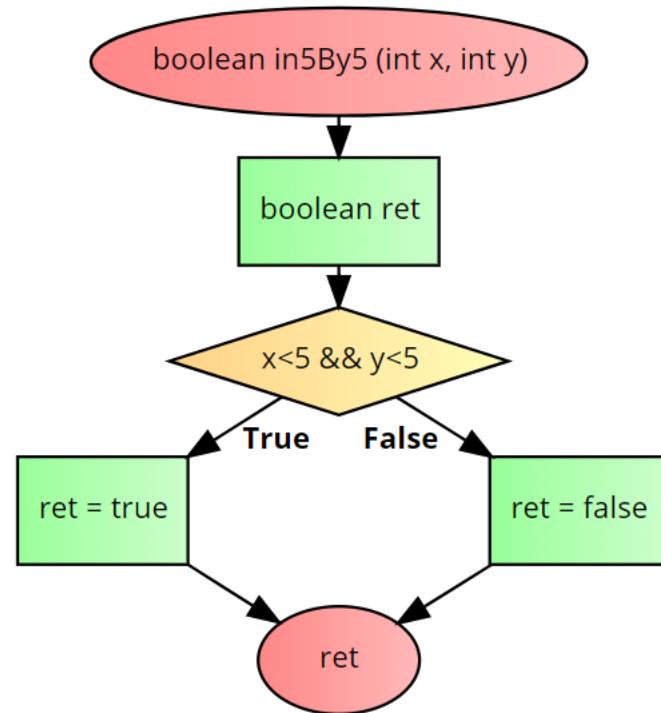
```
boolean in5By5 (int x, int y) {  
    boolean ret;  
    if (x<5 && y<5)  
        ret = true;  
    else ret = false;  
    return ret;  
}
```

Care este numărul minim de cazuri de testare necesar ?

TC	x	y	Decision ($x < 5$) && ($y < 5$)	Expected result	Actual result
1	2	9	T && F = F	false	false
2	9	2	F && T = F	false	false

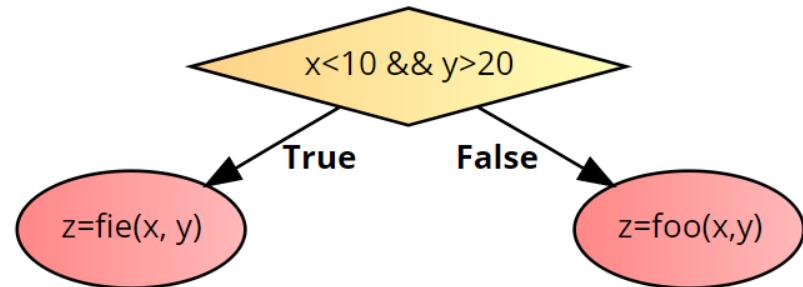
Criteriul de acoperire a condițiilor este îndeplinit 100%.

Toate rezultatele posibile ale evaluării condițiilor din decizie au fost explorate.



CC. Observații

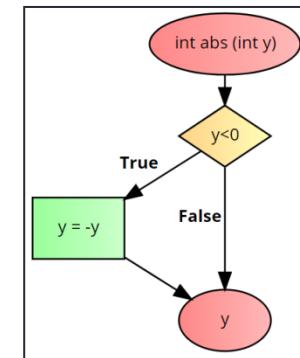
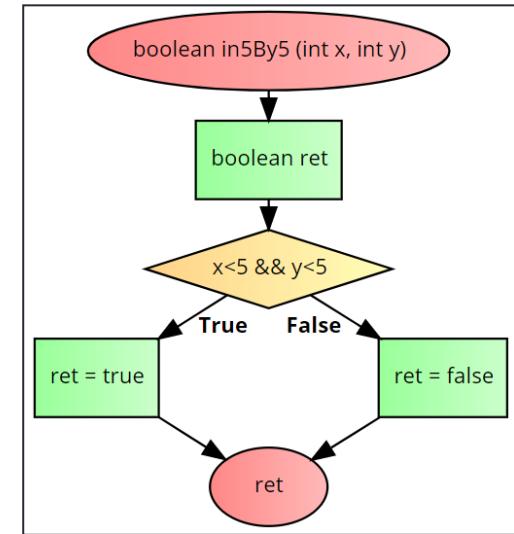
- e.g.,
if ($x < 10 \&\& y > 20$)
 { $z = \text{fie}(x, y)$; }
 else { $z = \text{foo}(x, y)$; }
- pentru $x=11$ și $y=21$, avem false $\&\&$ true = **false**;
- pentru $x=1$ și $y=1$, avem true $\&\&$ false = **false**;
- fiecare *condiție* selectată este acoperită prin evaluarea la true și false, dar *decizia* nu este acoperită, doar ramificația false este explorată;



Este necesară acoperirea deciziilor și condițiilor.

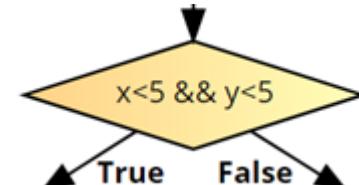
CC vs. DC

- În general, cc ==> dc;
 - prin acoperirea condițiilor se poate acoperi și decizia;
- caz particular:
 - cc = dc atunci când decizia conține doar o condiție;
 - e.g., decizia $y < 0$ este evaluată la true sau false, similar cu evaluarea condiției $y < 0$, care este evaluată la true sau false ==> acoperirea condiției este similară cu acoperirea deciziei;



Acoperirea deciziilor și condițiilor. Definiție

- **acoperirea deciziilor și condițiilor (engl. decision and condition coverage, **dcc**):**
 - proiectarea cazurilor de testare astfel încât:
 - fiecare condiție din fiecare decizie ia toate valorile posibile, cel puțin o dată;
 - fiecare decizie ia toate valorile posibile cel puțin o dată;
- regulă de selectare:
 - pentru fiecare decizie care conține mai multe condiții, fiecare condiție selectată va fi evaluată la true sau false și împreună cu decizia evaluată la true sau false se vor găsi pe cel puțin un drum.



DCC. Exemplu

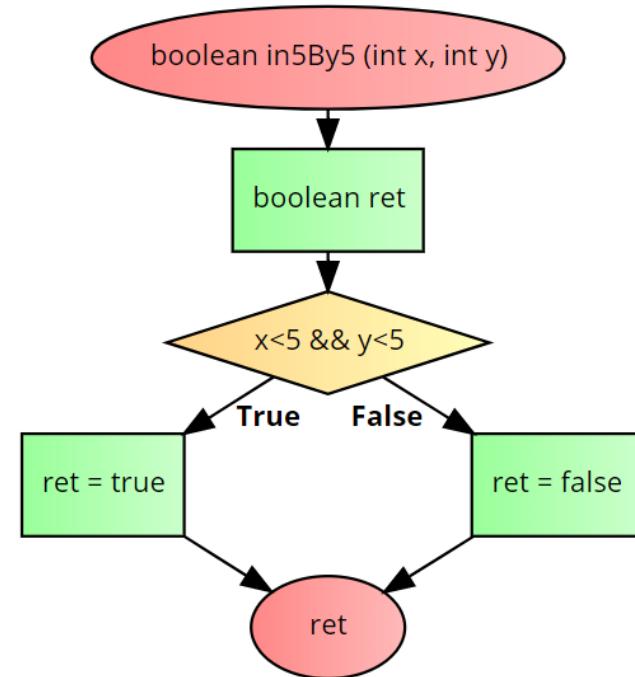
Fiecare *condiție* din fiecare decizie trebuie să fie executată cel puțin o dată cu fiecare din valorile posibile, e.g., **true, false**.

Fiecare *decizie* trebuie să fie executată cel puțin o dată cu fiecare din valorile posibile, e.g., **true, false**.

// returnează true dacă (x,y) este în cadranul (5,5).

```
boolean in5By5 (int x, int y) {  
    boolean ret;  
    if (x<5 && y<5)  
        ret = true;  
    else ret = false;  
    return ret;  
}
```

Care este numărul minim de cazuri de testare necesar ?



DCC. Exemplu (cont.)

```
// returnează true dacă (x,y) este în cadranul (5,5).  
boolean in5By5 (int x, int y) {  
    boolean ret;  
    if (x<5 && y<5)  
        ret = true;  
    else ret = false;  
    return ret;  
}
```

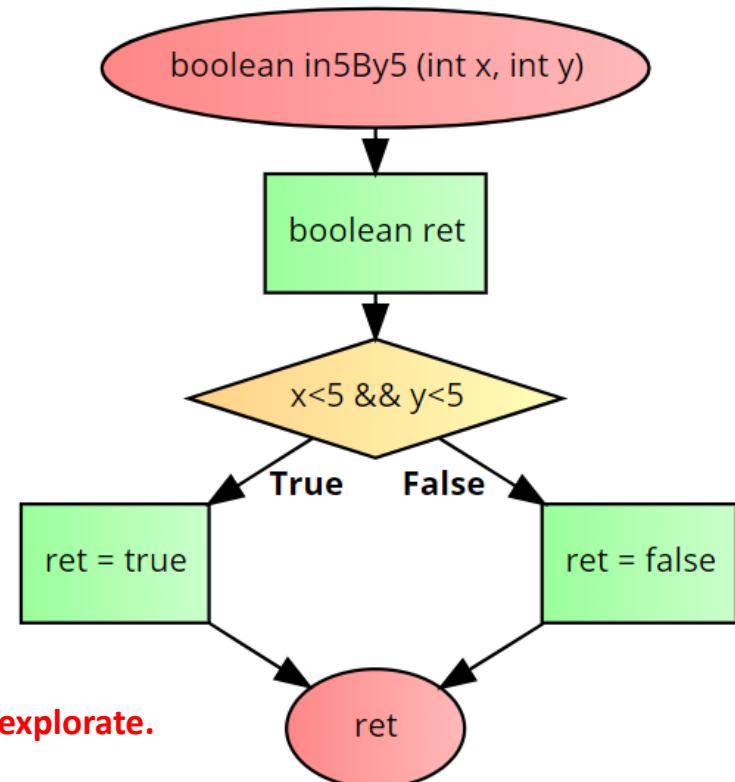
Care este numărul minim de cazuri de testare necesar ?

TC	x	y	Decision (x<5) && (y<5)	Expected result	Actual result
1	2	3	T && T = T	true	true
2	9	7	F && F = F	false	false

Criteriul de acoperire a deciziilor și condițiilor este îndeplinit 100%.

Toate rezultatele posibile ale evaluării condițiilor din decizie au fost explorate.

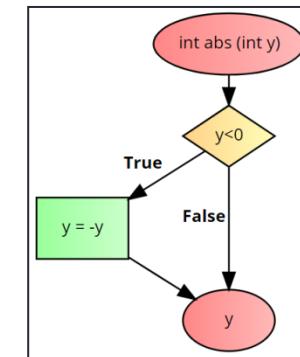
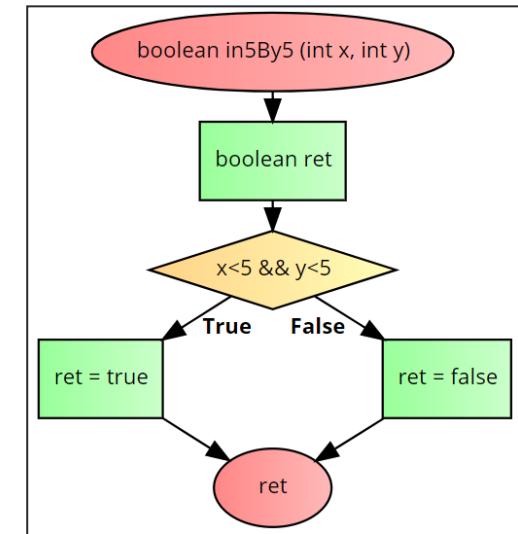
Ambele ramificații ale deciziei au fost explorate.



DCC vs. CC, DCC vs. DC

- **dcc ==> cc;**
 - prin acoperirea deciziilor și condițiilor se acoperă condițiile;
- **dcc ==> dc;**
 - prin acoperirea deciziilor și condițiilor se acoperă deciziile;
- caz particular:
 - **dcc = dc și dcc = cc atunci când decizia conține doar o condiție;**
 - e.g., decizia $y < 0$ este evaluată la true sau false, similar cu evaluarea condiției $y < 0$, care este evaluată la true sau false ==> acoperirea deciziei și condiției (dcc) este similară cu acoperirea deciziei (dc), care este similară cu acoperirea condiției (cc);

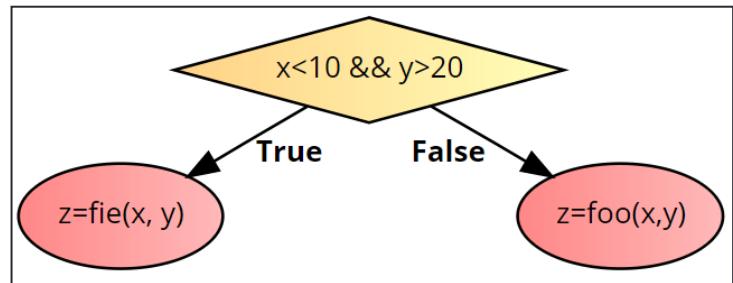
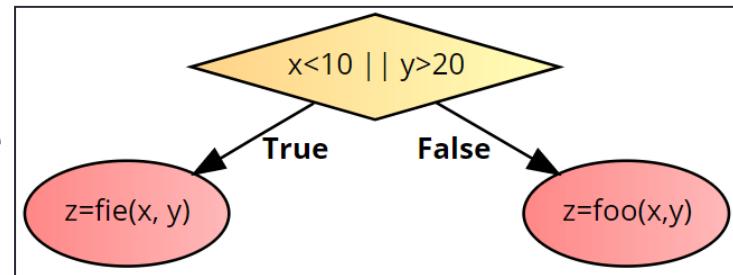
dcc se aplică doar atunci când decizia este formată din mai multe condiții.



DCC. Observații

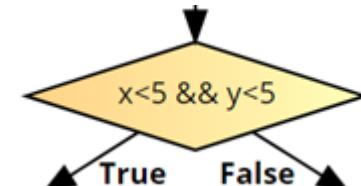
- condițiile logice care folosesc operatorii `&&` și `||` nu pot fi acoperite prin **dcc**, deoarece compilatorul realizează diverse optimizări (scurt-circuitare la evaluare);

Este necesară acoperirea condițiilor multiple.



Acoperirea condițiilor multiple. Definiție

- **acoperirea condițiilor multiple** (*engl. multiple condition coverage, mcc*):
 - proiectarea cazurilor de testare se realizează astfel încât:
 - toate combinațiile posibile ale valorilor de ieșire ale unei condiții, în fiecare decizie, să fie parcurse cel puțin o dată;
- regulă de selectare:
 - fiecare decizie care conține mai multe condiții, va combina fiecare condiție selectată care este evaluată la `true` sau `false` cu celelalte condiții în toate variantele posibile și împreună cu decizia evaluată la `true` sau `false` se vor găsi pe cel puțin un drum.

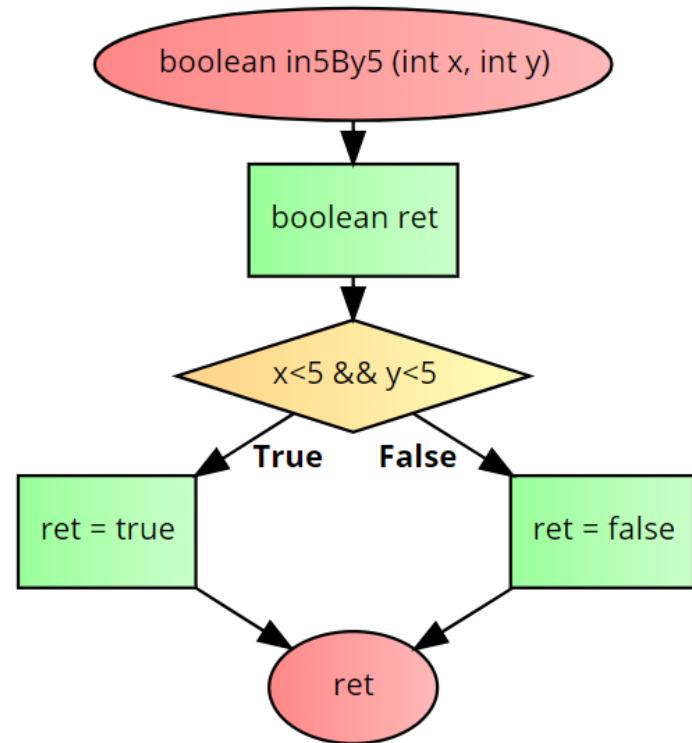


MCC. Exemplu

Fiecare *condiție* din fiecare decizie trebuie să fie executată
în toate combinațiile posibile cu toate celelalte condiții
din cadrul aceleiași decizii.

```
// returnează true dacă (x,y) este în cadranul (5,5).
boolean in5By5 (int x, int y) {
    boolean ret;
    if (x<5 && y<5)
        ret = true;
    else ret = false;
    return ret;
}
```

Care este numărul minim de cazuri de testare necesar ?

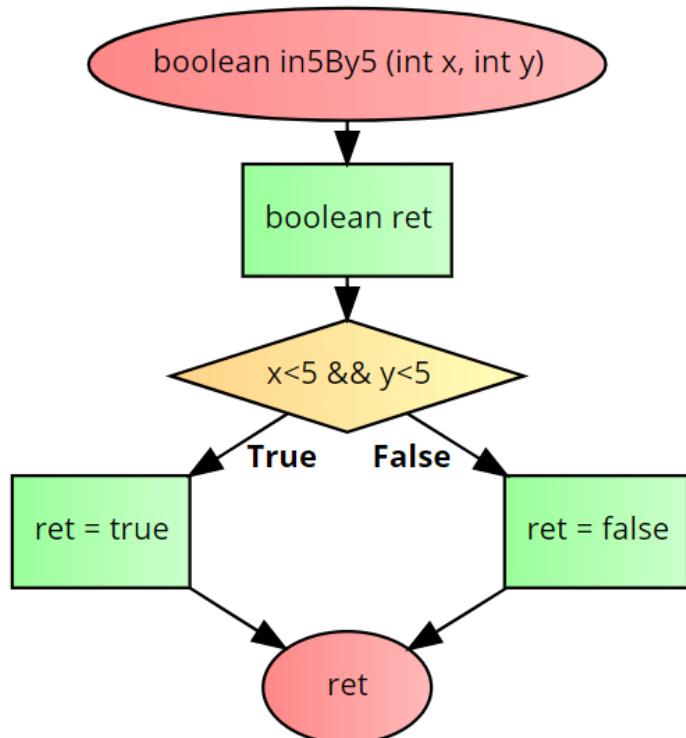


MCC. Exemplu (cont.)

```
// returnează true dacă (x,y) este în cadranul (5,5).  
boolean in5By5 (int x, int y) {  
    boolean ret;  
    if (x<5 && y<5)  
        ret = true;  
    else ret = false;  
    return ret;  
}
```

Care este numărul minim de cazuri de testare necesar ?

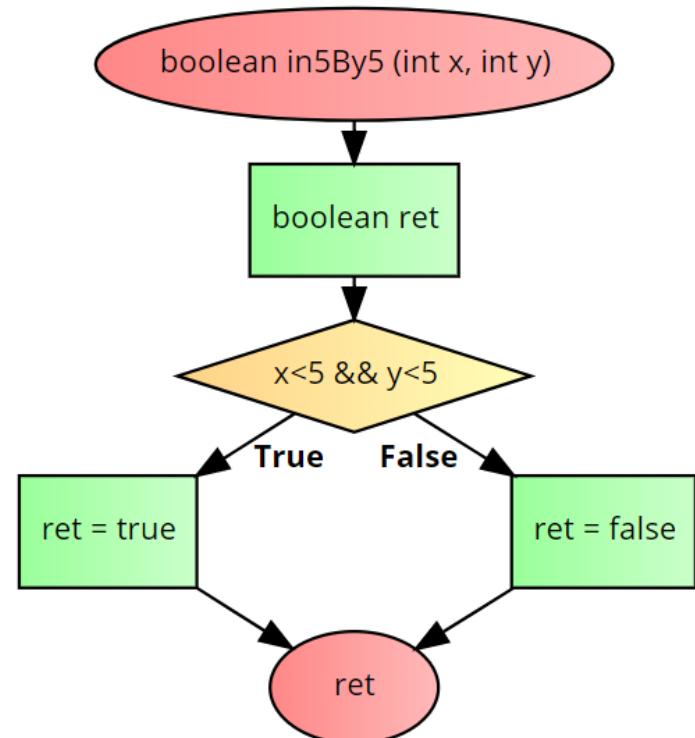
TC	x	y	Decision (x<5) && (y<5)	Expected result	Actual result
1	2	3	T && T = T	true	true
2	9	7	F && F = F	false	false
3	2	7	T && F = F	false	false
4	9	3	F && T = F	false	false



Criteriu de acoperire a condițiilor multiple îndeplinit 100%. Toate combinațiile posibile ale condițiilor au fost explorate.

MCC vs. DCC

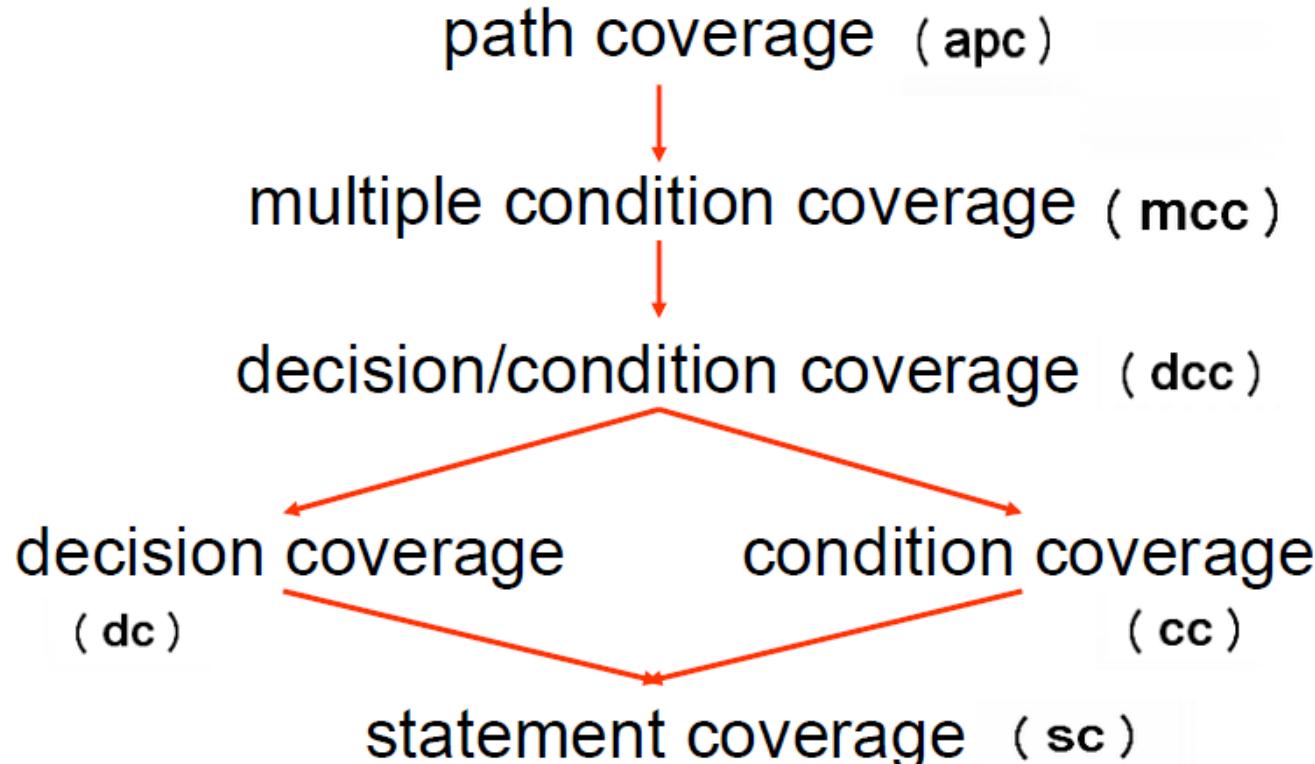
- **mcc ==> dcc;**
 - prin acoperirea multiplă a condițiilor se acoperă deciziile și condițiile;



Testare bazată pe acoperirea codului sursă. Reguli de acoperire minimală

- dacă programul are o singură condiție în fiecare vârf de decizie, atunci se aplică
 - **acoperirea deciziilor (dc), în acest caz $dc = cc$;**
- dacă programul are condiții multiple în vârfuri de decizie, atunci se aplică
 - **acoperirea condițiilor multiple (mcc).**

SC vs DC. vs. CC vs. DCC vs. MCC vs. APC

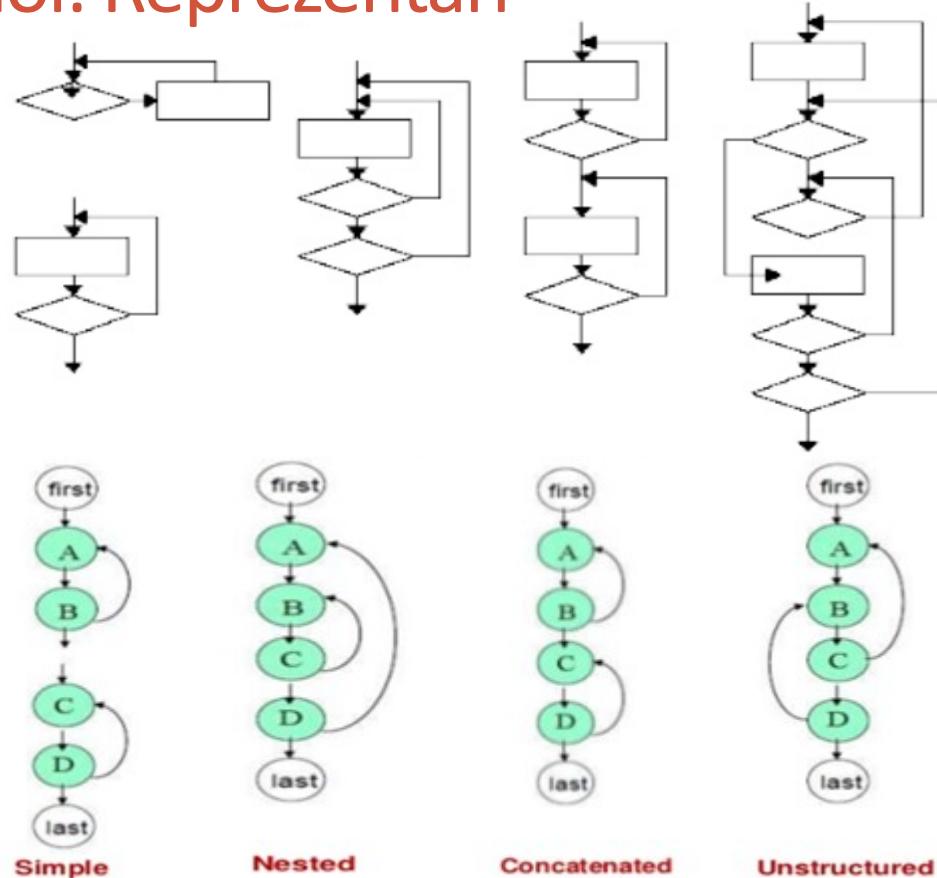


Acoperirea buclelor. Definiție

- **acoperirea buclelor (engl. loop coverage, **lc**);**
 - proiectarea cazurilor de testare astfel încât structurile repetitive să fie iterate de un număr *variabil* de ori;

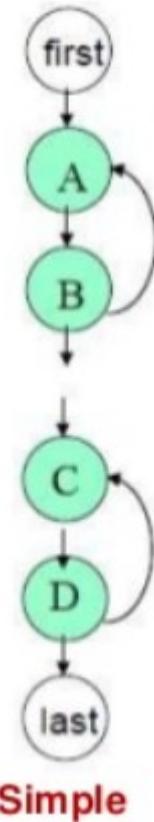
LC. Clasificarea buclelor. Reprezentări

- tipuri de bucle:
 - simple;
 - imbricate;
 - concatenate;
 - nestructurate.



LC. Bucle simple

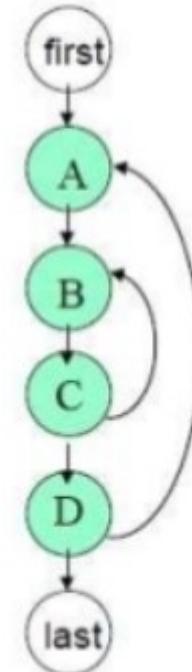
- **bucle simple** (n – numărul maxim de parcurgeri al buclei):
 - omiterea bublei (0 parcurgeri);
 - 1 parcuregere a buclei;
 - 2 parcurgeri ale buclei (evidențiază defecte de inițializare);
 - m parcurgeri ale buclei, unde $m < n$;
 - $n-1$ parcurgeri ale buclei;
 - n parcurgeri ale buclei;
 - $n + 1$ parcurgeri ale buclei.



LC. Bucle imbricate

- **bucle imbricate:**

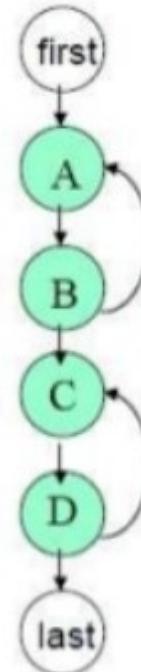
1. se pornește de la bucla cea mai interioară; toate celelalte bucle sunt setate pe valori minime;
2. se testează bucla cea mai interioară ca și buclă simplă, păstrând buclele exterioare la valoarea minimă a parametrului de iterare;
3. se progresează spre exterior, testându-se următoarea buclă și păstrând buclele exterioare la valorile minime;
4. se continuă până când toate buclele sunt testate.



Nested

LC. Bucle concatenate

- **bucle concatenate:**
 - dacă buclele sunt independente unele de altele:
 - se aplică testarea **bucelor simple**;
 - dacă buclele sunt dependente (e.g., variabila de indexare a primei bucle este valoarea inițială a celei de a două):
 - se aplică testarea **bucelor imbricate**.

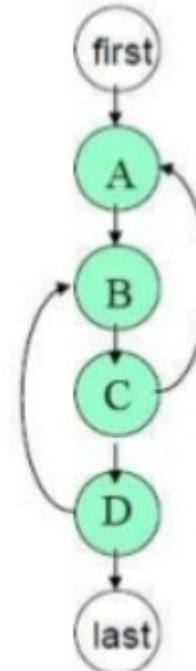


Concatenated

LC. Bucle nestructurate

- **bucle nestructurate:**

- în general, indică folosirea instrucțiunii `goto`;
- se recomanda restructurarea acestui tip de buclă pentru a reflecta elementele programării structurate.



Unstructured

TESTARE WHITE-BOX VS. TESTARE BLACK-BOX

Testare White-Box. Avantaje si dezavantaje

Testare White-Box vs. Testare Black-Box

Testare White-Box

Avantaje

- cazurile de testare sunt proiectate pe baza **structurii interne a codului sursă**, i.e., în funcție de structurile de programare folosite;
- identifică disfuncționalități în execuția anumitor secvențe de cod, e.g., unele structuri de programare nu sunt acoperite;
 - permite acoperirea cu teste a codului scris;

Dezavantaje

- **nu poate testa cerințe care nu sunt implementate**, nu poate identifica bug-urile din codul sursă care lipsește;
- **proiectarea cazurilor de testare poate începe doar după implementare**;
- testerul trebuie să cunoască limbajul de programare în care a fost elaborat codul sursă;
- **ineficientă pentru module de mari dimensiuni.**

Testarea Black-Box vs. Testare White-Box

Testare Black-Box

- Testare funcțională, testare comportamentală (*engl. behavioral testing*);
- cazurile de testare sunt proiectate pe baza specificațiilor, nu este necesar să avem acces la codul sursă;
- **suprinde ambiguitățile sau inconsistențele din specificații;**
- nu se există informații despre implementare;
- activitatea testerului este independentă de cea a programatorului; testerul poate proiecta cazurile de testare înainte de finalizarea codului sursă;
- eficientă și pentru module de mari dimensiuni.

Testare White-Box

- Testare structurală (*engl. structural testing*);
- cazurile de testare sunt proiectate pe baza structurii interne a codului sursă, i.e., în funcție de structurile de programare folosite;
- **nu poate testa cerințe care nu sunt implementate;**
- proiectarea cazurilor de testare poate începe doar după implementare;
- inefficientă pentru module de mari dimensiuni – construirea CFG și calculul CC sunt activități costisitoare.

Testarea Black-Box vs. Testarea White-Box

Întrebări:

- Ce cazuri de testare trebuie actualizate după modificarea specificațiilor și a codului sursă asociat?
- Ce cazuri de testare trebuie actualizate după modificarea codului sursă, fără modificarea specificațiilor?

PENTRU EXAMEN...

Pentru examen...

- **testare white-box:**
 - definiție, caracteristici, avantaje și dezavantaje;
 - CFG (definiție și construire), drumuri independente (definiție), CC (definiție, 3 moduri de calcul);
 - construirea CFG, determinarea drumurilor independente și calculul CC (3 moduri) pentru metode concrete;
 - criteriile de acoperire **apc, sc, dc, cc, dcc, mcc** și **lc** (definiție, compararea a două criterii, relațiile existente între criterii);
 - testare black-box vs. testare white-box.

Cursul următor...

- **Niveluri de testare**
 - Definiție. Clasificare
- **Testare unitară**
- **Testare de integrare**
- **Testare funcțională**
- **Testare de sistem**
 - Testare funcțională
 - Testare non-funcțională
- **Testare de acceptare**
- **Nivel de testare vs. Tip de testare**

Referințe bibliografice

- [Myers2004] Glenford J. Myers, *The Art of Software Testing*, John Wiley & Sons, Inc., 2004.
- [NT2005] K. Naik and P. Tripathy. *Software Testing and Quality Assurance*, Wiley Publishing, 2005.
- [Patton2005] R. Patton, *Software Testing*, Sams Publishing, 2005.
- [Collard2003] J. F. Collard, I. Burnstein. *Practical Software Testing*. Springer-Verlag New York, Inc., 2003.
- [Beizer1990] Beizer, B., *Software Testing Techniques*, Van Nostrand Reinhold., New York, 1990.

CURS 04.

NIVELURI DE TESTARE

Verificarea și validarea sistemelor soft

[21 Martie 2023]

Lector dr. Camelia Chisăliță-Crețu

Universitatea Babeș-Bolyai

Conținut

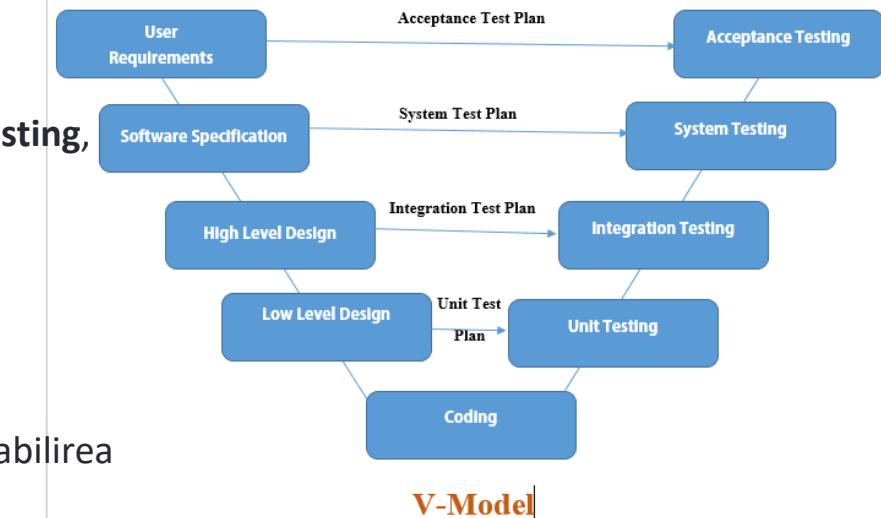
- **Niveluri de testare**
 - Definiție. Clasificare
- **Testare unitară**
 - Definiție. Motivație. Caracteristici
 - Proiectarea cazurilor de testare
 - Tipuri de bug-uri identificate
 - Reguli generale de aplicare
- **Testare de integrare**
 - Definiție. Motivație. Clasificare
 - Integrare non-incrementală. Integrare incrementală. Integrare mixtă
 - Compararea strategiilor de integrare
 - Testarea interfeței modulelor. Definiție. Clasificare
 - Tipuri de bug-uri identificate
 - Exemplu
- **Testare de sistem**
 - Definiție. Caracteristici
 - Testare funcțională
 - Testare non-funcțională
- **Testare de acceptare**
 - Definiție. Caracteristici. Etape de realizare
 - Clasificare
 - Alpha Testing. Beta Testing
 - Alpha Testing vs Beta Testing
 - Alte tipuri de testare de acceptare
 - Dificultăți de testare
- **Nivel de testare vs. Tip de testare**
 - Tip de testare. Nivel de testare. Definiție
 - Obiective de testare. Exemple
 - Retestare. Definiție
 - Testare de regresie. Definiție
 - Retestare vs Testare de regresie
- **Pentru examen...**
- **Bibliografie**

NIVELURI DE TESTARE

Definiție. Clasificare

Nivel de testare. Definiție. Clasificare

- **nivel de testare (engl. testing level):**
 - o serie de activități de testare asociate unei etape din procesul de dezvoltare a produsului soft;
- **clasificare:**
 - **testare unitară / testare de modul (engl. unit testing, module testing);**
 - etapa: implementare/codificare;
 - **testare de integrare (engl. integration testing);**
 - etapa: proiectare;
 - **testare de sistem (engl. system testing);**
 - etapa: specificarea cerințelor sistemului = stabilirea obiectivelor de realizat;
 - **testare de acceptare (engl. acceptance testing);**
 - etapa: descrierea cerințelor utilizatorului.



TESTARE UNITARĂ

Definiție. Motivație. Caracteristici

Proiectarea cazurilor de testare

Tipuri de bug-uri identificate

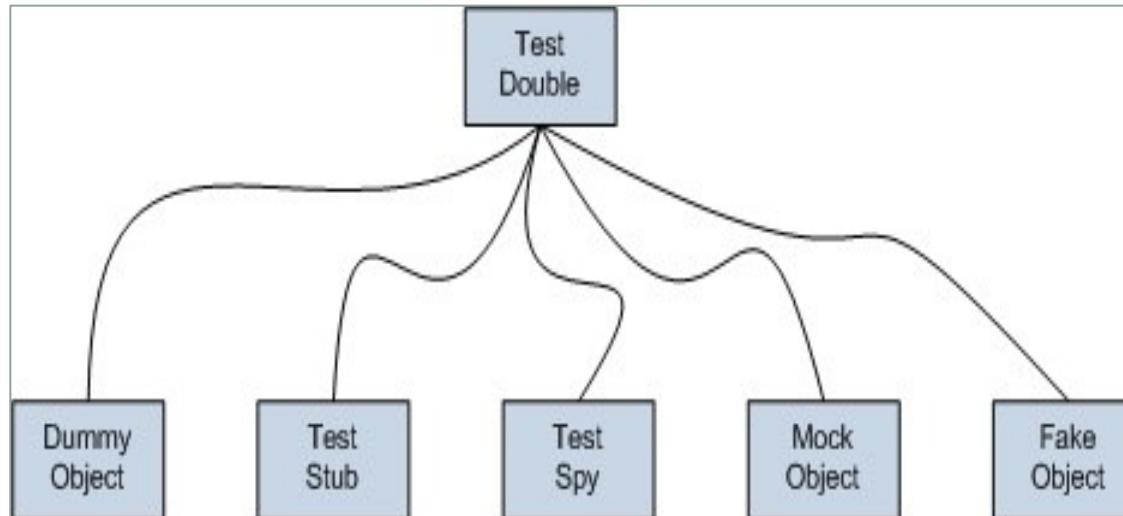
Reguli generale de aplicare

Testare unitară. Definiție. Motivație. Etape

- **testare unitară/ testare de modul** (*engl. unit testing, module testing*) :
 - testarea individuală a unor unități separate dintr-un sistem software (funcție, procedură, clasă, metodă);
- **motivație:**
 - gestionarea eficientă a modulelor sistemului – mai întâi se testează modulele;
 - proces de depanare eficient – aplicat la nivel de modul;
 - permite paralelizarea procesului de testare – testare simultană pentru mai multe module.
- **etape:**
 - contextul de testare;
 - proiectarea cazurilor de testare;
 - execuția cazurilor de testare și evaluirea rezultatelor testării.

Testare unitară. Tipuri de obiecte (1)

- la nivelul testării unitare se folosesc diferiți termeni pentru a indica obiecte, stări sau caracteristici care apar la proiectarea cazurilor de testare;
- literatura de specialitate indică abordări diferite în descrierea obiectelor folosite, denumite generic **Test Doubles**, sugerând funcționarea obiectelor utilizate în realitate [\[MeszarosFowler2006\]](#);

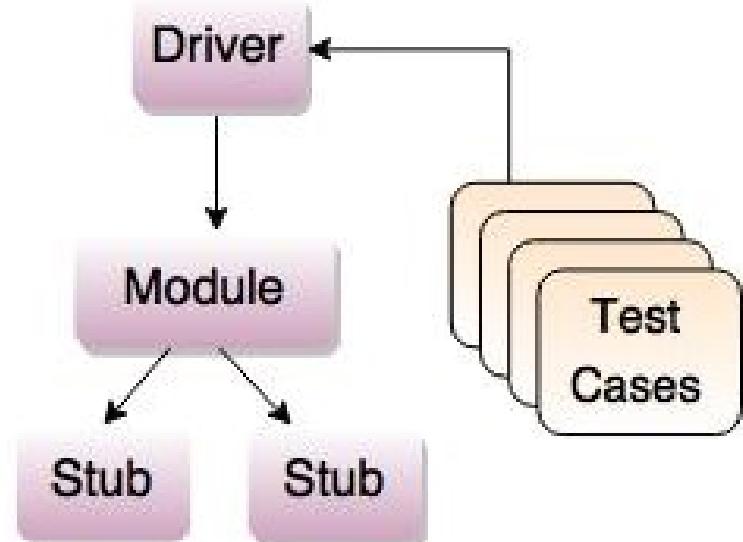


Testare unitară. Tipuri de obiecte (2)

- tipuri de obiecte utilizate de tool-uri:
 - **Dummy** – obiecte care sunt transmise ca parametri dar care **nu sunt folosite de metodele apelate**;
 - e.g., diversi parametri precizați doar pentru a respecta signatura metodei apelate;
 - **Fake** – obiecte cu **implementări funcționale**/utilizabile, dar **simpliste**, care nu sunt adecvate pentru a fi incluse în livrabilul către client;
 - e.g., o colecție de date in-memory;
 - **Stubs** – 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 unde au fost definite;
 - **Spies** – obiecte **stub** care pot păstra/reține informații referitoare la modul în care au fost folosite;
 - e.g., un serviciu pentru e-mail care reține numărul de mesaje transmise;
 - **Mocks** – obiecte pentru care s-a stabilit **un anumit comportament** (behavior expectations) și sunt utilizate pentru a observa interacțiunea cu obiectul supus testării.

Testare unitară. Context de testare

- tipuri de module:
 - **driver (engl. driver)**:
 - modul apelant al modulului testat, care furnizează datele de intrare modulului testat;
 - **stub (engl. dummy subprogram)**:
 - modul apelat în cadrul modulului testat, înlocuiește modulul apelat în contextul real;
 - arată că modulul testat apelează un modul subordonat;
 - returnează o valoare prestabilită în modulul testat care să îi permită să își continue execuția;
- pentru fiecare modul testat trebuie să existe un driver dedicat și mai multe module stub, dacă este necesar;
- modulele driver și stub sunt create suplimentar (*engl. overhead*) și nu sunt livrate împreună cu produsul final;



Testare unitară. Proiectarea cazurilor de testare

- **caracteristici:**
 - aplică tehnici de testare black-box, grey-box și white-box;
 - folosește documentele care conțin specificația modulelor;
 - informații necesare proiectării unui caz de testare pentru un modul:
 - specificația modulului;
 - codul sursă pentru modul;
- **tipuri de bug-uri identificate:**
 - **testarea black-box:**
 - nerespectarea condițiilor impuse în specificații;
 - înțelegerea greșită a specificațiilor;
 - **testarea white-box:**
 - înțelegerea greșită a precedenței operatorilor;
 - inițializare incorectă;
 - lipsa acurateței/preciziei;
 - operații aplicate eronat.

Testare unitară. Reguli generale de aplicare (1)

1. număr de pași de executat;
2. execuție: planificare și durată;
3. consecvență;
4. atomicitate;
5. responsabilitate unică;
6. izolarea testelor;
7. izolarea de mediul de execuție;
8. izolarea claselor;
9. automatizare completă;
10. *self-descriptive*;
11. fără condiții logice;
12. fără bucle;
13. fără tratarea excepțiilor;
14. utilizarea instrucțiunilor assert;
15. utilizarea de mesaje sugestive;
16. fără testare în codul sursă livrat;
17. separarea pe module și niveluri ale arhitecturii;
18. gruparea testelor în funcție de tipul de testare.

Testare unitară. Reguli generale de aplicare (2)

- **număr de pași de executat:**
 - 3-5 pași:
 1. **set up;**
 2. date de intrare;
 3. apelarea metodei testate;
 4. verificarea rezultatului (**assert**);
 5. **tear down;**

Testare unitară. Reguli generale de aplicare (3)

- **timp de execuție:**
 - frecvența de execuție:
 - testare după implementare (*engl. test after development*): de câteva ori pe zi;
 - dezvoltare dirijată de testare (*engl. test driven development*): de câteva ori pe oră;
 - execuție după salvare, IDE (*engl. IDE runs tests after save*): la fiecare câteva minute;
 - mod de execuție (singular sau suită de teste):
 - timp de execuție pentru 10 teste = timp de execuție 1 test x 10;
 - **un test care se execută greu, încetinește întreaga suită de teste;**
 - valori medii:
 - un test < 200 milisecunde;
 - o suită cu număr de redus de teste < 10 secunde;
 - o suită cu număr consistent de teste < 10 minute.

Testare unitară. Reguli generale de aplicare (4)

- **consecvență:**
 - execuția repetată a aceluiași test ar trebui să returneze în mod repetat același rezultat, dacă nu au avut loc modificări asupra codului sursă;
- cod sursă problematic:
 - Date current Date = new Date();
 - Int value = random.nextInt(100);
- soluții:
 - utilizarea obiectelor dummy, mock, stub, fake;
 - injectarea dependențelor.

Testare unitară. Reguli generale de aplicare (5)

- **atomicitate:**
 - rezultate posibile ale execuției unui test:
 - **passed**;
 - **failed**;
 - nu există teste care „au trecut” doar parțial;
 - **dacă un punct de execuție este failed ==> întregul test este failed.**

Testare unitară. Reguli generale de aplicare (6)

- **acțiune/ responsabilitate unică** (*engl. single responsibility*):
 - un caz de testare investighează un singur scenariu de execuție;
- se testează comportamentul metodei:
 - **o metodă, mai multe utilizări (comportamente)**
 - ==> mai multe teste și cel puțin un test pentru fiecare comportament;
 - mai multe instrucțiuni assert în același test – doar dacă verifică același comportament;
 - **o utilizare (comportament) descrisă prin folosirea mai multor metode**
 - ==> un singur test;
 - E.g.: o metodă care apelează metode private/ protected/ publice, simple, e.g., getters, setters, constructori simpli.

Testare unitară. Reguli generale de aplicare (7)

- acțiune/ responsabilitate unică (*engl. single responsibility*):

- o metodă, mai multe utilizări (comportamente) ==> mai multe teste;

```
testMethod() {
    ...
    assertTrue(behaviour1);
    assertTrue(behaviour2);
    assertTrue(behaviour3);
}
```

```
testMethodCheckBehaviour1 () {
    ...
    assertTrue(behaviour1);
}
testMethodCheckBehaviour2 () {
    ...
    assertTrue(behaviour2);
}
testMethodCheckBehaviour3 () {
    ...
    assertTrue(behaviour3);
}
```

Testare unitară. Reguli generale de aplicare (8)

- **acțiune/ responsabilitate unică** (*engl. single responsibility*):
 - o utilizare (**comportament**) descrisă prin folosirea mai multor metode ==> un singur test;
 - comportament 1 = condition1 + condition2 + condition3 ;
 - comportament 2 = condition4 + condition5;

```
testMethodCheckBehaviours () {  
    ...  
    assertTrue (condition1);  
    assertTrue (condition2);  
    assertTrue (condition3);  
    ...  
    assertTrue (condition4);  
    assertTrue (condition5);  
}
```

```
testMethodCheckBehaviour1 () {  
    ...  
    assertTrue (condition1);  
    assertTrue (condition2);  
    assertTrue (condition3);  
}  
testMethodCheckBehaviour2 () {  
    ...  
    assertTrue (condition4);  
    assertTrue (condition5);  
}
```

Testare unitară. Reguli generale de aplicare (9)

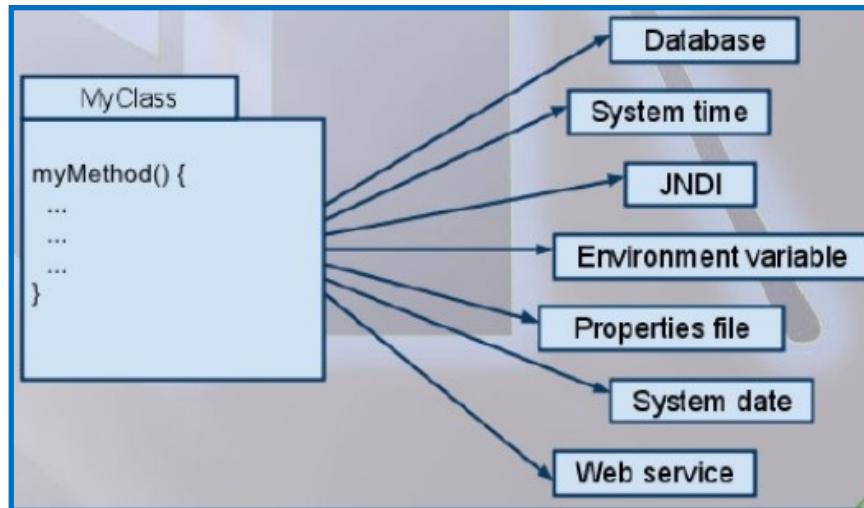
- **izolarea testelor (unele de altele):**
 - testele trebuie să fie independente unele de altele.
 - **la execuții diferite (la momente de timp diferite, în ordine diferită) ale aceluiași test trebuie să se obțină aceleași rezultate.**
 - **nu** se partajează starea/ contextul de execuție între teste;
 - variabile folosite la testare, e.g., JUnit – *variabilele partajate sau nu între teste*: @Before, @BeforeClass.

Testare unitară. Reguli generale de aplicare (10)

- **izolarea testelor de mediul de execuție (context):**
 - testele trebuie izolate de influențele mediului de execuție;
 - E.g.,
 - baze de date;
 - apelarea serviciilor web;
 - Java Naming and Directory Interface (JNDI);
 - variabile de mediu definite local;
 - fișiere de proprietăți;
 - configurările de dată și oră ale sistemului.

Testare unitară. Reguli generale de aplicare (11)

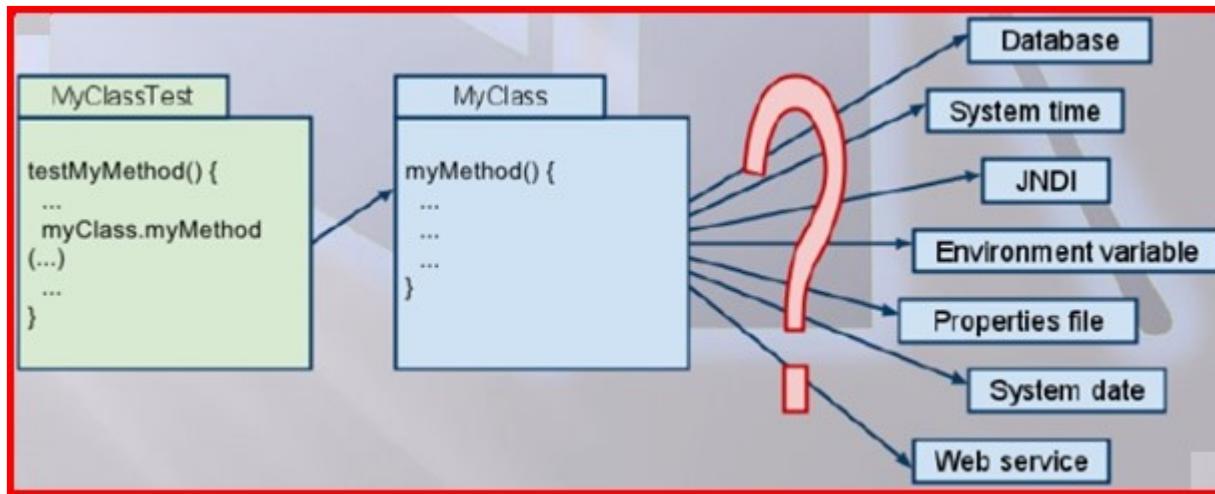
- izolarea testelor de mediul de execuție (context):
 - codul sursă livrabil (engl. **production code**) folosește *mediul de execuție*;



Testare unitară. Reguli generale de aplicare (12)

- izolarea testelor de mediul de execuție (context):

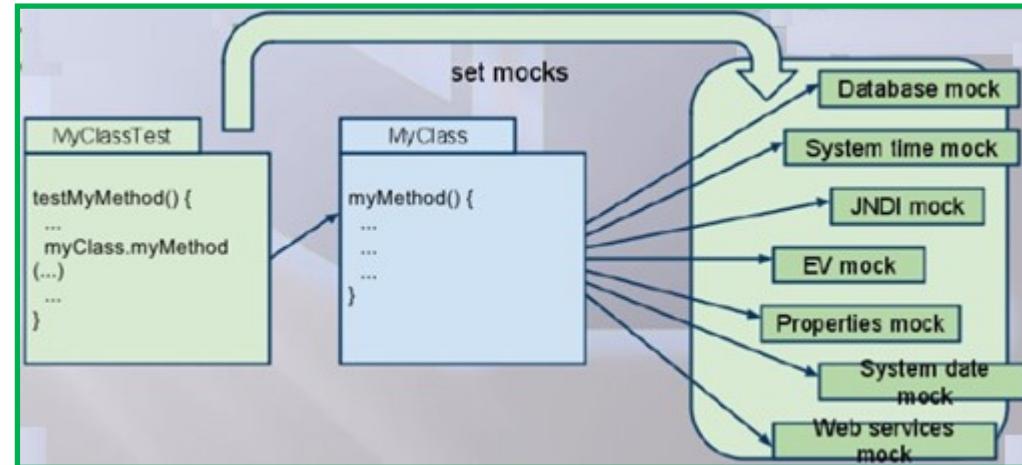
- la testare se folosește un **mediu de testare**, care nu este întotdeauna identic cu **mediul de execuție** de la dezvoltarea codului sursă sau de la client;



Testare unitară. Reguli generale de aplicare (13)

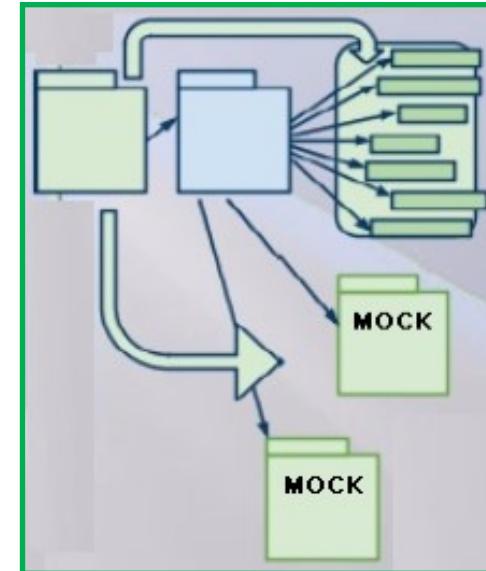
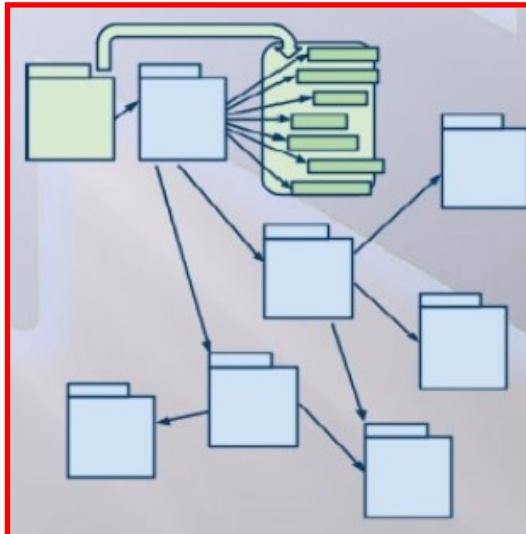
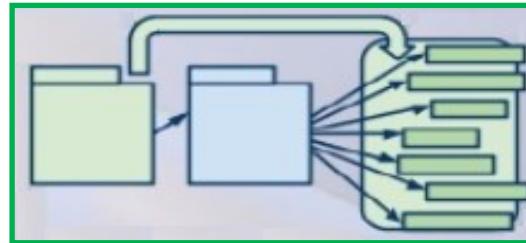
- izolarea testelor de mediul de execuție (context):
- **soluție:** se utilizează obiecte mock care indică un anumit mediu de utilizare;

- Avantaje:
 - rapiditate;
 - ușor de dezvoltat;
 - reutilizabilitate;
- E.g.: biblioteci Java:
 - EasyMock;
 - Jmock;
 - Mockito.



Testare unitară. Reguli generale de aplicare (14)

- izolarea claselor:
- clase fără dependențe
- clase cu dependențe
 - soluție: dependențe dummy, mock, stub, fake;



Testare unitară. Reguli generale de aplicare (15)

- **izolarea claselor:**
 - dificil de realizat dacă avem un cod sursă greu de testat;
 - se recomandă:
 - folosirea **metodelor Factory** sau **injectarea dependentelor**, în locul apelului constructorilor în cadrul metodelor;
 - **folosirea interfețelor.**

Testare unitară. Reguli generale de aplicare (16)

- **automatizare completă:**
 - fără pași execuții manual pe durata testării.
- se automatizează:
 - execuția testelor;
 - colectarea rezultatelor testării;
 - stabilirea rezultatului testării (**passed**/ **failed**);
 - trasmisarea rezultatelor testării prin: e-mail, IDE integration, etc;

Testare unitară. Reguli generale de aplicare (17)

- **self-descriptive:**
 - la nivelul testării unitare un test reprezintă:
 - documentație la nivel de dezvoltare;
 - metodă de specificare care reflectă versiunea actualizată a cerințelor;
- **caracteristici:**
 - un test trebuie să fie ușor de citit și înțeles;
 - denumirile variabilelor, metodelor și claselor trebuie să fie *self-descriptive*;
 - **nu trebuie să conțină condiții logice sau iterații;**
 - numele cazurilor de testare trebuie să fie sugestive pentru a indica condiția de succes sau eșec:
 - `public void canMakeReservation();`
 - `public void cannotAddNewBook();`

Testare unitară. Reguli generale de aplicare (18)

- **fără instrucțiuni logice:**
 - un caz de testare **nu** ar trebui să conțină instrucțiunea if sau switch;
- **nu** există incertitudini legate de:
 - *datele de intrare* ==> toate datele de intrare sunt cunoscute;
 - *comportamentul așteptat* ==> comportamentul metodei este predictibil;
 - *datele de ieșire* ==> rezultatele așteptate ar trebui să fie bine definite.

Testare unitară. Reguli generale de aplicare (19)

- **fără instrucțiuni logice:**

- mai multe condiții trebuie să se regăsească în cazuri de testare distințe, nu în același test, fiind tratate ca și ramificații ale instrucțiunilor alternative, i.e., if, switch;

```
testMethodBeforeOrAfter() {
    ...
    if (before) {
        assertTrue(behaviour1);
    } else if (after) {
        assertTrue(behaviour2);
    } else { //now
        assertTrue(behaviour3);
    }
}
```

```
testMethodBefore() {
    before = true;
    assertTrue(behaviour1);
}
testMethodAfter() {
    after = true;
    assertTrue(behaviour2);
}
testMethodNow() {
    before = false;
    after = false;
    assertTrue(behaviour3);
}
```

Testare unitară. Reguli generale de aplicare (20)

- **fără instrucțiuni iterative** (i.e., while, do while, for):
 - scenarii tipice pentru iterații:
 - **câteva sute de iterații:**
 - dacă sunt necesare câteva sute de iterații, atunci testul este destul de complicat și **este necesar să fie simplificat**;
 - **câteva iterații:**
 - se recomandă refactorizarea codului care trebuie să se repete într-o metodă, care să fie apelată ulterior explicit de câte ori este necesar;
 - **număr de iterații necunoscut:**
 - numărul de iterații este greu de evitat, indicând faptul că e posibil ca **testul să fie incorrect** și se recomandă specificarea mai clară a datelor de intrare;

```
testThrowingMyException() {  
    try {  
        myMethod(param);  
        fail("MyException expected");  
    } catch(MyException ex) {  
        //OK  
    }  
}
```

Testare unitară. Reguli generale de aplicare (21)

- **instructiunea assert:**
- se recomandă:
 - utilizarea instructiunilor assert disponibile în platforma de testare;
 - crearea propriilor asertii pentru verificarea conditiilor complexe care se repetă în diferite teste;
 - reutilizarea propriilor metode asertive;
 - utilizarea asertiunilor în cadrul buclelor.

Testare unitară. Reguli generale de aplicare (22)

- **mesaje sugestive în instrucțiunea assert:**
 - prin citirea mesajului din aserțiune se poate recunoaște ușor problema care a determinat eşuarea testului;
- **avantaje:**
 - permit îmbunătățirea documentației codului sursă;
 - oferă informații asupra problemei dacă testul a eşuat (failed).

Testare unitară. Reguli generale de aplicare (23)

- **codul sursă nu include teste:**
- se recomandă:
 - separarea testelor de codul sursă livrat;
 - clasele să **nu** definească metode și/sau attribute care sunt folosite doar la testare.

Testare unitară. Reguli generale de aplicare (24)

- **gruparea testelor în funcție de tipul de testare:**
- după modulul testat:
 - organizarea testelor în pachete corespunzătoare modulelor testate;
 - utilizarea unei abordări ierarhizate;
 - scăderea timpului de execuție pentru suitele de teste prin împărțirea acestora în suite de dimensiuni mai mici, i.e., suitele de mici dimensiuni pot fi executate mai des/frecvent;
- după tip:
 - scopul / obiectivele testării;
 - frecvența de execuție;
 - momentul execuției suitei;
 - acțiunea în caz de eșec.

TESTARE DE INTEGRARE

Definiție. Motivație. Clasificare

Integrare non-incrementală. Integrare incrementală. Integrare mixtă

Compararea strategiilor de integrare

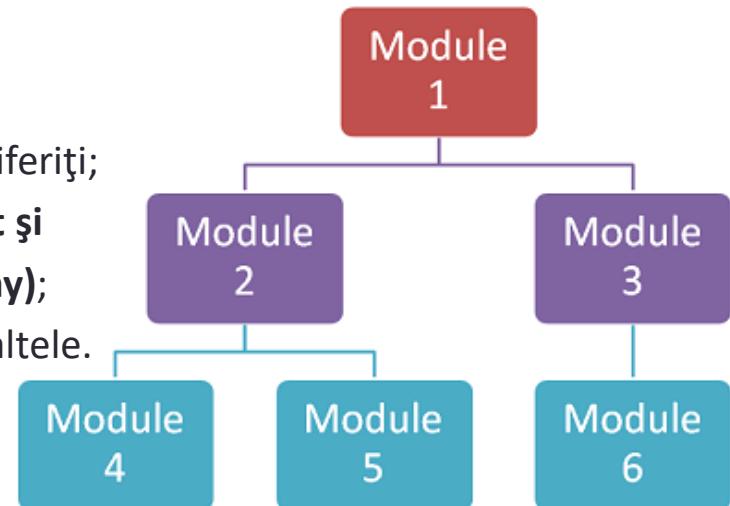
Testarea interfeței modulelor. Definiție. Clasificare

Tipuri de bug-uri identificate

Exemplu

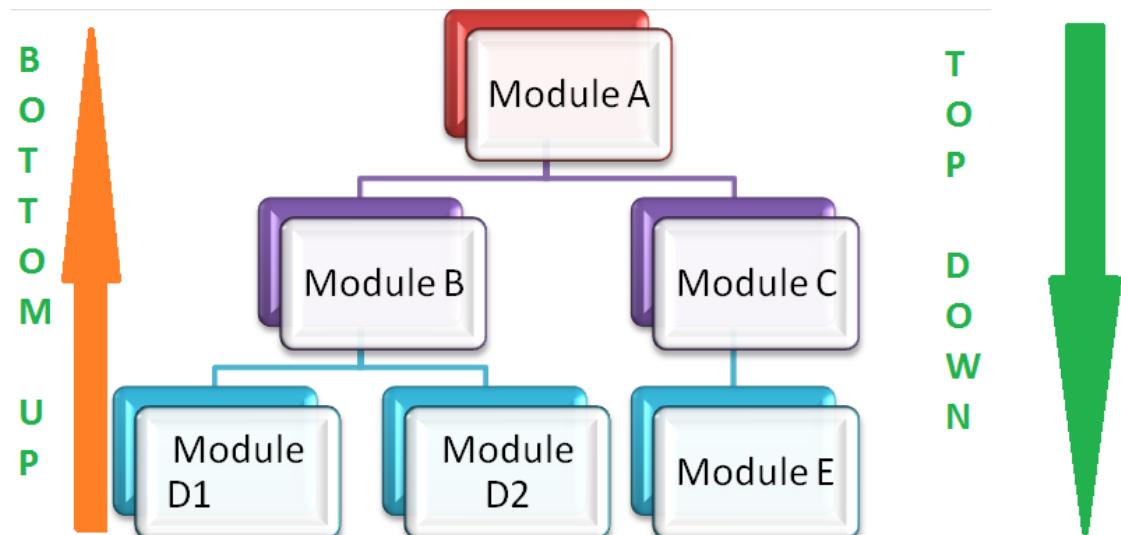
Testare de integrare. Definiție. Motivație

- **testare de integrare** (*engl. integration testing*):
 - nivel de testare în care modulele individuale se combină și se testează ca un grup;
 - permite construirea structurii programului pe măsură ce este testat pentru a **identifica erorile la nivelul interfeței dintre module**;
- **motivație**:
 - module diferite sunt implementate de programatori diferiți;
 - testarea unitară se desfășoară într-un **mediu controlat și izolat**, folosind entități **driver și stub (mock, fake, dummy)**;
 - unele module pot genera mai multe defecțiuni decât altele.



Testare de integrare. Clasificare

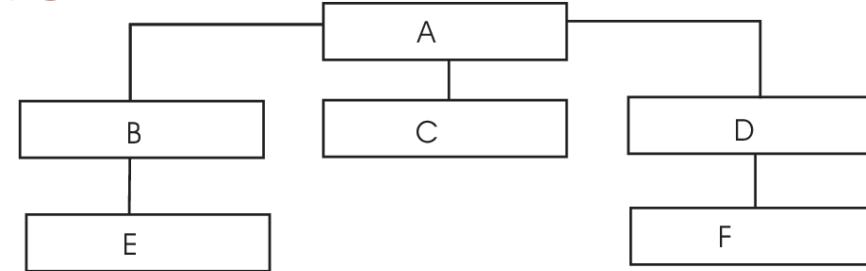
- abordări de integrare/ clasificare:
 - non-incrementală
 - **big-bang**;
 - incrementale
 - **top-down**;
 - **bottom-up**;
 - mixtă
 - **sandwich**.



Integrarea Big-bang. Descriere

- Procesul de integrare Big-bang:

1. testare unitară pentru fiecare modul, folosind:
 - un modul driver;
 - câteva module stub;



2. se combină simultan modulele pentru construirea funcționalității programului;

- **dezavantaje:**

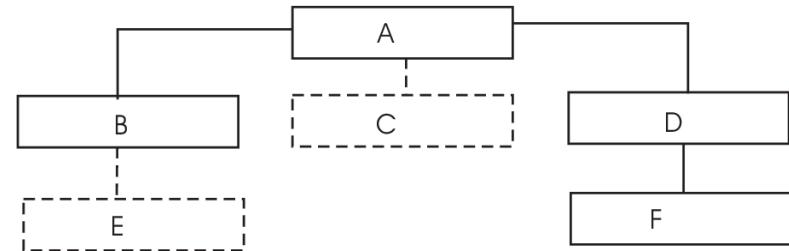
- necesită un volum de muncă ridicat;
- depanare dificilă – este dificil de izolați modulul care a determinat defecțiunea;
- dacă programul este complex – este dificil de urmărit modul în care s-au executat funcționalitățile.

- **avantaje:**

- dezvoltare și testare unitară paralelă.

Integrare incrementală Top-down. Descriere

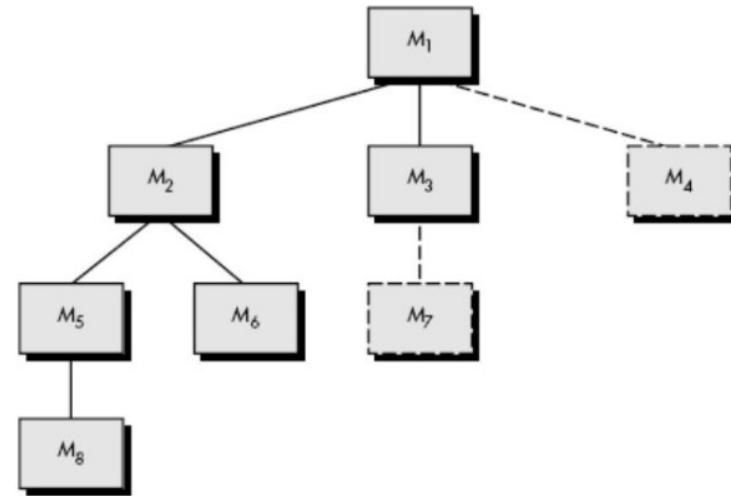
- integrare incrementală Top-down (*engl. Top-down incremental integration*):
 - permite construirea și testarea programului adăugând module noi pe parcurs;
 - integrarea modulelor se face de sus în jos, de la modulul principal, modulele subordonate sunt încorporate succesiv;
 - defectele sunt ușor de izolat și corectat;
 - testarea este aplicată sistematic.
- tipuri de integrare Top-down:
 - **în adâncime** (*engl. depth-first integration*);
 - **pe niveluri** (*engl. breadth-first integration*);



- **integrarea depth-first**
 - integrează toate componentele de pe o ramificație majoră a arhitecturii aplicației;
 - ordinea de integrare a ramificațiilor depinde de caracteristicile aplicației;
- **integrarea breadth-first**
 - integrează toate componentele care se află pe nivelul direct subordonat, i.e., pe orizontală;

Integrare incrementală Top-down. Algoritm

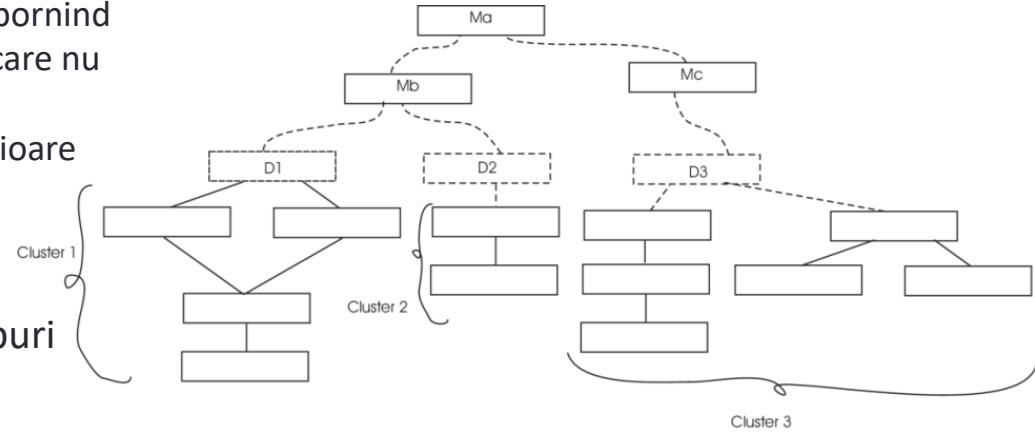
- după testarea unitară a modulului principal folosind **stub-uri** pentru modulele imediat subordonate, procesul de **integrare Top-down** constă în:
 - în funcție de tipul de integrare ales (**depth-first/ breadth-first**), se înlocuiește un stub cu un modul real;
 - se testează cu modulul subordonat concret care a fost integrat;
 - pentru integrarea unui nou modul se repetă pașii 1 și 2.
- driver** = modulul principal; nu se folosesc alte drivere;
- stub** = înlocuiește un modul direct subordonat;



- **E.g.,**
 - **depth-first:** M1, M2, M5, M8, M6, M3, M7, M4;
 - **breadth-first:** M1, M2, M3, M4, M5, M6, M7, M8.

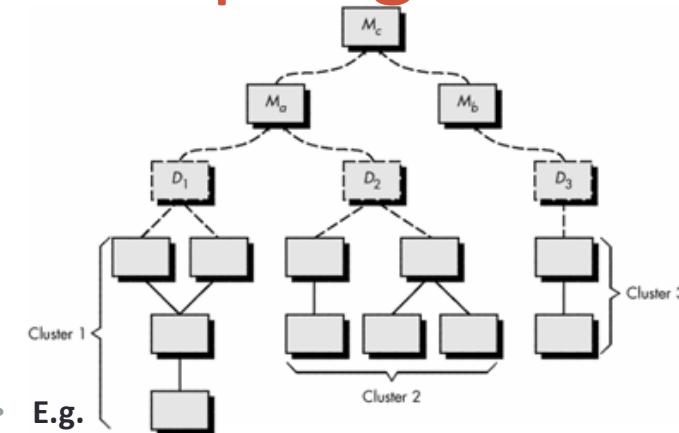
Integrare incrementală Bottom-up. Descriere

- **integrare incrementală Bottom-up (engl. Bottom-up incremental integration):**
 - permite construirea și testarea programului pornind de la modulele atomice, i.e., componente care nu au module dependente;
 - defectele modulelor aflate pe nivelurile inferioare sunt ușor de izolat și corectat.
- modulele terminale se pot organiza în grupuri (engl. clusters);
- **driver** = se folosesc doar pentru modulele terminale (care nu au module subordonate) sau grupurile de module;
- **stub** = nu se folosesc;



Integrare incrementală Bottom-up. Algoritm

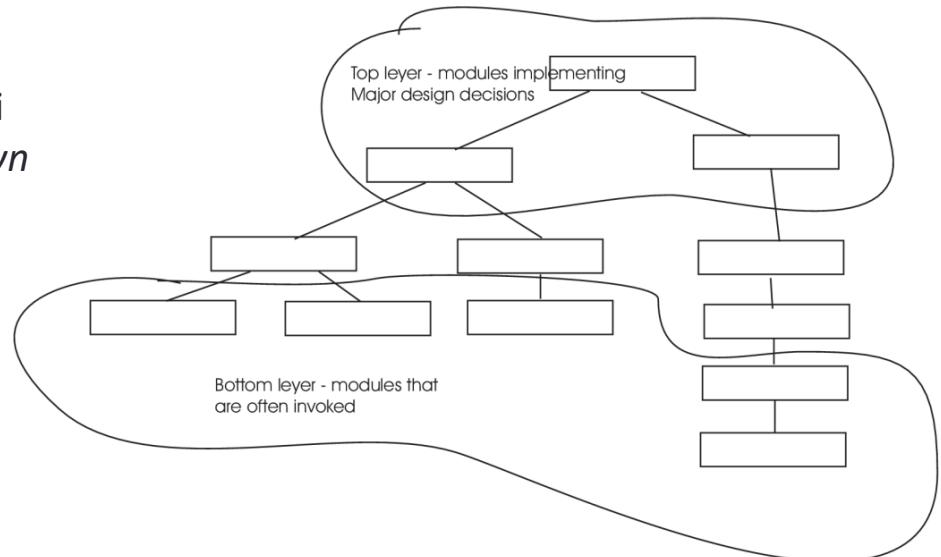
- Procesul de integrare Bottom-up:
 - pentru toate modulele terminale sau clusteri de module se descriu drivere și se testează;
 - *module terminale*:
 - se înlocuiește driver-ul cu modulul de pe nivelul imediat superior și se testează, continuând integrarea modulelor spre partea superioară a structurii programului;
 - *clusteri de module*:
 - se înlătură driver-ul, iar clusterul de module se combină cu alte module continuând integrarea spre partea superioară a structurii programului.



- E.g.
 - componentele se combină în clusterii 1 , 2 și 3;
 - clusterii se testează folosind driver-ele D1, D2 și D3;
 - se înlătură D1 și D2, iar clusterul 1 și clusterul 2 sunt integrați în Ma;
 - similar, D3 este înlăturat iar clusterul 3 este integrat cu modulul Mb.
 - Ma și Mb se vor integra în Mc.

Integrare Sandwich. Descriere

- **integrare sandwich (engl. sandwich integration):**
 - permite construirea și testarea programului combinând abordările de integrare *top-down* și *bottom-up*;
- Procesul de integrare Sandwich:
 1. **modulele terminale** (bottom-layer):
 - integrare *bottom-up*;
 2. **modulul principal** (top-layer):
 - integrare *top-down*;
 3. **celelalte module** (middle-layer):
 - integrare *big-bang*.



Testare de integrare. Compararea strategiilor de integrare

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

Testarea interfeței modulelor. Definiție. Clasificare

- **testarea interfeței modulelor** (*engl. interface integration testing*):
 - se realizează la integrarea mai multor module pentru obținerea unor sisteme de dimensiuni mai mari;
 - relevantă în dezvoltarea orientată pe obiecte, comportamentul acestora fiind descris prin intermediul interfețelor;
 - **obiective**: identificarea defectelor care pot apărea la utilizarea unei interfețe a unui alt modul sau false presupuneri legate de interfața unui modul;
- **comunicarea între module** poate fi bazată pe:
 - **parametri**: la transmiterea datelor de la o metodă la alta;
 - **memorie partajată**: o zonă de memoria este partajată între module;
 - **comportament**: un sub-sistem încapsulează un set de metode care sunt apelate de alte sub-sisteme;
 - **mesaje**: un sub-sistem are nevoie de serviciile altor sub-sisteme;

Testarea interfeței modulelor. Tipuri de bug-uri identificate

- **tipuri de bug-uri** [[NT2005](#), pagina 161]:
 - utilizarea greșită a interfeței (*engl. interface misuse*):
 - un modul apelează un alt modul eronat, e.g., parametrii sunt transmiși în ordinea greșită;
 - înțelegerea greșită a semnificației interfeței (*engl. interface misunderstanding*):
 - un modul face presupuneri greșite referitare la comportamentul unui alt modul apelat, e.g., se interpretează că mai mulți parametri de același tip au altă semnificație;
 - greșeli de sincronizare (*engl. timing errors*):
 - modulul apelat și modulul apelant folosesc unități de timp diferite, e.g., minute, secunde, milisecunde, iar informația obținută își pierde acuratețea.

Testare de integrare. Reguli generale de aplicare

- se studiază arhitectura aplicației pentru **identificarea modulelor critice**; aceste module se testează cu **prioritate**;
- testerul este familiarizat cu modulele care se integrează (arhitectura modulelor);
- se aplică o **strategie de integrare** care să permită atingerea **obiectivelor** testării;
- fiecare modul este testat înainte de a execuția testelor de integrare, i.e., unit testing;
- se utilizează **documentația care descrie interacțiunile dintre fiecare două module** (interfețele acestora și modul de colaborare).

TESTARE DE SISTEM

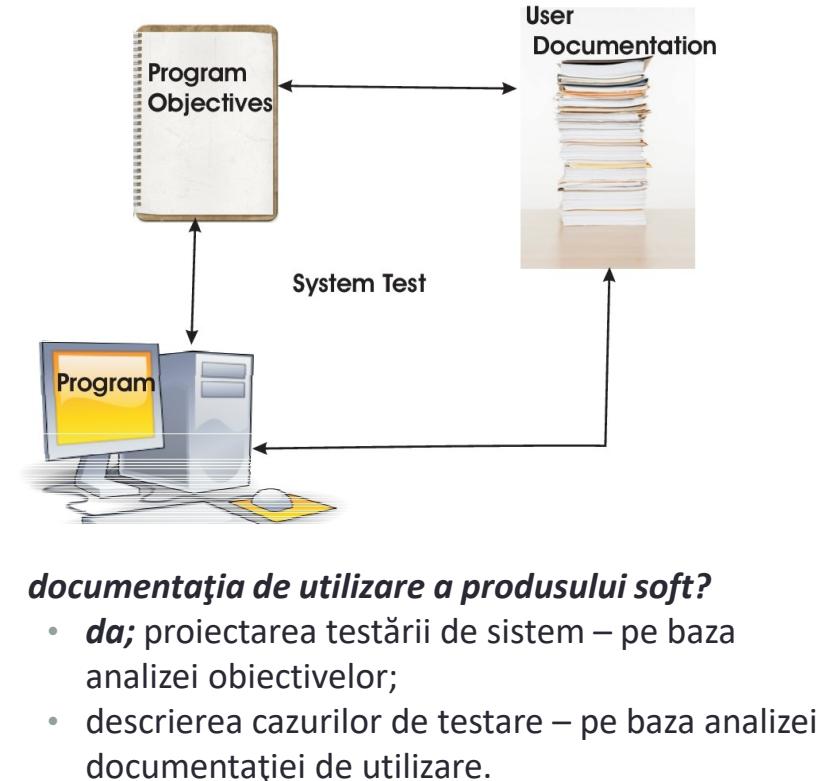
Definiție. Caracteristici

Testare funcțională

Testare non-funcțională

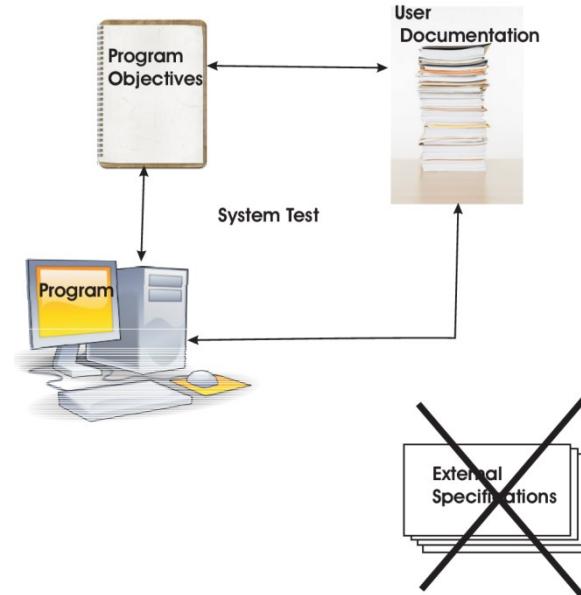
Testare de sistem. Definiție. Caracteristici

- **testare de sistem (engl. system testing):**
 - verifică dacă produsul dezvoltat respectă obiectivele inițiale;
- elaborarea cazurilor de testare se bazează pe:
 - **documentul cu specificațiile sistemului?**
 - **nu**, deoarece pot apărea erori în procesul de transpunere a obiectivelor în specificații externe;
 - **documentele cu obiectivele stabilite?**
 - **nu**, deoarece nu conțin descrierea exactă a interfețelor externe ale programului;
 - obiectivele nu oferă informații cu privire la funcționalitatea sistemului (interfețe ale



Testare de sistem. Caracteristici

- nu există metodologie pentru proiectarea cazurilor de testare în testarea de sistem;
 - se proiectează două categorii de teste asociate:
 - cerințele funcționale
==> **testare funcțională**;
 - cerințele non-funcționale
==> **testare non-funcțională**
(engl. **non-functional testing**);
- **la acest nivel de testare, proiectarea cazurilor de testare este determinată de obiectivele de testare și de strategia aleasă, evidențiind creativitatea și experiența.**



Testare de sistem. Testare funcțională a sistemului

- **testare funcțională a sistemului** (*engl. function/functional testing, use case testing*):
 - testarea cerințelor descrise în specificațiile sistemului;
 - proces care identifică neconcordanțele existente între comportamentul programului și specificația acestuia, din punctul de vedere al utilizatorului;
- elaborarea cazurilor de testare se bazează pe **criteriul black-box**;
- folosește specificația sistemului.

Testare funcțională a sistemului. Proiectarea cazurilor de testare

- **caz de utilizare vs. caz de testare:**

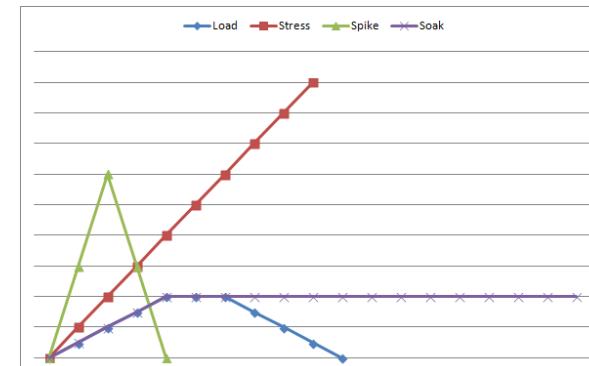
- cazurile de utilizare descriu fluxul de execuție al unui proces în cadrul sistemului, folosind **scenarii** de utilizare frecventă;
- **cazurile de testare sunt derivate din cazurile de utilizare** și pot indica prezența unui bug la utilizarea concretă a sistemului;
- fiecare caz de testare se bazează de obicei pe un scenariu și acoperă diferite ramificații de execuție (indicate prin cazuri speciale de date de intrare sau condiții speciale);
- pentru un caz de testare specifică:
 - **toate precondițiile** care este necesar să fie satisfăcute pentru execuția cazului de utilizare cu succes;
 - **postcondițiile** impuse asupra rezultatelor obținute;
 - **descrierea stării finale a sistemului** după ce testarea cazului de utilizare s-a realizat cu succes;

Testare de sistem. Testare non-funcțională. Definiție

- **testare non-funcțională (engl. non-functional testing):**
 - verifică modul în care sistemul îndeplinește cerințele non-funcționale;
 - stabilește dacă sistemul este pregătit pentru a fi efectiv utilizat;
- în [[Myers2004](#), Cap 6] sunt descrise 15 tipuri de testare (non-funcțională) de sistem pentru care se scriu cazuri de testare:
 - Volume testing;
 - Stress testing;
 - Usability testing;
 - Security testing;
 - Performance testing;
 - Storage testing;
 - Configuration testing;
 - Compatibility/Conversion testing;
 - Instability testing;
 - Reliability testing;
 - Recovery testing;
 - Serviceability testing;
 - Documentation testing;
 - Procedure testing;
 - Facility testing.

Testare de sistem. Tipuri de testare non-funcțională (1)

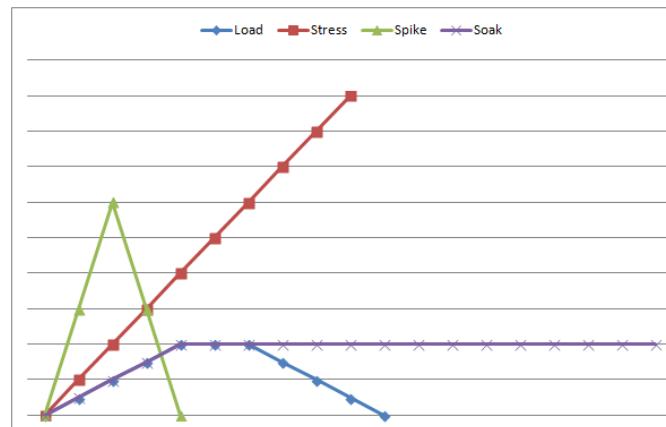
- **performance testing:** - cât de bine face sistemul ceea ce trebuie să facă
 - fiecare sistem are cerințe referitoare la performanța utilizării;
 - e.g., execuția unei funcționalități să fie realizată într-un anumit interval de timp, iar resursele să nu fie infinite;
 - bug-urile legate de performanță indică deficiențe la nivel de proiectare care determină degradarea performanței sistemului soft în timpul utilizării;
 - **obiectiv:** identificarea blocajelor determinante de reducerea performanței componentelor și a sistemelor, i.e., **bottlenecks**;
 - tipuri:
 - **Capacity Testing, Load Testing;**
 - **Volume Testing, Stress Testing;**
 - **Soak Testing, Spike Testing.**



Testare de sistem. Tipuri de testare non-funcțională (2)

- **capacity testing:**

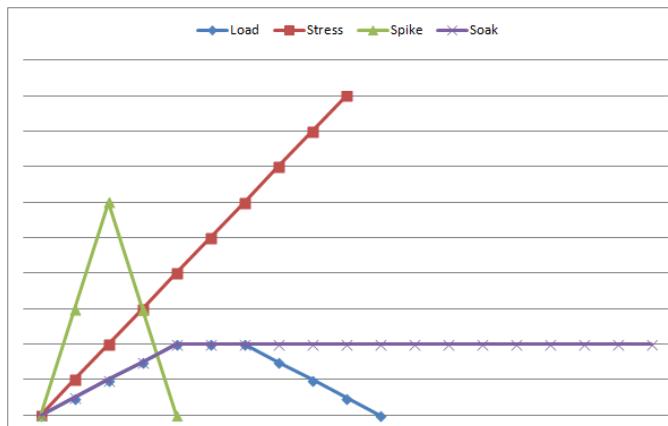
- permite prevenirea eventualelor probleme determinate de creșterea numărului de utilizatori sau a volumului de date;
- **obiectiv:** obținerea de informații referitoare la **nivelul maxim de utilizatori și/sau date care nu afectează performanța**:
 - E.g.: câte fișiere pot fi folosite fără a afecta performanța.



Testare de sistem. Tipuri de testare non-funcțională (3)

- **load testing:**

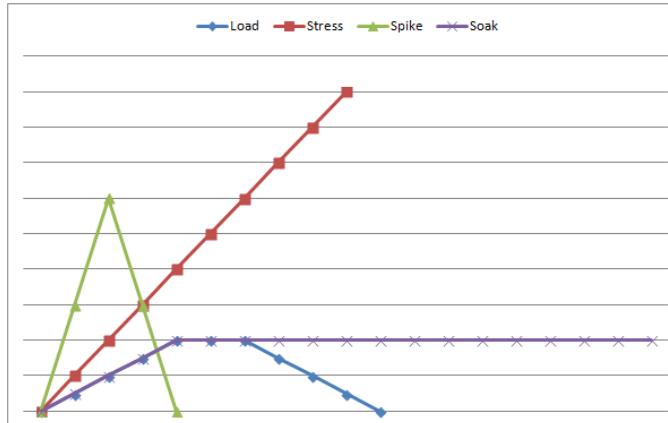
- evaluează capacitatea sistemului de opera cu volume de date normale sau în condiții de solicitare (vârf);
- **obiectiv:** obținerea de informații referitoare la comportamentul sistemului în **anumite condiții de utilizare, i.e., normale și de solicitare:**
 - E.g.: creșterea numărului de fișiere folosite.



Testare de sistem. Tipuri de testare non-funcțională (4)

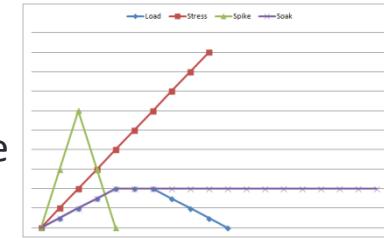
- **volume testing:**

- evaluează comportamentul sistemului atunci când se gestionează volume mari de date;
- **obiectiv:** obținerea de informații legate de **funcționarea aplicației cu un volum de date ridicat**:
 - E.g.: creșterea dimensiunii fișierelor folosite.



Testare de sistem. Tipuri de testare non-funcțională (5)

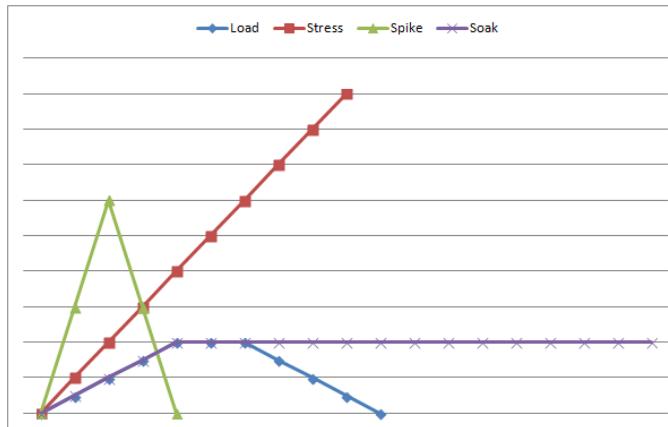
- **stress testing:** - pune presiune asupra limitelor sistemului
 - investighează dacă comportamentul softului se degradează în condiții extreme de utilizare și când nu are acces la resursele necesare;
 - este de dorit ca degradarea softului cauzată de creșterea numărului de cereri să se realizeze acceptabil, fără să ducă la un eşec imediat în timpul testării; sistemul nu trebuie să eşueze catrastofal;
 - relevant pentru sistemele distribuite care pot indica o degradare severă atunci când rețeaua devine încărcată;
 - **obiectiv:** obținerea de informații despre **modul în care sistemul funcționează în condiții extreme, dincolo de limitele normale:**
 - E.g.: creșterea numărului de fișiere folosite până la failure și chiar mai mult;
 - verifică dacă are loc o pierdere inacceptabilă de date și/sau imposibilitatea de a utiliza anumite servicii.



Testare de sistem. Tipuri de testare non-funcțională (6)

- **soak testing:**

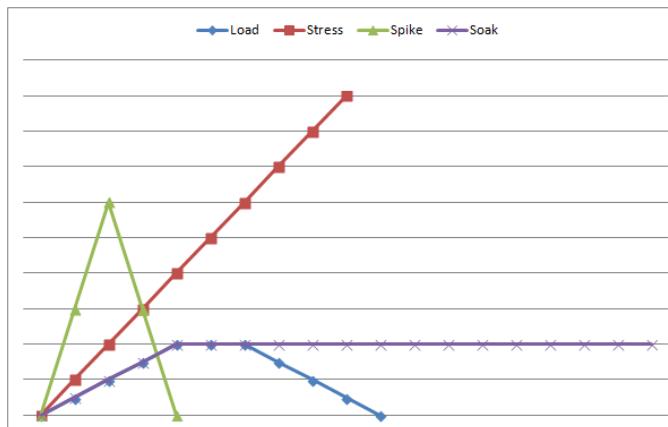
- verifică dacă sistemul face față unui volum mare de date pentru o perioadă mai mare de timp;
- **obiectiv:** obținerea de informații legate de **posibilitatea de a utiliza volum de date și care sunt consecințele dacă se depășește volumul pentru care a fost proiectat:**
 - E.g.: verifică dacă creșterea dimensiunii fișierelor folosite este suportată.



Testare de sistem. Tipuri de testare non-funcțională (7)

- **spike testing:**

- evaluează comportamentul sistemului la schimbări brusă și extreme a condițiilor de lucru, i.e., puține sarcini, multe sarcini;
- **obiectiv:** obținerea de informații legate de **punctele slabe ale sistemului determinate de modificarea rapidă a condițiilor de lucru**:
 - E.g.: creșterea bruscă și diminuarea rapidă a numărului de fișiere cu care se lucrează.



Testare de sistem. Tipuri de testare non-funcțională (8)

- **reliability testing:**
 - investighează capacitatea un soft de a funcționa conform aşteptărilor utilizatorului, chiar și atunci când eșuează;
 - **determină cât timp și cât de eficient poate funcționa sistemul fără eroare;**
- **security testing:**
 - calitatea, securitatea și gradul de încredere în aplicație (*engl. reliability*) sunt caracteristici depedindente;
 - defectele produsului soft pot fi exploataate pentru a identifica breșe de securitate;
 - **presupune simularea unor atacuri la nivel de securitate, având scopul de a identifica vulnerabilitățile softului.**

TESTARE DE ACCEPTARE

Definiție. Caracteristici. Etape de realizare. Clasificare

Alpha Testing. Beta Testing

Alpha Testing vs Beta Testing

Alte tipuri de testare de acceptare

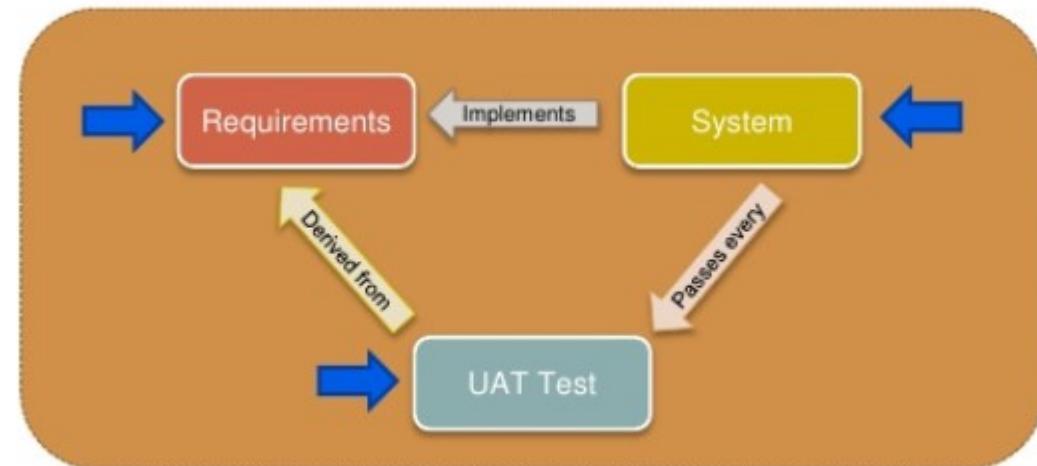
Dificultăți de testare

Testare de acceptare. Definiție. Clasificare

- **testare de acceptare (engl. user acceptance testing, UAT):**
 - procesul de testare prin care se verifică dacă programul îndeplinește cerințele inițiale și nevoile curente ale utilizatorului final;
 - nu este responsabilitatea dezvoltatorului produsului;
 - tester = clientul/ beneficiarul;
 - realizată efectiv de către client, care aplică tehnici de testare black-box.
-
- **tipuri de testare de acceptare:**
 - **alpha testing, beta testing;**
 - contract acceptance testing;
 - regulation acceptance testing;
 - operational acceptance testing.

Testare de acceptare. Etape de realizare

- **etape de realizare:**
 - definirea criteriilor prin care produsul soft este considerat funcțional;
 - creare unei suite de cazuri de testare pentru UAT;
 - rularea testelor UAT;
 - evaluarea și raportarea rezultatelor.



Testare de acceptare. Alfa Testing. Beta Testing

	Alpha testing	Beta testing
Când?	Înainte de livrare	Înainte de livrare
Cine?	clientul sau alte persoane desemnate, care testează produsul pe o platformă instalată la dezvoltator;	clientul testează produsul soft pe o platformă unde dezvoltatorul nu poate interveni, într-un mediu instalat la beneficiarul produsului soft;
Unde?	produsul soft se folosește într-un mediu controlat, unde programatorul poate interveni imediat pentru a elmina bugurile;	produsul soft este folosit în mediul pentru care a fost dezvoltat;
Cum?	se folosesc tehnici de testare black-box ;	se folosesc tehnici de testare black-box ;
Raportare bug-uri?	defectele sunt inventariate și eliminate/rezolvate imediat ;	clientul inventariază dificultățile de utilizare ale produsului soft și le raportează programatorului pentru a fi rezolvate.

Testare de acceptare. Alpha Testing vs Beta Testing

	Alpha testing	Beta testing
Tipuri de testare ?	reliability testing și security testing NU se realizează în profunzime;	realibility testing, security testing și robustness testing se realizează în detaliu;
Eliminare bug-uri ?	deficiențele majore pot fi rezolvate imediat de programator;	deficiențele raportate sunt investigate ulterior, iar îmbunătățirile și corecturile se regăsesc în versiunile ulterioare ale produsului soft;
Relevanță?	permite simularea unui mediu real de utilizare înainte de a fi trimis la client (beta testing);	furnizează un feedback autentic (real) din partea utilizatorului final.

Testare de acceptare. Tipuri de testare de acceptare

- **contract acceptance testing:**
 - produsul soft este testat din perspectiva îndeplinirii unor criterii și specificații care sunt precizate într-un **contract de colaborare** între dezvoltator și client;
 - criteriile și specificațiile de acceptanță sunt stabilite la semnarea contractului, **înainte** de dezvoltarea softului;
- **regulation acceptance testing, i.e., compliance acceptance testing:**
 - verifică dacă produsul soft dezvoltat respectă regulamentele și legile în vigoare referitoare de utilizarea și funcționarea unui produs soft specific;
- **operational acceptance testing, i.e., operational readiness testing, production acceptance testing:**
 - verifică dacă există fluxurile informaționale necesare pentru utilizarea produsului soft de către client (e.g., planuri de backup, training pentru utilizatori, diferite procese de întreținere și verificări de securitate).

Testare de acceptare. Dificultăți de testare

- **dificultăți care apar la nivelul testării de acceptare:**
 - cerințe care nu sunt descrise satisfăcător (clar, complet, corect);
 - planificarea târzie (defectuoasă) a activităților de testare;
 - testarea nu este realizată riguros, nu este o activitate planificată și monitorizată;
 - identificarea tardivă a defectelor, cu dificultăți de eliminare a acestora.

TIP DE TESTARE VS NIVEL DE TESTARE

Tip de testare. Nivel de testare. Definiție

Obiective de testare. Exemple

Retestare. Definiție

Testare de regresie. Definiție

Retestare vs Testare de regresie

Tip de testare. Nivel de testare. Definiție

- **nivel de testare** (*engl. testing level*):
 - o serie de activități de testare asociate unei etape din procesul de dezvoltare al produsului soft;
 - **Ce testezi?**
- **tip de testare** (*engl. testing type*):
 - mijlocul prin care un **obiectiv** al testării, stabilit anterior pentru un **nivel de testare**, poate fi realizat;
 - **Cum testezi?**

Tip de testare. Exemple

- exemple de tipuri de testare:
 - **testarea unei metode:**
 - se poate realiza prin aplicarea unor criterii de testare (black-box, white-box), la nivelul *testării unitare* sau *de integrare*;
 - **testarea unei caracteristici non-funcționale:**
 - se realizează prin aplicarea unui anumit tip de testare, e.g., testare de performanță, testare de utilizabilitate, cu scopul de a evalua o caracteristică a calității produsului soft, la nivelul *testării de sistem*;
 - **testarea după eliminarea unui bug:**
 - se realizează prin aplicarea **re-testării** (*engl. re-testing, confirmation testing*) după depanare, la **orice nivel de testare**;
 - **testarea legată de eliminarea unui bug:**
 - se realizează prin **testarea de regresie** (*engl. regression testing*), pentru a verifica dacă eliminarea unui bug nu are efecte secundare asupra softului, la **orice nivel de testare**.

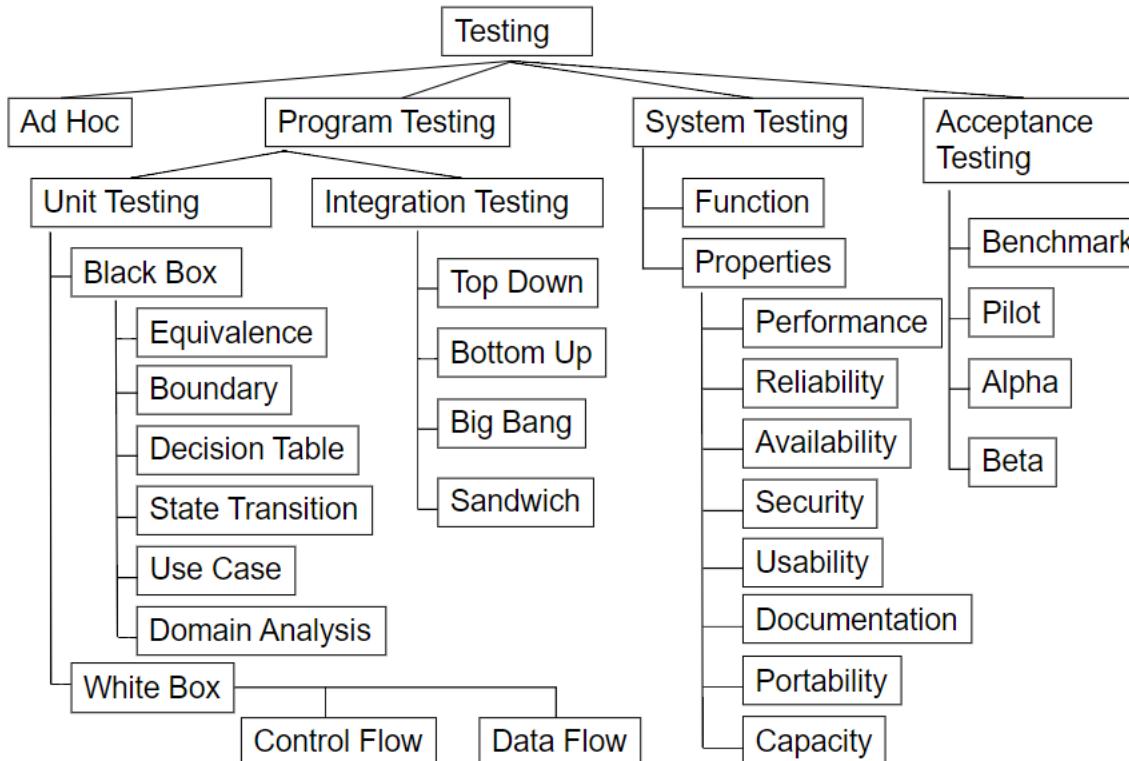
Re-testare. Definiție

- **re-testare (engl. re-testing, confirmation testing):**
 - re-execuția testelor care au pus în evidență anterior un bug ce se presupune că a fost eliminat;
- **scop:** confirmarea că defectul a fost eliminat;
- cazurile de testare re-executate sunt identice cu cele rulate anterior;

Testare de regresie. Definiție

- **testare de regresie (engl. regression testing):**
 - re-execuția unor teste care au fost rulate anterior cu succes;
- **scop:** identificarea efectelor secundare (bug-uri) care pot apărea în urma modificării unor module;
- cazurile de testare se pot organiza în teste de regresie, care permit testarea:
 - tuturor funcționalităților sistemului;
 - funcționalităților cu probabilitate ridicată de a fi afectate de modificări;
 - comportamentului componentelor sistemului care au fost modificate.
- **testare de regresie ≠ re-testare;**

Niveluri de testare. Tipuri de testare



PENTRU EXAMEN...

Pentru examen...

- **Niveluri de testare. Definiții și caracteristici:**
 - testare unitară;
 - testare de integrare;
 - 4 strategii (big-bang, top-down, bottom-up, sandwich), descriere, comparare;
 - testare de sistem;
 - testare funcțională;
 - 5 tipuri de testare non-funcțională (volume, stress, load, usability, security) [\[Mye04\]](#).
 - testare de acceptare;
 - alpha testing, beta testing.
- **Tip de testare vs Nivel de testare. Definiții și caracteristici:**
 - re-testare;
 - testare de regresie.

Cursul următor...



- Curs 05:
 - Tematică
 - Test Automation Demo: Selenium WebDriver + Serenity BDD
 - Performance Testing
 - Companie IT invitată: **Evozon**
 - Data: **Marți, 28 Martie 2023;**
 - Orele: **08:00-10:00;**
 - Desfășurare: **Sala 2/I (N. Iorga), Clădirea Centrală a UBB.**

Seminar 04. Create and Solve a Puzzle (1)

- **Create and Solve a Puzzle**
 - **termen:** **26 aprilie 2023;**
 - **echipe:** max. 3 studenți/echipă, i.e., echipe de forma (A, B, C) sau (A, B);
 - echipele participante pot rezolva unul sau ambele task-uri de mai jos; pentru fiecare task rezolvat corect se acordă **câte 2 puncte** de activitate la seminarul 4 fiecărui membru al echipei;
 - **Task 01. Create a Puzzle:** cu min. 20 concepte studiate la VVSS;
 - se va accesa pagina <https://www.puzzle-maker.com/> și se va alege tipul de joc **crossword**;
 - se va crea un puzzle folosind min. 20 termeni studiați în cadrul disciplinei VVSS;
 - se va genera puzzle-ul pe fundal alb și se va posta pe channel-ul **#Games**;
 - echipa se va înscrie în fișierul **Seminar 04** pentru a putea primi punctajul pentru crearea puzzle-ului;
 - pentru acest task se poate folosi orice alt tool care permite obținerea puzzle-ului în forma cerută.

Seminar 04. Create and Solve a Puzzle (2)

- **Create and solve a puzzle**
 - **Task 02. Solve a Puzzle;**
 - o echipă poate rezolva un puzzle propus de o altă echipă;
 - o echipă poate rezolva un puzzle chiar dacă nu a propus anterior un puzzle;
 - după rezolvarea puzzle-ului (prin editarea fișierului sau listare, apoi rezolvare, apoi poză/scan) de către echipa (X, Y, Z), acesta se trimite **doar** echipei (A, B, C) care a propus puzzle-ul, pentru a fi evaluat;
 - fiecare răspuns corect primește 0.10 puncte (0.10×20 întrebări = 2 puncte);
 - echipa (A, B, C) evaluatează soluția primită și completează în fișierul [Seminar 04](#) componența echipei care a oferit soluția și punctajul corespunzător (e.g., 2 puncte, 1.8. puncte);
 - după ce au fost evaluate soluții oferite de max. 6 echipe participante, echipa care a propus puzzle-ul va posta soluția la puzzle în thread-ul postării initiale pe channel-ul **#Games**, indicând prin aceasta faptul că nu mai poate primi spre evaluare alte soluții.

Referințe bibliografice

- **[Myers2004]** Glenford J. Myers, *The Art of Software Testing*, John Wiley & Sons, Inc., 2004.
- **[NT2005]** K. Naik and P. Tripathy. *Software Testing and Quality Assurance*, Wiley Publishing, 2005.
- **[MeszarosFowler2006]** Meszaros, G., Fowler, M., *Test Doubles*,
<https://martinfowler.com/bliki/TestDouble.html>

Intro to automation testing



• About me

- My name: Cosmin Ciocan
- Test Lead with over a decade of experience in software testing



<https://www.linkedin.com/in/cosmin-ciocan-84523089/>

- Today's agenda

- Intro to Automation Testing

- Demo

- Intro to Performance Testing

- Demo



AUTOMATION TESTING

In software **testing**, **test automation** is the use of special software (separate from the software being **tested**) to control the execution of **tests** and the comparison of actual outcomes with predicted outcomes.



- AUTOMATION TESTING - WHY

- Automation testing is cheaper

- Automation testing is faster

- Automation testing is reliable

- Automation testing reduces human and technical risks

- Quick feedback during development

- Reduces the need for manual testing - reduces testing resources



- AUTOMATION TESTING - WHEN

- Projects with long lifespan



- AUTOMATION TESTING - WHEN

- Projects with long lifespan

- Start early



- AUTOMATION TESTING - WHEN

- Projects with long lifespan

- Start early

- Experienced and technical testers



- AUTOMATION TESTING - WHEN

- Projects with long lifespan

- Start early

- Experienced and technical testers

- Frequent regression testing

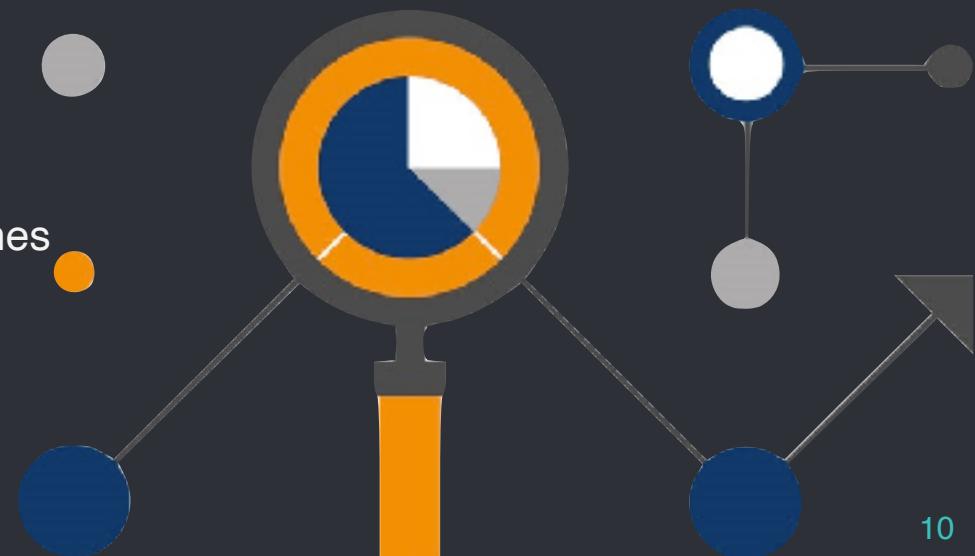


• AUTOMATION TESTING - WHAT and HOW

Automation testing is a tool which a tester can use to help him.

Depending on the application, the strategy would be to start with:

- Core application features
- Smoke tests
- Regression tests
- Tests you need to run multiple times



- AUTOMATION TESTING - WHEN NOT

- Projects with short lifespan

- Inexperienced testers

- Unstable design

- Insufficient time/resources



ONE DOES NOT SIMPLY



AUTOMATE ALL OF THE APPLICATION

- AUTOMATION TESTING - WHAT NOT

- Don't automate everything

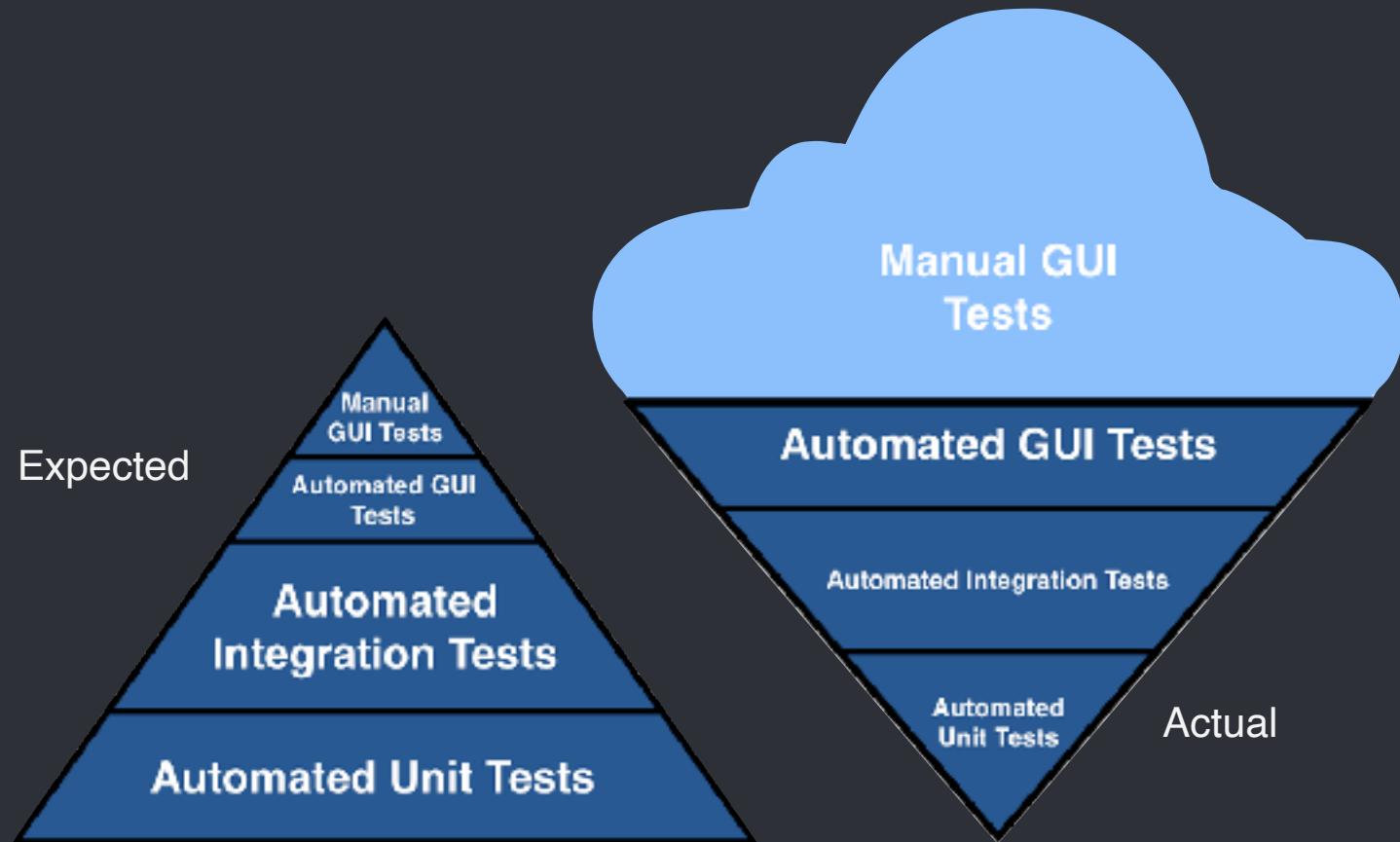
- Don't automate tests that are easier to do manually

- Don't automate areas that will change

- Don't automate design



AUTOMATION TESTING - THE PYRAMID OF TESTING



• AUTOMATION TESTING - HOW

Automation is more than a record and play tool



JUnit



- Automation needs a framework
- Good automation needs code review and refactoring
- Automation needs maintenance
- Automation is more than test execution

- AUTOMATION TESTING - WHAT TO USE



- AUTOMATION TESTING - WHAT TO USE

In order to start writing tests, we need:

- Programming language
- Selenium WebDriver
- Browser





- AUTOMATION TESTING - Selenium WebDriver

WebDriver is a tool for automating web application testing, and in particular to verify that they work as expected.

Creating a WebDriver object - opening a browser:



```
System.setProperty("webdriver.chrome.driver", "C:/drivers/chromedriver.exe");
WebDriver driver = new ChromeDriver();
```



```
System.setProperty("webdriver.firefox.driver", "C:/drivers/geckodriver.exe");
WebDriver driver = new FirefoxDriver();
```



```
System.setProperty("webdriver.ie.driver", "C:/drivers/
internetexplorerdriver.exe");
WebDriver driver = new IeDriver();
```



- AUTOMATION TESTING - WebDriver methods

- Manipulating the **WebDriver** object:



```
WebDriver driver = new ChromeDriver();
```

```
driver.get(String url)      - Navigates to URL
```

```
driver.findElement(By Selector) - Returns element
```

```
driver.findElements(By Selector) - Returns a list of elements
```

```
driver.getTitle() - Returns the title of the page as String
```

```
driver.manage().window().maximize() - Maximizes the window
```

```
driver.close() - Closes browser window
```



- AUTOMATION TESTING - WebDriver methods

- Manipulating the **WebElement** object:



```
WebElement element = driver.findElement(By Selector)
```

```
element.click()
```

- Clicks on the element

```
element.sendKeys(String text)
```

- Sends text to element

```
element.clear()
```

- Clears text from element

```
element.getText()
```

- Returns the text from the element

```
element.isDisplayed()
```

- Returns TRUE or FALSE if element is displayed



DEMO

- **Demo project on GitHub**

- <https://github.com/cosminevo/TrelloDemoApp>

- Read the [README.md](#) file
- In order to run the tests you need to have an account, and generate the KEY and TOKEN for that user



Thank you!

- # Performance Testing

Gabi Kis – UBB - March 2023

- Here are the main topics:



- Intro
 - What is performance testing
 - Performance testing types
- Web page performance analysis
 - Good practices
 - Tools & Demo
- Load testing
 - Requirements gathering
 - What and how to script
 - Configure for execution
 - Load testing demo
 - Interpret and report results
- Q&A anytime

WHAT is it?

“

“Performance testing is a testing practice performed to determine how a system performs in terms of responsiveness and stability under a particular workload.”
– Wikipedia.



● NFR - Performance Testing

- **Functional vs. Non Functional** Requirements
 - **Functional** requirements describe **what** the system should do
 - **Non-functional** requirements describe **how** the system should behave
- **Why?**
 - Demonstrate that system meets performance criteria
 - Find which parts perform badly – find bottlenecks
 - Improve overall performance of the system
- **What?**
 - Performance specifications
 - Concurrency/throughput
 - Server response time
 - Other ...

● Performance Testing Types

Load

- under expected specific load
- find bottlenecks

Stress

- above expected load
- upper limits of capacity
- determine the breaking point

Spike

- short period of time
- extreme load
- recovery of the system

Volume

- large volume/amount of data
- check performance with large data

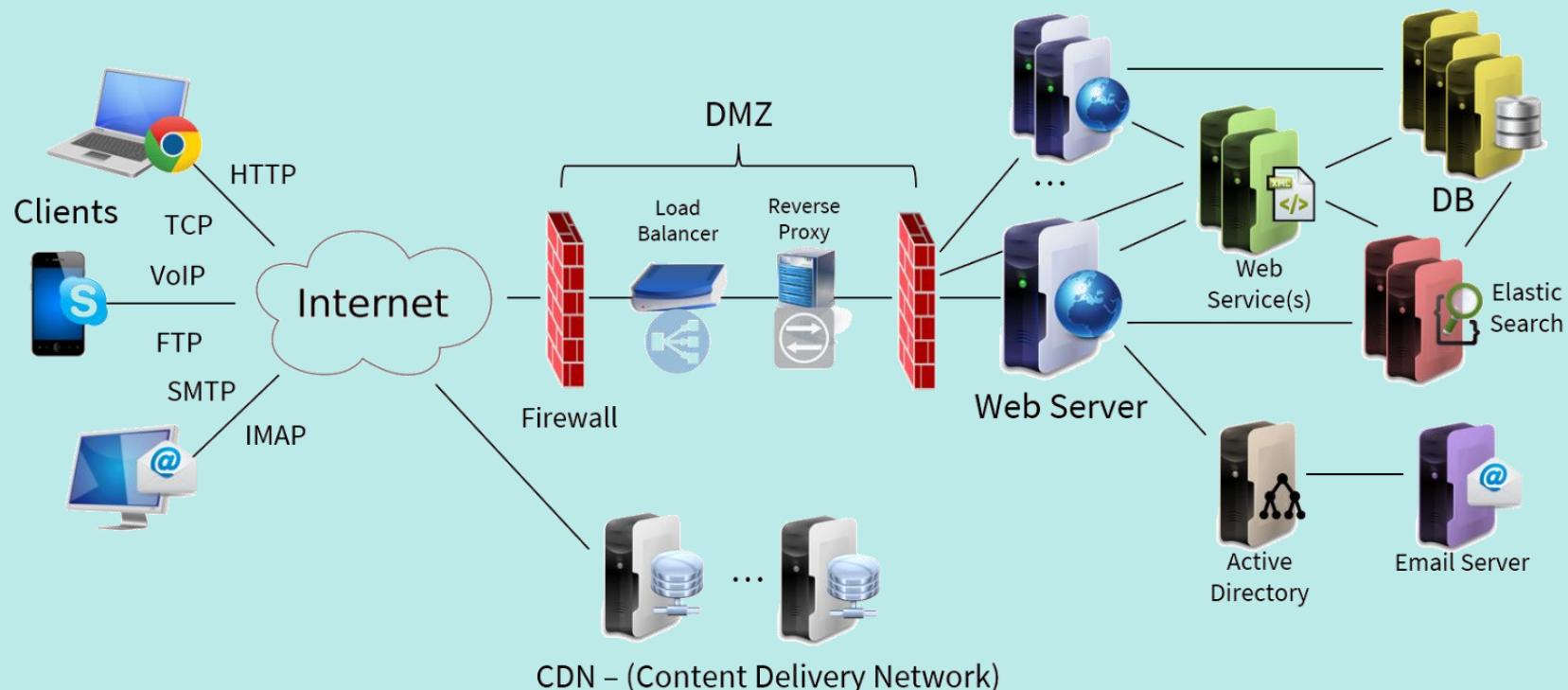
Endurance/Soak

- long periods of time
- memory leaks
- performance degradation
- reliability of the system

Scalability

- ability to handle a growing amount
- scale up, scale out

Application Network Architecture



• Web Page Performance Analysis



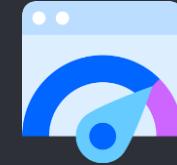
Analyze the content of a web page
Generate suggestions to make that page faster

● Web Page Performance Analysis

Good practices

- Minify HTML / CSS / JavaScript
- Prioritize visible content
- Avoid landing page redirects
- Leverage browser caching
- Optimize images
- Enable compression
- Remove Render-Blocking JavaScript

Google



PageSpeed Insights

<https://pagespeed.web.dev/>

DEMO

• Load Testing



To understand the behavior of the system under a **specific expected load** (e.g. multiple users).

- Load Testing - Tools



Open source Java application

Designed to load test functional behavior and measure performance



Open-source load and performance testing framework

Based on Scala, Akka and Netty



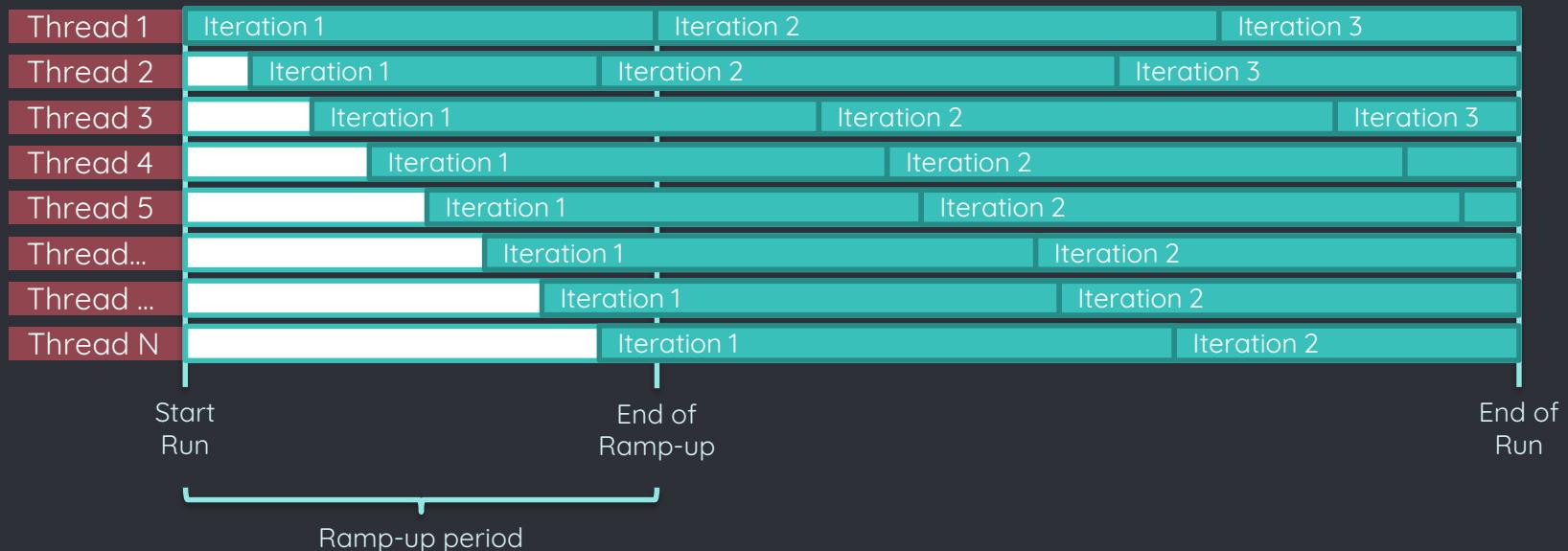
Open source load testing tool

Uses Python to define user behavior

- **Steps for performance (load) testing**

- Gather requirements
- Define scenario(s) – user journey
- Implement scenario – write JMeter script
- Configure script for execution
- Configure environment
- Run (on different configurations – if needed)
- Interpret and analyze results
- Report findings and possible improvements

• Threads, Iterations, Users & Ramp-up



N - Number of Threads (users)

Load Testing

DEMO



https://jmeter.apache.org/download_jmeter.cgi

<https://jmeter-plugins.org/wiki/PluginsManager/>

<https://blazedemo.com/>

Load testing - RESULTS

- ❑ Gather results and monitoring data
- ❑ Track script and execution notes (environment, configuration info)
- ❑ Interpret results, suggest improvements
- ❑ Plot results (e.g. average response time / # of threads)
- ❑ Historical comparison (previous builds)
- ❑ Summarize findings on each result type (load times, bottlenecks, errors, improvements, performance degradation)
- ❑ Include an executive summary

WHY?

DID YOU KNOW?



1 IN 4 VISITORS

would abandon a website that takes more than 4 seconds to load

46% OF USERS

don't revisit poorly performing websites

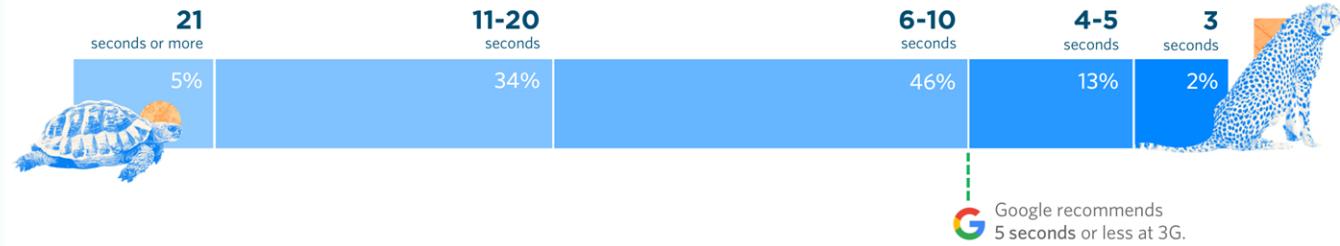
64% OF SHOPPERS

who are dissatisfied with their site visit will shop somewhere else next time

1 SECOND DELAY

reduces customer satisfaction by 16%

Most marketers' landing pages don't hit acceptable speeds





Ask us anything!

- Takeaways



Cannot cover everything, therefore, a subset of journeys/pages can be used and balanced based on the real/estimated usage.



Configure and adjust the testing based on the real usage of the application and the behavior of the end-users.

Check out our summer internships!

Testing



.NET



Java

Not open yet!

Unlock your potential with our internship program - gain real-world experience and jumpstart your career!



Verificarea și Validarea Sistemelor Soft

Curs 8. Corectitudinea programelor (Floyd. Hoare. Dijkstra)
Partea a II-a

Lector dr. Camelia Chisălită-Crețu

Universitatea Babeș-Bolyai
Cluj-Napoca

25 Aprilie 2023

1 Metode pentru demonstrarea corectitudinii

- Instrucțiuni cu sentinelă. Non-determinism
- Derivarea formală a programelor

2 Dezvoltarea algoritmilor corecți din specificații

- Rafinare
- Reguli de rafinare

3 Analiză statică. Analiză dinamică

4 Urmează...

5 Bibliografie

Instrucțiuni cu sentinelă

● **instrucțiune cu sentinelă (engl. guarded command)**

- o listă de instrucțiuni prefixată de o expresie booleană;
- dacă expresia booleană este inițial evaluată la true atunci lista instrucțiunilor este eligibilă pentru execuție;
- sintaxă:
 - $< \text{guarded command} > ::= < \text{guard} > \rightarrow < \text{guarded list} >$
 - $< \text{guard} > ::= < \text{boolean expression} >$
 - $< \text{guarded list} > ::= < \text{statement} > \{ ; < \text{statement} > \}$
 - $< \text{guarded command set} > ::=$
 $\qquad < \text{guarded command} > \{ \square < \text{guarded command} > \}$
 - $< \text{alternative construct} > ::= \text{if } < \text{guarded command set} > \text{ fi}$
 - $< \text{repetitive construct} > ::= \text{do } < \text{guarded command set} > \text{ od}$
 - $< \text{statement} > ::= < \text{alternative construct} > \mid$
 $\qquad < \text{repetitive construct} > \mid \text{"other statements"}$

Non-determinism. Exemple.

- **Exemplu . Maximul a două numere:**

```
if x ≥ y → m := x
□ y ≥ x → m := y
fi
```

Cea mai slabă precondiție

- **Hoare** – introduce precondiția suficientă astfel încât algoritmul să obțină rezultate corecte;
 - totuși nu există certitudinea că algoritmul se va termina;
- **Dijkstra** – introduce precondiția necesară și suficientă astfel încât algoritmul să permită obținerea rezultatului corect;
 - cea mai slabă precondiție (*engl. weakest precondition, wp*)
 - $wp(S, R)$, unde
 - S – mulțime de instrucțiuni;
 - R – predicat (condiție) asupra stării programului;
 - pornind execuția instrucțiunilor S dintr-o stare p , execuția se termină și starea în care se ajunge satisfacă pe R ;
 - wp - transformă o precondiție într-o postcondiție R (*engl. predicate transformer*).

Proprietățile wp [Dij75]

1 legi:

- Legea miracolului exclus;
- Legea monotoniei;
- Legea conjuncției;
- Legea disjuncției.

2 operatori:

- atribuire (`:=`);
- concatenare (`;`);

3 structuri:

- secvențială;
- alternativă;
- repetitivă.

Proprietățile wp [Fre10]

1 Legea miracolului exclus

pentru orice S , pentru toate stările, unde $R = FALSE$ are loc:

$$\text{wp}(S, FALSE) = FALSE;$$

2 Legea monotoniei

pentru orice S și orice două post-condiții, astfel încât pentru toate stările $P \Rightarrow Q$, pentru toate stările are loc:

$$\text{wp}(S, P) \Rightarrow \text{wp}(S, Q);$$

3 Legea conjuncției

pentru orice S și orice două post-condiții P și Q , pentru toate stările:

$$\text{wp}(S, P) \wedge \text{wp}(S, Q) = \text{wp}(S, P \wedge Q);$$

4 Legea disjuncției

pentru orice S determinist și orice post-condiții P și Q , pentru toate stările:

$$(\text{wp}(S, P) \vee \text{wp}(S, Q)) \Rightarrow \text{wp}(S, P \vee Q).$$

Operatorul de atribuire și concatenare

● operatorul de atribuire ($:=$)

- semantica expresiei $x := E$ se poate descrie prin:
 - $\text{wp}("x := E", R) = R_E^x$, unde
 - R_E^x – o copie a predicatului R , pentru care, fiecare apariție a variabilei x este înlocuită de E .

● operatorul de concatenare ($:$)

- semantica expresiei de concatenare ; se poate descrie prin:
 - $\text{wp}("S1; S2", R) = \text{wp}(S1, \text{wp}(S2, R))$;
 - $S1, S2$ – blocuri de instrucțiuni;
 - R – postcondiție.

Structura alternativă

Definition

1. Fie IF descrisă prin **if** $B_1 \rightarrow SL_1 \quad \square \dots \square B_n \rightarrow SL_n$ **fi**. Fie BB descrisă prin $(\exists i : 1 \leq i \leq n : B_i)$, atunci $wp(IF, R) = (BB \wedge (\forall i : 1 \leq i \leq n : B_i \Rightarrow wp(SL_i, R)))$.

Theorem

Substituția simplă

1. Pentru $(\forall i : 1 \leq i \leq n : (Q \wedge B_i) \Rightarrow wp(SL_i, R))$ pentru toate stările, atunci $(Q \wedge BB) \Rightarrow wp(IF, R)$ are loc în toate stările.

- $t : SSet \rightarrow Z$, $SSet$ – domeniul stărilor;
- Fie $wdec(S, t)$ – cea mai slabă precondiție definită pentru S , pentru care funcția t descrește în starea finală, față de cea inițială.

Theorem

1. Pentru $(\forall i : 1 \leq i \leq n : (Q \wedge B_i) \Rightarrow wdec(SL_i, t))$, pentru toate stările se poate spune că $(Q \wedge BB) \Rightarrow wdec(IF, t)$ are loc în toate stările.

Construcția repetitivă

Definition

2. Fie DO descrisă prin **do** $B_1 \rightarrow SL_1 \square \dots \square B_n \rightarrow SL_n$ **od**. Fie $H_0(R) = (R \wedge \neg BB)$ și pentru $k > 0$, $H_k(R) = (wp(IF, H_{k-1}(R))) \vee H_0(R)$, atunci, prin definiție, $wp(DO, R) = (\exists k : k \geq 0 : H_k(R))$.

Theorem

3. Dacă pentru toate stările avem $(P \wedge BB) \Rightarrow (wp(IF, P) \wedge wdec(IF, t) \wedge t \geq 0)$ atunci pentru toate stările avem $P \Rightarrow wp(DO, P \wedge \neg BB)$.

Definition

3. T este condiția satisfăcută de toate stările și $wp(S, T)$ este cea mai slabă precondiție care garantează terminarea programului S .

Theorem

4. Dacă $(P \wedge BB) \Rightarrow wp(IF, P)$ pentru toate stările, atunci $(P \wedge wp(DO, T) \Rightarrow wp(DO, P \wedge \neg BB))$ pentru toate stările.

Rafinare

- Date de intrare: X ;
Date de ieșire: Z ;
pre-condiție: $\varphi(X)$
- program abstract
 $Z : [\varphi, \psi]$
post-condiție: $\psi(X, Z)$
- rafinare
 \prec – are semnificația se *rescrie prin...*
 $Z = P_0 \prec P_1 \prec P_2 \prec \dots \prec P_{n-1} \prec P_n$
- reguli de rafinare
 - regula atribuirii;
 - regula compunerii secvențiale;
 - regula alternanței;
 - regula iterației.

Rafinare [Fre10]

- **Regula atribuirii:**

$$[\varphi(v/e), \psi] \prec v := e$$

- **Regula compunerii secvențiale:**

$$[\eta_1, \eta_2] \prec [\eta_1, \gamma] \\ \qquad \qquad \qquad [\gamma, \eta_2]$$

(γ - **predicat auxiliar**

(engl. **middle predicate**))

- **Regula alternanței:**

$$cond = c_1 \vee c_2 \vee \dots \vee c_n;$$

$$[\eta_1, \eta_2] \prec$$

$$\text{if } c_1 \rightarrow [\eta_1 \wedge c_1, \eta_2]$$

$$\square \quad c_2 \rightarrow [\eta_1 \wedge c_2, \eta_2]$$

:

:

$$\square \quad c_n \rightarrow [\eta_1 \wedge c_n, \eta_2]$$

fi

- **Regula iterației:**

$$cond = c_1 \vee c_2 \vee \dots \vee c_n$$

$$[\eta, \eta \wedge \neg cond] \prec$$

$$\text{do } c_1 \rightarrow [\eta \wedge c_1, \eta \wedge TC]$$

$$\square \quad c_2 \rightarrow [\eta \wedge c_2, \eta \wedge TC]$$

:

$$\square \quad c_n \rightarrow [\eta \wedge c_n, \eta \wedge TC]$$

od

Exemple

- **Rafinare.pdf.**

Instrumente software pentru analiza statică și analiza dinamică

- ESC2Java - Extended Static Checker to Java - **Seminar 06**;
- JML- Java Modeling Language - **Seminar 06**;

Pentru examen...

● teoria Dijkstra

- **rafinare:** definiții reguli;
- rafinare algoritmi din specificații
(link: [Rafinare.pdf](#))
(4 probleme – [Seminar 06](#)):
 - împărțire întreagă (cât și rest);
 - rădăcină pătrată;
 - înmulțire prin adunări repetate;
 - cel mai mare divizor comun a două numere naturale.

Urmează...

- Curs 9: Raportarea bug-urilor

Bibliografie I

- [Dij75] E. Dijkstra.
Guarded commands, nondeterminacy and formal derivation of programs.
CACM, 8(18):453–457, 1975.
- [Fre10] M. Frentiu.
Verificarea și validarea sistemelor soft.
Presa Universitară Clujeană, 2010.

Verificarea și Validarea Sistemelor Soft

Curs 8. Corectitudine (Floyd. Hoare. Dijkstra)

Partea I

Lector dr. Camelia Chisăliță-Crețu

Universitatea Babeș-Bolyai
Cluj-Napoca

25 Aprilie 2023

1 Evaluarea calității softului

- Evaluarea calității softului
- Verificarea programelor

2 Metoda lui Floyd

- Metoda aserțiunilor inductive
- Metoda lui Floyd. Parțial corectitudine
- Metoda lui Floyd. Terminare

3 Axiomatizarea lui Hoare

- Triplete Hoare. Semantică
- Parțial corectitudine. Reguli deductive
- Total corectitudine. Reguli deductive

4 Bibliografie

Evaluarea calității softului

- **calitatea softului –**

- conformitatea cu **cerințele funcționale și de performanță** precizate, **documentate explicit** în **standardele de dezvoltare și caracteristici implicate** ale unui produs soft dezvoltat. [Scott Pressman, 2005]

- **corectitudine** – proprietate a unui program de a respecta specificațiile și a oferi rezultate corecte [Fre10].

Verificarea programelor

- **metode formale** pentru verificarea programelor:

- bazate pe demonstrarea corectitudinii:
 - asistate de calculator, presupune *verificarea corectitudinii codului sursă asociat programului*;
 - aplicate programelor care trebuie să se termine și să obțină un rezultat (**curs 08**);
- bazate pe modele:
 - automate, presupune *verificarea proprietăților programului*;
 - aplicate sistemelor concurente; se aplică în etapele post-dezvoltare, e.g., verificarea modelelor.

Metode pentru demonstrarea corectitudinii programelor

- Metoda lui Floyd –
 - metoda aserțiunilor inductive;
- Axiomatizarea lui Hoare –
 - axiome și reguli deducțive;
 - dezvoltarea algoritmilor din specificații;
- Limbajul lui Dijkstra –
 - instrucțiuni cu santinelă;
 - non-determinism;
 - derivarea formală a programelor.

● Robert W Floyd
(8 Iunie 1936 – 25 Septembrie 2001)



● Sir Charles Antony Richard Hoare
(11 January 1934)



● Edsger Wybe Dijkstra
(11 Mai 1930 – 6 August 2002)



Metoda Iui Floyd. Metoda aserțiunilor inductive

Aplicabilitate:

- pentru a demonstra:
 - ① parțial corectitudinea programului;
 - ② terminarea programului;
 - ③ **total corectitudinea = parțial corectitudinea programului + terminarea programului.**

Folosește:

- **precondiție** – condiția satisfăcută de datele de intrare ale programului;
- **postcondiție** – condiția care trebuie satisfăcută de rezultatele programului;
- **algoritm** – descrierea programului (codul sursă);

Etape de aplicare:

- ① identificarea unui punct de tăietură în fiecare buclă;
- ② identificarea unei multimi de aserțiuni inductive;
- ③ construirea și demonstrarea condițiilor de verificare/terminare.

Parțial corectitudine. Etape de realizare

- ① se aleg puncte de tăietură în cadrul algoritmului:
 - două puncte de tăietură particulare: un punct de tăietură la începutul algoritmului, un punct de tăietură la sfârșitul algoritmului;
 - cel puțin un punct de tăietură în fiecare instrucțiune repetitivă;
- ② pentru fiecare punct de tăietură se alege câte un predicat invariant (aserțiune):
 - punctul de intrare - $\varphi(X)$;
 - punctul de ieșire - $\psi(X, Z)$;
- ③ se construiesc și se demonstrează condițiile de verificare:
 - ① $\forall X \forall Y (P_i(X, Y) \wedge R_{\alpha_{i,j}}(X, Y) \rightarrow P_j(X, r_{\alpha_{i,j}}(X, Y)))$;
 - ② Y – vector de variabile cu rezultate intermediare;
 - ③ $\alpha_{i,j}$ – drumul de la punctul de tăietură i la punctul de tăietură j ;
 - ④ P_i și P_j – predicate invariante în punctele de tăietură i și j asociate;
 - ⑤ $R_{\alpha_{i,j}}(X, Y)$ – predicat care dă condiția de parcurgere a drumului α ;
 - ⑥ $r_{\alpha_{i,j}}(X, Y)$ – funcție care indică transformările variabilelor Y de pe drumul α ;

Theorem

1. Dacă toate condițiile de verificare sunt adevărate atunci programul P este parțial corect în raport cu specificația $(\varphi(X), \psi(X, Z))$. [Fre10]

Parțial corectitudine. Exemplu.

- algoritmul pentru ridicarea la putere prin înmulțiri repetitive: $z = x^y$;

- Algoritmul putere(x, y, z) este:

A: $\varphi(X) ::= (x > 0 \wedge y \geq 0)$

$z := 1; u := x; v := y;$

cattimp ($v > 0$) execută

B: $\eta(X, Y) ::= z * u^v = x^y$

dacă ($v \% 2 == 0$)

atunci $u := u * u; v := v/2;$

altfel $v := v - 1; z := z * u;$

sfdacă

sfcattimp

C: $\psi(X, Z) ::= z = x^y$

sfAlg;

- se aleg punctele de tăietură: A, B și C;

- se stabilesc predicatele invariante pentru punctele de tăietură alese: $\varphi(X)$, $\psi(X, Z)$ și $\eta(X, Y)$;

- drumurile α între punctele de tăietură: $\{\alpha_{AB}, \alpha_{BB}, \alpha_{BC}, \alpha_{AC}\}$

$\Rightarrow \{\alpha_{AB}, \alpha_{BB}_{\text{atunci}}, \alpha_{BB}_{\text{altfel}}, \alpha_{BC}, \alpha_{AC}\}$;

- $R_{\alpha_{i,j}}(X, Y)$ – predicate pentru parcurgerea drumurilor $\alpha_{i,j}$;

- $r_{\alpha_{i,j}}(X, Y)$ – funcții care indică transformările variabilelor Y de pe drumurile $\alpha_{i,j}$;

- pentru fiecare drum α se construiește și se demonstrează condiția de verificare de forma

$\forall X \forall Y (P_i(X, Y) \wedge R_{\alpha_{i,j}}(X, Y) \rightarrow P_j(X, r_{\alpha_{i,j}}(X, Y)))$;

Terminarea algoritmului. Etape de realizare.

- ① se aleg punctele de tăietură în cadrul algoritmului;
- ② pentru fiecare punct de tăietură se alege câte un predicat invariant;
- ③ se alege o **mulțime convenabilă** M (i.e., o mulțime parțial ordonată, care nu conține nici un sir descrescător infinit) și o funcție descrescătoare u_i ;
 - în punctul de tăietură i funcția aleasă este $u_i : D_X \times D_Y \rightarrow M$;
- ④ se scriu condițiile de terminare:
 - condiția de terminare pe drumul $\alpha_{i,j}$ este:
$$\forall X \forall Y (\varphi(X) \wedge R_{\alpha_{i,j}}(X, Y) \rightarrow (u_i(X, Y) > u_j(X, r_{\alpha_{i,j}}(X, Y))))$$
 - dacă s-a demonstrat parțial corectitudinea, atunci condiția de terminare poate fi:
$$\forall X \forall Y (P_i(X) \wedge R_{\alpha_{i,j}}(X, Y) \rightarrow (u_i(X, Y) > u_j(X, r_{\alpha_{i,j}}(X, Y))))$$
- ⑤ se demonstrează condițiile de terminare:
 - la trecerea de la un punctul de tăietură i la j valorile funcției u descresc, i.e., $u_i > u_j$.

Theorem

2. Dacă toate condițiile de terminare sunt adevărate atunci programul P se termină în raport cu predicatul $\varphi(X)$. [Fre10]

Sistemul axiomatic al lui Hoare

● Relații și notații:

- **deductibilitate:** \models ;
 $g_1, g_2, \dots, g_m \models h$ are semnificația: “formula predicativă h (concluzia) este deductivă din formulele predicative g_1, g_2, \dots, g_m (premisele)”;;
- **implicația:** \Rightarrow ;
 $P \Rightarrow P'$ are semnificația: “dacă P este satisfăcut atunci are loc și P' ”;
- **negația:** \neg ;
 $\neg b$ are semnificația: “negația expresiei logice b ”;

● contribuțiile lui Hoare:

- **triplet** – precondiție, bloc de instrucțiuni, postcondiție;
- **axioma atribuirii** pentru: instrucțiunea de atribuire;
- **reguli deductive** pentru: structura secvențială, structura alternativă și structura repetitivă;
- **demonstrarea parțial și total corectitudinii, dezvoltarea corectă a algoritmilor folosind triplete.**

Triplete Hoare

- $\{\varphi\} P \{\psi\}$ – triplet Hoare, unde:
 - φ este precondiția;
 - ψ este postcondiția;
- notația are semnificația:
“dacă execuția programului P începe dintr-o stare care satisface φ , atunci starea în care se ajunge după execuția lui P va satisface ψ ”;

Triplete Hoare. Exemple (1)

- Care dintre următoarele triplete sunt valide?
 - ① $\{x = 5\} \quad x := x * 2 \quad \{true\};$
 - ② $\{x = 5\} \quad x := x * 2 \quad \{x > 0\};$
 - ③ $\{x = 5\} \quad x := x * 2 \quad \{x = 10 \parallel x = 5\};$
 - ④ $\{x = 5\} \quad x := x * 2 \quad \{x = 10\};$
- toate tripletele sunt valide;
- $\{x = 5\} \quad x := x * 2 \quad \{x = 10\}$ – cel mai util triplet;
- $\{x = 10\}$ – cea mai puternică postcondiție.

Triplete Hoare. Exemple (2)

- Care dintre următoarele triplete sunt valide?
 - 1 $\{x = 5 \&\& y = 10\} z := x/y \{z < 1\}$;
 - 2 $\{x < y \&\& y > 0\} z := x/y \{z < 1\}$;
 - 3 $\{y \neq 0 \&\& x/y < 1\} z := x/y \{z < 1\}$;
- toate tripletele sunt valide;
- $\{y \neq 0 \&\& x/y < 1\} z := x/y \{z < 1\}$ – cel mai util triplet;
- $\{y \neq 0 \&\& x/y < 1\}$ – cea mai slabă precondiție.

Semantica tripletelor Hoare

● corectitudine parțială

- notație: $\models_{par} \{\varphi\}P\{\psi\}$
- **tripletul $\{\varphi\}P\{\psi\}$ este satisfăcut relativ la corectitudinea parțială**, dacă pentru orice stare care satisfacă φ , starea rezultată după execuția programului P satisfacă postcondiția ψ , având condiția că *programul se termină*;
- nu garantează că P se termină;

● corectitudine totală

- notație: $\models_{tot} \{\varphi\}P\{\psi\}$
- **tripletul $\{\varphi\}P\{\psi\}$ este satisfăcut relativ la corectitudinea totală**, dacă pentru orice stare care satisfacă φ , programul P se termină, iar starea rezultată după execuția programului P satisfacă postcondiția ψ ;
- garantează că P se termină.

Parțial corectitudine. Reguli deductive

- axioma atribuirii;
- regula compunerii secvențiale;
- regula consecinței;
- regula alternanței;
- regula iterației.

Axioma atribuirii

- $\models_{par} \{\varphi(x|e)\} \ x := e \ \{\psi(x)\}$
are semnificația "dacă prin înlocuirea lui x în $\varphi(x)$ cu e obținem o afirmație adevărată, atunci după atribuirea $x := e$ afirmația $\psi(x)$ va fi adevărată."
- Fie tripletul $\{P\} \ X := Y + 2 \ \{Q\}$
 - fiind dat Q , care este predicatul pentru care P are loc?
 - pentru orice P astfel încât $[P \Rightarrow \langle X \leftarrow Y + 2 \rangle (Q)]$

Regula compunerii secvențiale

- dacă $\models_{par} \{\varphi\} S\{\omega\}$ și $\models_{par} \{\omega\} T\{\psi\}$
atunci $\models_{par} \{\varphi\} S; T\{\psi\}$;

Regula consecinței

- dacă

$\varphi_1 \Rightarrow \varphi_2$, $\models_{par} \{\varphi_2\} P \{\psi_2\}$ și $\psi_2 \Rightarrow \psi_1$

atunci

$\models_{par} \{\varphi_1\} P \{\psi_1\}$

Regula alternanței

• dacă

$\models_{par} \{\varphi \wedge cond\} S\{\psi\}$ și $\models_{par} \{\varphi \wedge \neg cond\} T\{\psi\}$
atunci propoziția

$\{\varphi\} IF (cond) THEN S ELSE T END\{\psi\}$ este parțial corectă în raport cu specificația (φ, ψ) .

Regula iterației

- Care sunt condițiile de realizare ale structurii repetitive while, astfel încât:
 $\{\varphi\} WHILE (cond) DO S END \{\psi\}$
 - presupunem că instrucțiunea while se termină , i.e., $\neg cond$;
 - în general, nu se cunoaște de câte ori se va executa S;
- considerăm un predicat η care rămâne satisfăcut după execuția S:
 - $\{\eta\}S\{\eta\}$ η este un **predicat invariant**;
 - la ieșirea din buclă avem $\eta \wedge \neg cond$;
 - pentru stabilirea post-condiției, $\{\eta\}$ trebuie ales astfel încât $[\eta \wedge \neg cond \Rightarrow \psi]$.

Regula iterației (cont.)

- dacă $\models_{par} \{\varphi \wedge cond\} S \{\psi\}$
atunci $\{\varphi\} WHILE (cond) DO S END \{\psi\}$,
cu condiția că există un predicat invariant η asociat buclei, astfel încât:

- $[\varphi \Rightarrow \eta]$ η este satisfăcut la intrare în buclă;
- $[\eta \wedge \neg cond \Rightarrow \psi]$ η obține pe ψ la ieșirea din buclă;
- $\{cond \wedge \eta\} S \{\eta\}$ η este satisfăcut la fiecare iterare.

Regula iterației. Exemple

Demonstrarea parțial corectitudinii folosind regula iterației:

- **Exemplu 1.** $z = 2^N$;

Dezvoltarea algoritmilor (parțial corecți), folosind regula iterației:

- **Exemplu 2.** $R = A * B$;
- **Exemplu 3.** $R = A^B$.

Regula iterației. Exemplu 1.

- efectuarea calculului: $z = 2^N$:

• $\varphi : \{N \geq 0\}$
 $m := 0; y := 1;$
 $\eta : \{y = 2^m\}$
 $WHILE (m! = N) DO \eta : \{y = 2^m\}$
 $y := 2 * y;$
 $m := m + 1$
 END
 $\psi : \{y = 2^N\}$

- trebuie demonstrat că invariantul η

- este satisfăcut la intrare în buclă;
- rămâne satisfăcut în buclă $\{\eta\}$ $y := 2 * y; m := m + 1; \{\eta\}$
- obține post-condiția $[\eta \wedge (m = N) \Rightarrow (y = 2^N)]$.

Regula iterației. Exemplu 2.

- Înmulțire prin adunări repetitive – “ R este A adunat de B ori”: $R = A * B$:

- $\varphi : \{B \geq 0\}$
- $\psi : \{R = A * B\} \Rightarrow \{B \geq 0\} S \{R = A * B\}$
- rezolvare (dezvoltarea tripletului):

$\varphi : \{B \geq 0\}$
“init R”
WHILE (cond) DO
“update R”
END
 $\psi : \{R = A * B\}$

- **regulă:** se înlocuiește în postcondiția ψ unul din termeni cu o variabilă pentru a obține predicatul invariant η asociat buclei, astfel încât $[(\eta \wedge \neg cond) \Rightarrow \psi]$;
 - se introduce variabila b în ψ și se determină invariantul η asociat buclei, descris prin: $R = A * b$;
 - pentru a obține postcondiția, se alege **cond** să fie $(b \neq B)$, unde $[(R = A * b) \wedge \neg(b \neq B) \Rightarrow (R = A * B)]$.

Regula iterației. Exemplu 2 (cont.)

- Înmulțire prin adunări repetate:

- invariantul – η : $(R = A * b)$;
- condiția de execuție a buclei (santinela) – cond: $(b \neq B)$;
- pentru a asigura că invariantul este satisfăcut inițial, se efectuează inițializarea: $R := 0; b := 0$;
- în fiecare iterare: (1) b este incrementat cu 1; (2) R este actualizat, obținând:

$$\varphi : \{B \geq 0\}$$

$$R := 0; b := 0;$$

WHILE $(b \neq B)$ DO $\eta : \{R = A * b\}$

$$R := ? \Rightarrow R := R + A$$

$$b := b + 1$$

END

$$\psi : \{R = A * B\}$$

Regula iterativă. Exemplu 3.

- ridicare la putere prin înmulțiri repetitive – “ R este A înmulțit de B ori”:

$$R = A^B;$$

- $\{ \varphi : (A > 0) \wedge (B \geq 0) \} \leq \{ \psi : R = A^B \}$
- rezolvare (dezvoltarea tripletului):
 - pentru obținerea invariantului se înlocuiește în ψ o constantă cu o variabilă, obținându-se: $\eta : R = A^b$;

$$\varphi : \{(A > 0) \wedge (B \geq 0)\}$$

$$R := ?; b := 0; \Rightarrow R := 1;$$

WHILE ($b \neq B$) DO $\eta : \{R = A^b\}$

$$R := ?; \Rightarrow R := R * A;$$

$$b := b + 1$$

END

$$\psi : \{R = A^B\}$$

Total corectitudine. Reguli deductive

- **atribuire**

$\{\varphi\} X := E \{\psi\}$ cu condiția că $[\varphi \Rightarrow (X \leftarrow E)(\psi)]$;

- **compunere**

$\{\varphi\} S; T \{\psi\}$ cu condiția că
există R astfel încât $\{\varphi\} S \{R\}$ și $\{R\} T \{\psi\}$;

- **alternanță**

$\{\varphi\} IF (cond) THEN S ELSE T END \{\psi\}$ cu condiția că
 $\{\varphi \wedge cond\} S \{\psi\}$ și $\{\varphi \wedge \neg cond\} T \{\psi\}$

- **Observație:** similar cu regulile corectitudinii parțiale!

Total corectitudine. Iterația.

- fie tripletul $\{\varphi\} WHILE (cond) DO S END \{\psi\}$
- cum demonstrăm că execuția buclei se termină?
- soluție:
 - se identifică o expresie întreagă V astfel încât:
 - valoarea V este non-negativă (i.e., $V \geq 0$) și
 - valoarea V este strict descrescătoare la fiecare iterare, $\{V = K\} S \{V < K\}$
- V – “invariant al buclei”, expresia își păstrează caracteristicile de la o iterare la alta.

Total corectitudine. Exemplu

- ridicare la putere prin înmulțiri repetitive – “ R este A înmulțit de B ori”:

$$R = A^B;$$

- $\{(A > 0) \wedge (B \geq 0)\} \mathrel{\text{S}} \{R = A^B\}$
- invariantul buclei este: $\eta : R = A^b \wedge (B \geq b)$;
 $\varphi : \{(A > 0) \wedge (B \geq 0)\}$

$$R := 1; b := 0;$$

WHILE ($b \neq B$) DO $\eta : R = A^b \wedge (B \geq b)$;

$$R := R * A;$$

$$b := b + 1$$

END

$$\psi : \{R = A^B\}$$

- se definește V – o construcție care variază la nivelul buclei – descris prin expresia $(B - b)$;
- V este strict descrescătoare la fiecare iterare a buclei, deoarece $[(B - (b + 1)) < (B - b)]$
- Cum demonstrăm că V este o expresie non-negativă?
 - demonstrând că $(B \geq b)$ este un invariant al buclei.

Total corectitudine. Regula iterației (rezumat)

- pentru a demonstra

$\models_{tot} \{\varphi\} WHILE (cond) DO S END \{\psi\}$

se identifică un predicat invariant η al buclei și o expresie V , invariantă la nivelul buclei, astfel încât:

- η este satisfăcut inițial $[\varphi \Rightarrow \eta]$;
- η determină obținerea post-condiției prin condiția de ieșire din buclă $[(\eta \wedge \neg cond) \Rightarrow \psi]$;
- η se menține satisfăcut după execuția blocului S , i.e.,
 $\{\eta\} S \{\eta\}$;
- expresia V este strict descrescătoare la fiecare iterare
 $\{V = K\} S \{V < K\}$;
- expresia V este întotdeauna non-negativă;
 $[\eta \Rightarrow (V \geq 0)]$.

Pentru examen...

- metoda lui Floyd:
 - demonstrarea parțial corectitudinii, terminării și total corectitudinii ([Fre10], Cap.1) – probleme:
 - căutarea unei valori într-un sir ordonat (**Seminar 5**);
 - determinarea celui mai mare divizor comun a două numere naturale (**Seminar 5**);

Urmează...

- Limbajul Dijkstra

Bibliografie I

[Fre10] M. Frentiu.
Verificarea și validarea sistemelor soft.
Presa Universitară Clujeană, 2010.

Sumar

Exemplul 1. Împărțire întreagă (cât și rest)	1
Exemplul 2. Rădăcină pătrată	2
Exemplul 3. Înmulțire prin adunări repetate	3
Exemplul 4. Cel mai mare divizor comun al două numere naturale	4
Example 5. Raising a number to a power by multiplications	5
Example 6. Insertion	6
Example 7. InsertionSort	8

Exemplul 1. Împărțire întreagă (cât și rest)

Specificare

$$\varphi: (x \geq 0) \wedge (y > 0)$$

$$\psi: (x = q * y + r) \wedge (0 \leq r < y)$$

A ₀ :	Subalgoritmul ImplInt (x,y,q,r) este: [φ, ψ] sImplInt
------------------	--

Fie $\eta: (x = q * y + r) \wedge (0 \leq r)$ un predicat intermedian (middle predicate). Prin aplicarea regulii compunerii secvențiale:

A ₁ :	Subalgoritmul ImplInt (x,y,q,r) este: [φ, η] [η, ψ] sImplInt
------------------	--

Predicatul η devine true prin atribuirea $(q,r) := (0,x)$.

A ₂ :	Subalgoritmul ImplInt (x,y,q,r) este: $(q,r) \leftarrow (0,x)$ [$\eta, \eta \wedge r < y$] sImplInt
------------------	--

Predicatul η este un predicat invariant. Prin aplicarea regulii iterației:

A ₃ :	Subalgoritmul ImplInt (x,y,q,r) este: $(q,r) \leftarrow (0,x)$ DO $r \geq y$ ---> [$r \geq y \wedge \eta, \eta \wedge TC$] OD sImplInt
------------------	---

Dezvoltarea algoritmilor corecți din specificații

Pentru ca DO să se termine, r trebuie să scadă (să descrească); deoarece $r \geq y$, putem reduce valoarea lui r cu y, adică $r \leftarrow r - y$.

η trebuie să rămână true și în post-condiție, deci este necesar ca:

$$q * y + r = q * y + r - y + y = (q+1) * y + (r-y).$$

Astfel, r și q își modifică valoarea.

$A_4:$	Subalgoritmul ImplInt(x,y,q,r) este: $(q,r) \leftarrow (0,x)$ DO $r \geq y \rightarrow$ $(q,r) \leftarrow (q+1, r-y)$ OD sfImplInt
--------	---

Exemplul 2. Rădăcină pătrată

- $r = [\sqrt{n}]$; se știe că $r \leq \sqrt{n} < r+1$;
- post-condiția este $r^2 \leq n < (r+1)^2$;

Specificare:

$$\varphi: \quad n > 1$$

$$\psi: \quad r^2 \leq n < (r+1)^2$$

$$A_0$$

Subalgoritmul RadPatrata(n, r) este:

$$[\varphi, \psi]$$

endRadPatrata

Rescriem predicatul de ieșire în forma: $(r^2 \leq n < q^2) \wedge (q=r+1)$.

Folosim predicatul intermedian (middle predicate) $\eta ::= (r^2 \leq n < q^2)$.

$$A_1$$

Subalgoritmul RadPatrata (n, r) este:

$$[\varphi, \eta]$$

$$[\eta, \psi]$$

sfRadPatrata

Predicatul η devine true în A_1 pentru $r=0$ și $q=n+1$.

$$A_2$$

Subalgoritmul RadPatrata (n, r) este:

$$(q,r) \leftarrow (n+1,0)$$

$$[\eta, \eta \wedge (q=r+1)]$$

$$\{\psi\}$$

sfRadPatrata

Pentru A_2 se poate aplica regula iterăției.

$$A_3$$

Subalgoritmul RadPatrata (n,r) este:

$$(q,r) \leftarrow (n+1,0)$$

DO $q \neq r+1 \rightarrow$

$$[\eta \wedge q \neq r+1, \eta \wedge TC]$$

OD

sfRadPatrata

Pentru ca DO să se termine, este necesar ca r sau q să descrească; q-r trebuie să devină 1 la final.

Dezvoltarea algoritmilor corecți din specificații

Expresia $p=(q+r)/2$ satisfacă condiția $r < p < q$; iar $(q-r)$ se actualizează prin modificarea intervalului $[r,q]$ la $[r,p]$ sau $[p,q]$.

Dar η trebuie să rămână true în post-condiție, deci este necesar ca:

- dacă $(p^2 \leq n)$ atunci atribuirea $r \leftarrow p$ satisfacă invariantul η ;
- dacă $(p^2 > n)$ atunci atribuirea $q \leftarrow p$ satisfacă invariantul η .

A₄

Subalgoritmul RadPatrata (n,r) este:

```

( $q,r$ )  $\leftarrow$  ( $n+1,0$ )
DO  $q > r+1$  ->
     $p \leftarrow (q+r)/2$ 
    IF  $p^2 \leq n \rightarrow r \leftarrow p$ 
     $\square p^2 < n \rightarrow q \leftarrow p$ 
    FI
OD
sfRadPatrata

```

Exemplul 3. Înmulțire prin adunări repetitive

Specificare:

$$\varphi : (x \geq 0) \wedge (y \geq 0)$$

$$\Psi : z = x * y$$

A₀

Subalgoritmul Produs(x,y,z) este:

$[\varphi, \Psi]$

sfProdus

Post-condiția Ψ este satisfăcută dacă se utilizează un predicat intermedian η :

$$\eta ::= (z + u * v = x * y) \wedge (v \geq 0).$$

De asemenea, se aplică regula compunerii secvențiale:

A₁

Subalgoritmul Produs (x,y,z) este:

$[\varphi, \eta]$

$[\eta, \Psi]$

sfProdus

Programul abstract A₁ devine true prin atribuirea $(u,v,z) \leftarrow (x,y,0)$.

A₂

Subalgoritmul Produs (x,y,z) este:

$(z,u,v) \leftarrow (0,x,y)$

$[\eta, \Psi]$

sfProdus

Programul abstract $[\eta, \Psi]$ se poate rescrie prin $[\eta, \eta \wedge (v=0)]$, ceea ce permite aplicarea regulii iterăției:

A₃

Subalgoritmul Produs (x,y,z) este:

$(z,u,v) \leftarrow (0,x,y)$

DO $v \neq 0$ ->

$[\eta \wedge v \neq 0, \eta \wedge \text{TC}]$

OD

sfProdus

Dezvoltarea algoritmilor corecți din specificații

Pentru ca DO să se termine, este necesar să micșorăm pe v:

- prima posibilitate:
 - o $v \leftarrow v-1$; dar η trebuie satisfăcut și în postcondiție, deci este necesar ca:
 - $z+u^*v = z + u + u^*(v-1)$ și atribuirea $z \leftarrow z+u$ trebuie să aibă loc;
- a doua posibilitate:
 - o $v \leftarrow v/2$, dacă v este par; dar η trebuie satisfăcut și în post-condiție, deci este necesar ca:
 - $z+u^*v = z + (u^*2)^* v/2$ și atribuirea $(u,v) := (u+u, v/2)$ trebuie realizată.

 A_4

Subalgoritmul Produs (x,y,z) este:

```

 $(z,u,v) \leftarrow (0,x,y)$ 
DO  $v > 0\}$ 
  Do  $(v \% 2 == 0) \rightarrow$ 
     $(u,v) \leftarrow (u+u, v \text{ div } 2)$ 
  OD
   $(z,v) \leftarrow (z+u, v-1)$ 
OD
sfProdus
  
```

Exemplul 4. Cel mai mare divizor comun al două numere naturale

Specificare:

$$\varphi : x > 0, y > 0$$

$$\psi : d = \text{cmmmdc}(x,y)$$

 A_0

Subalgoritmul CMMDC (x,y,d) este:

[φ, ψ]
sfCMMDC

Predicatul intermediu $\eta ::= \text{cmmdc}(d,s) = \text{cmmdc}(x,y)$ este utilizat pentru a aplica regula compunerii secvențiale.

 A_1

Subalgoritmul CMMDC (x,y,d) este:

[φ, η]
[η, ψ]
sfCMMDC

Programul abstract A_1 devine true prin atribuirea $(d,s) = (x,y)$, folosind regula atribuirii:

 A_2

Subalgoritmul CMMDC (x,y,d) este:

($d,s \leftarrow (x,y)$)
[η, ψ]
sfCMMDC

Dacă $d=s$ atunci η implică pe ψ . Astfel, se poate scrie următorul program abstract:

 A_3

Subalgoritmul CMMDC (x,y,d) este:

($d,s \leftarrow (x,y)$)
[$\eta, \eta \wedge (d=s)$]
sfCMMDC

Prin aplicarea regulei iterăției se obține:

 A_2

Subalgoritmul CMMDC (x,y,d) este:

Dezvoltarea algoritmilor corecți din specificații

```
(d,s) ← (x,y)
DO d≠s →
    [η ∧ d≠s, η ∧ TC]
    OD
sfCMMDC
```

Pentru $d \neq s$ avem condițiile $d > s$ și $d < s$. Se știe că pentru $d > s$ avem $\text{cmmdc}(d,s) = \text{cmmdc}(d-s,s)$ și atribuirea $d \leftarrow d-s$ păstrează predicatul η invariant.

A₃

Subalgoritmul CMMDC(x,y,d) este:

```
(d,s) ← (x,y)
DO d≠s →
    IF d>s → d←d-s
    □ d<s → s←s-d
    FI
    OD
    CMMDC ← s
sfCMMDC
```

Example 5. Raising a number to a power by multiplications

Compute $z = x^y$ by multiple multiplications

Specification:

A ₀ :	$\varphi : (x > 0) \wedge (y \geq 0)$ $\Psi : z = x^y$
------------------	---

The predicate $\eta ::= (z * u^v = x^y) \wedge (v \geq 0)$ implies Ψ if $v=0$. Using it as a middle predicate we can apply the sequential composition rule:

A ₁ :	Subalgorithm RaisingPower(x,y,z) is: [φ, η] [η, Ψ] endRaisingPower
------------------	--

The η becomes true if $(z,u,v) = (1,x,y)$ (in the first abstract program):

A ₂ :	Subalgorithm RaisingPower(x,y,z) is: $(z,u,v) \leftarrow (1,x,y)$ [$\eta, \eta \wedge v=0$] endRaisingPower
------------------	--

The predicate η is invariant, we can apply the iteration rule.

A ₃ :	Subalgorithm Putere(x,y,z) is: $(z,u,v) \leftarrow (1,x,y)$ DO $v \neq 0 \rightarrow$ [$\eta \wedge v \neq 0, \eta \wedge \text{TC}$] OD endRaisingPower
------------------	---

For the DO to terminate we must decrease v :

First possibility: $v \leftarrow v-1$. But η should hold also in the post-condition, so we must have: $z * u^v = z * u * u^{v-1}$. So also the assignment $(z,v) \leftarrow (z * u, v-1)$ is needed.

Dezvoltarea algoritmilor corecți din specificații

Second possibility: $v \leftarrow v/2$, if v is even. . But η should hold also in the post-condition, so we must have: $z^* u^v = z^* (u^* u)^{v/2}$. So also the assignment $(u, v) \leftarrow (u^* u, v/2)$ is needed.

A4	Subalgorithm RaisingPower1 (x, y, z) is: $(z, u, v) \leftarrow (1, x, y)$ DO $v \neq 0$ $(z, v) \leftarrow (z^* u, v-1)$ OD endRaisingPower
----	--

A4	Subalgorithm RaisingPower2 (x, y, z) is: $(z, u, v) \leftarrow (1, x, y)$ DO $v \neq 0$ DO (v even) $\rightarrow (u, v) \leftarrow (u^* u, v/2)$ OD $(z, v) \leftarrow (z^* u, v-1)$ OD endRaisingPower
----	--

Example 6. Insertion

$A = (a_1, a_2, \dots, a_n)$ an array with n components ordered in decrease order and x a value. Insert x in A such that A remains ordered and A contains a new value x .

The predicate ORD is define by:

$$\text{ORD}(n, A) ::= (\forall i, j: 1 \leq i, j \leq n, i \leq j \Rightarrow a_i \leq a_j)$$

Specification:

$$\varphi ::= \text{ORD}(n, A) \wedge (n \text{ natural})$$

$$\psi ::= \text{ORD}(n+1, A) \text{ and } (A \text{ contains the initial elements and a new element } x)$$

$A_0:$	$[\varphi, \psi]$
--------	-------------------

There are two possibilities ($x < a_n$ and $n \neq 0$) or ($x \geq a_n$ or $(n=0)$):

$A_1:$	Subalgorithm Insert(n, A, x) is: Dacă $x < a_n$ și $n \neq 0$ atunci $[\varphi \wedge (x < a_n) \wedge n \neq 0, \psi]$ altfel $[\varphi \wedge ((x \geq a_n) \vee (n=0)), \psi]$ sfdacă endInsert
--------	---

A doua propoziție nestandard se rafinează printr-o atribuire

$A_2:$	Subalgorithm Insert(n, A, x) is: Dacă $x < a_n$ și $n \neq 0$ atunci $[\varphi \wedge (x < a_n) \wedge n \neq 0, \psi]$ altfel $(n, a_{n+1}) \leftarrow (n+1, x)$ sfdacă endInsert
--------	---

Dezvoltarea algoritmilor corecți din specificații

Să notăm prin η următorul predicat

$$\text{ORD}(n, A) \wedge [(x < a_1) \wedge (p=1) \vee (a_{p-1} \leq x < a_p) \wedge (1 < p \leq n)]$$

Care este o postcondiție pentru o problemă de căutare și să folosim regula secvenței. Ajungem la:

$A_3:$	<p>Subalgorithm Insert(n, A, x) is:</p> <p>IF $x < a_n$ and $n \neq 0 \rightarrow$ $[\varphi \wedge (x < a_n) \wedge n \neq 0, \eta]$ $[\eta, \psi]$ $\square(\text{not } x < a_n \text{ and } n \neq 0) \rightarrow (n, a_{n+1}) \leftarrow (n+1, x)$</p> <p>FI endInsert</p>
--------	---

Vom satisface postcondiția η în urma apelului subalgoritmului de căutare, astfel că ajungem la:

$A_4:$	<p>Subalgorithm Insert(n, A, x) is:</p> <p>IF $x < a_n$ and $n \neq 0 \rightarrow$ CALL SEARCH(x, n, A, p) $[\eta, \psi]$ $\square(\text{not } x < a_n \text{ and } n \neq 0) \rightarrow (n, a_{n+1}) \leftarrow (n+1, x)$</p> <p>FI endInsert</p>
--------	---

After the search we know that x is between a_{p-1} and a_p , so x must be inserted on position p , so we have

$$a'_{i+1} \leftarrow a_i, \text{ for } i=n, n-1, \dots, p$$

and

$$a'_p \leftarrow x.$$

$$n' \leftarrow n+1$$

We use the assignments:

$$i \leftarrow n;$$

DO $i \geq p \rightarrow$

$$a_{i+1} \leftarrow a_i$$

$$i \leftarrow i-1$$

OD

$A_5:$	<p>Subalgorithm Insert(n, A, x) is:</p> <p>IF $x < a_n$ and $n \neq 0 \rightarrow$ CALL SEARCH(x, n, A, p) $i \leftarrow n$ DO $i \geq p \rightarrow$ $a_{i+1} \leftarrow a_i$ $i \leftarrow i-1$ OD $a_p \leftarrow x$ $n \leftarrow n+1$ $\square(\text{not } x < a_n \text{ and } n \neq 0) \rightarrow (n, a_{n+1}) \leftarrow (n+1, x)$</p> <p>FI endInsert</p>
--------	---

Dezvoltarea algoritmilor corecți din specificații

Another refinement regarding the $n \leftarrow n+1$ assignment:

$A_5:$	Subalgorithm Insert(n, A, x) is: IF $x < a_n$ and $n \neq 0 \rightarrow$ CALL SEARCH(x, n, A, p) $i \leftarrow n$ DO $i \geq p \rightarrow$ $a_{i+1} \leftarrow a_i$ $i \leftarrow i - 1$ OD $a_p \leftarrow x$ $\square (\text{not } x < a_n \text{ and } n \neq 0) \rightarrow (n, a_{n+1}) \leftarrow (n+1, x)$ FI $n \leftarrow n+1$ endInsert
--------	--

Example 7. InsertionSort

Let $A = (a_1, a_2, \dots, a_n)$ be an array with n integer components. The problem requires to order the components of A .

Specification

$\varphi ::= n \geq 2, A$ has integer components

$\psi ::= \text{ORD}(n, A)$ and A has the same elements as in the precondition

$A_o:$	$[\varphi, \psi]$
--------	-------------------

We use the middle predicate $\text{ORD}(k, A)$ and apply the sequential composition rule:

$A_1:$	Subalgorithm InsertSort(n, A) is: $[\varphi, \text{ORD}(k, A)]$ $[\text{ORD}(k, A), \psi]$ endInsertSort
--------	---

The first abstract program may be refined to an assignment

$A_2:$	Subalgorithm InsertSort(n, A) is: $k \leftarrow 1$ $[\text{ORD}(k, A), \psi]$ endInsertSort
--------	--

We can rewrite the remained abstract program remarking that

$$\text{ORD}(k, A) \wedge (k=n) \Rightarrow \psi$$

$A_3:$	Subalgorithm InsertSort(n, A) is: $k \leftarrow 1$ $[\text{ORD}(k, A), \text{ORD}(k, A) \wedge (n=k)]$ endInsertSort
--------	---

We now can apply the iteration rule

Dezvoltarea algoritmilor corecți din specificații

A ₄ :	Subalgorithm InsertSort(n,A) is: k←1 DO k<n → [ORD(k,A) and k<n, ORD(k,A) and TC] OD endInsertSort
------------------	---

For the DO to terminate we must increase k:

First possibility: k←k+1. But $\eta(k) := \text{ORD}(k,A)$ invariant – by modifying k by k+1 the predicate $\eta(k|k+1)$ must be true.

A ₅ :	Subalgorithm InsertSort(n,A) is: k←1 DO k<n → [k<n and $\eta(k)$, $\eta(k k+1)$] OD endInsertSort
------------------	--

The abstract program

$$[(k < n) \wedge \text{ORD}(k,A), \text{ORD}(k+1,A)]$$

Corresponds to the following subproblem:

If $\text{ORD}(k,A)$ (the first k elements in A are ordered) then modify the A such that the first k+1 elements to be ordered. This can be achieved by calling a subalgorithm that inserts the a_{k+1} component such that after insertion the postcondition $\text{ORD}(k+1,A)$ is true.

A ₄ :	Subalgorithm InsertSort(n,A) is: k←1 DO k<n → CALL INSERT(k,A, a _{k+1}) OD endInsertSort
------------------	---

CURS 09.

RAPORTAREA BUG-URILOR

Verificarea și validarea sistemelor soft
[2 Mai 2023]

Lector dr. Camelia Chisăliță-Crețu
Universitatea Babeș-Bolyai

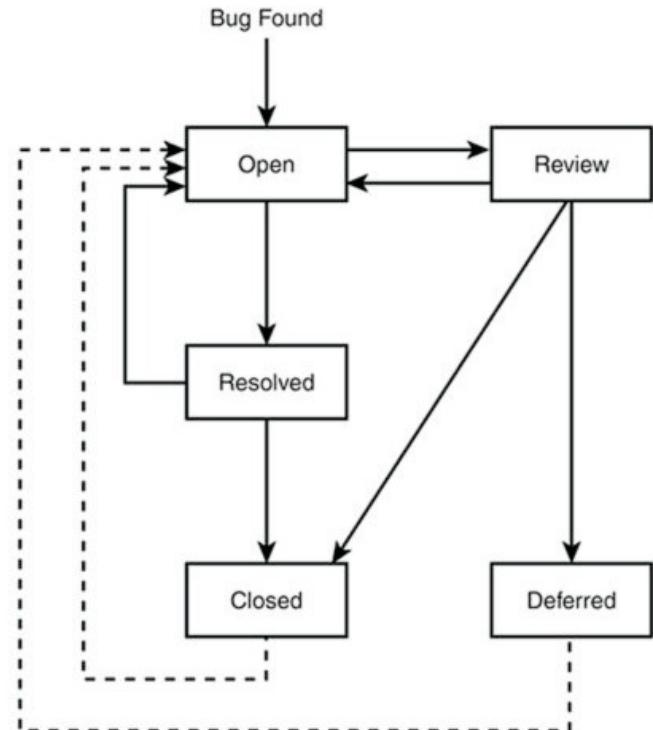
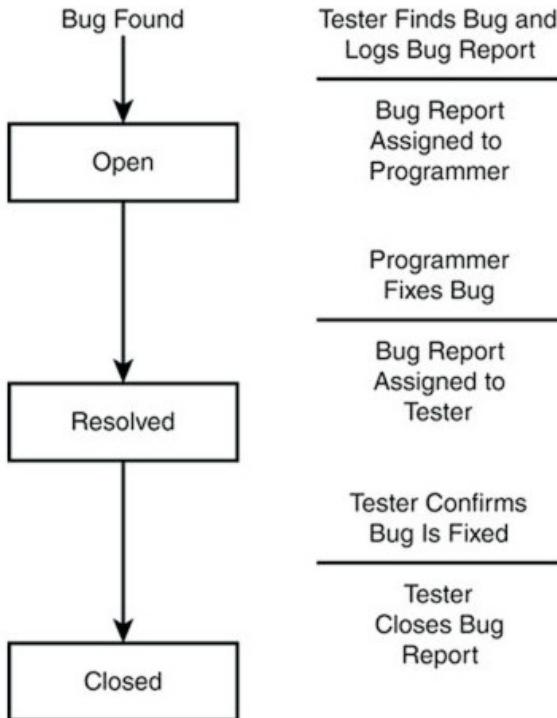
Conținut

- **Ciclul de viață al unui bug**
 - Etape ale ciclului de viață al unui bug
 - Abordări
 - Principii și reguli generale de raportare a bugurilor
- **RIMGEA**
 - Definiție. Componente
 - Aplicabilitate. Obiective
 - Reprezentare conceptuală
 - Replicate
 - Isolate
 - Maximize
 - Generalize
 - Externalize
 - Neutral tone
 - Tipuri de bug-uri
 - Bug de implementare
 - Bug de proiectare
 - Exemple
 - **Clasificarea bug-urilor pe baza atributelor de calitate**
 - **Bibliografie**

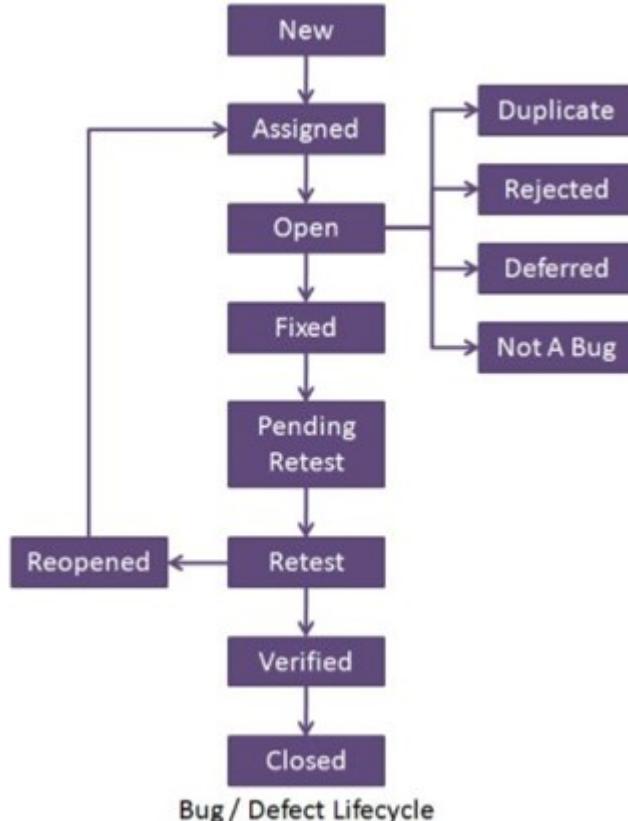
CICLUL DE VIAȚĂ AL UNUI BUG

Etape ale ciclului de viață al unui bug. Abordări
Principii și reguli generale de raportare a bugurilor

Ciclul de viață al unui bug (1)



Ciclul de viață al unui bug (2)



- stări ale unui bug stabilite de **tester**:
 - New, Pending Testing, Retest, Reopened, Verified, Closed;
- stări ale unui bug stabilite de **programator**:
 - Assigned, Open (Duplicate, Rejected, Deffered, Not a Bug), Fixed;
- **sursa**: [[ISTQBCertification2023](#)]

Principii generale de raportare a unui bug

- principii de raportare a unui bug [[Patton2005](#)]:
 - se raportează imediat după identificare;
 - se realizează o descriere a bug-ului;
 - nu se fac aprecieri subiective referitoare la bug-ul raportat;
 - se urmărește starea bug-urilor (corectat sau nu) raportate anterior, i.e., se folosește un sistem de monitorizare a bugurilor (*engl. bug tracking system*);

Principii și reguli generale de raportare a unui bug

- Izolarea și reproducerea bug-urilor [[Patton2005](#)]:
 - **fii suspicios** – nu te baza pe ceea ce au făcut sau spus alții, fii consecvent și riguros;
 - **acordă atenție timpului** – nu ignora durata de realizare a unei operații (e.g., momentul zilei, utilizarea unui device care lucrează încet, viteza de prelucrare a datelor, etc.);
 - **acordă atenție domeniilor de valori** – valori limită, volum de date mare, alocare și accesare a memoriei;
 - **starea unui bug** – poate fi mascată de încheierea aparent cu succes a unei operații; un bug poate fi evidențiat de execuția într-o anumită ordine a pașilor de execuție și nu de momentul în care a apărut;
 - dependențele existente între resursele utilizate și interacțiunea cu memoria, partajarea rețelei și a componentelor hard;
 - componente hard nu trebuie ignorate, acestea se pot degrada și reacționează imprevizibil.

RIMGEA

Definiție. Componente

Aplicabilitate. Obiective

Reprezentare conceptuală

Replicate. Isolate. Maximize. Generalize. Externalize. Clear communication

Bug de implementare. Bug de proiectare

Exemple

RIMGEA. Definiție. Componente

- RIMGEA [[BBST2008](#)]
 - grup de reguli utilizat pentru investigarea și îmbunătățirea descrierii unui bug;
- **componente**
 - Replicate – reproducerea bug-ului;
 - Isolate – izolarea bug-ului;
 - Maximize – maximizarea bug-ului;
 - Generalize – generalizarea bug-ului;
 - Externalize – externalizarea bug-ului;
 - And say it clear and dispassionately – atitudine neutră la raportarea bug-ului.

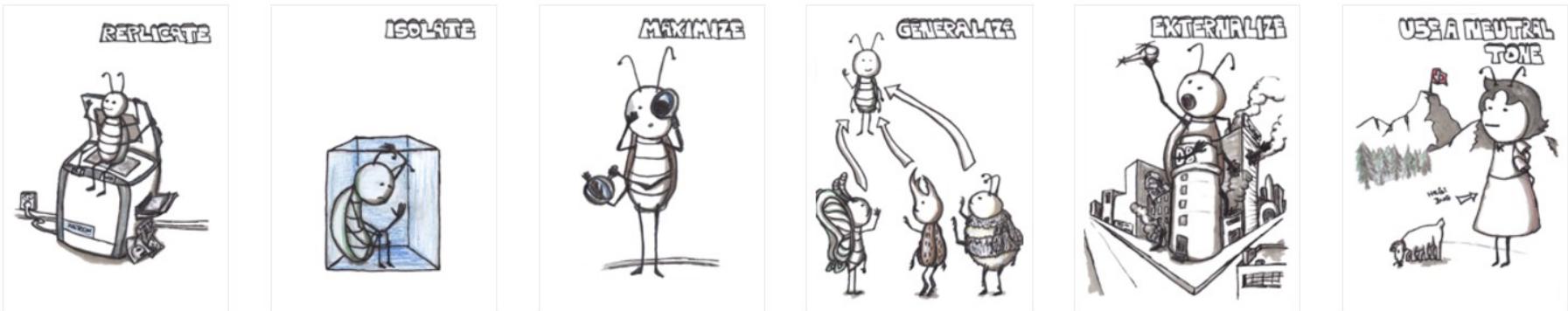
RIMGEA. Aplicabilitate

- tehnica RIMGEA se aplică pentru la raportarea:
 - **bug-urilor de implementare**
 - programul funcționează într-o manieră pe care proiectantul și programatorul o consideră nepotrivită, inadecvată;
 - pune în evidență o deficiență, i.e., greșală, de scriere sau implementare a soluției adoptate;
 - *A program with a coding error will behave in a way the designer, programmer or tester will agree is improper.*
 - **bug-urilor de proiectare**
 - programul funcționează conform proiectării și implementării;
 - pune în evidență o deficiență, i.e., greșală, în abordarea modului de rezolvare;
 - *A program with a design error behaves in a way the designer and programmer intended.*

RIMGEA. Obiective

- obiective:
 - **la raportarea bug-urilor de implementare**
 - elaborarea unui raport care să conțină **o listă minimală de pași** care să demonstreze cu certitudine existența unui bug în codul sursă;
 - **la raportarea bug-urilor de proiectare**
 - elaborarea unui raport care să precizeze clar **cum** este aspectul de proiectare defectuos și **felul în care** acesta reduce nejustificat calitatea softului.

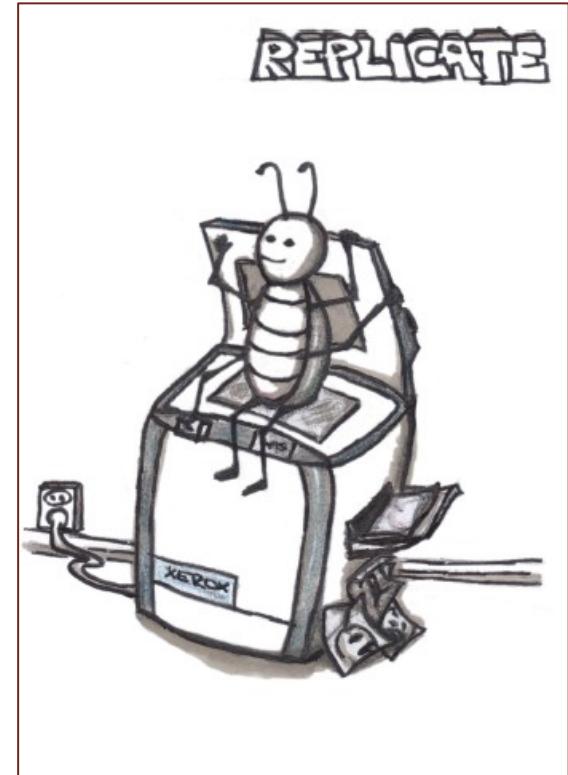
RIMGEA. Reprezentare conceptuală



sursa: [\[Altom2016\]](#)

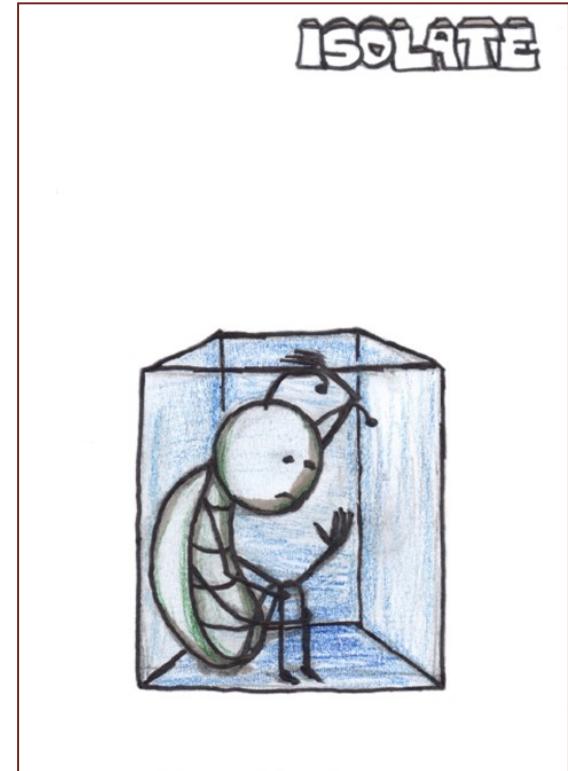
RIMGEA. Replicate

- **reproducerea** (*engl. replicate*):
 - activitate de testare prin care se identifică **ce** este necesar pentru ca bug-ul să apară de fiecare dată când se dorește manifestarea lui;
 - pune în evidență și cazurile în care bug-ul **nu** poate fi reprodus la cerere și descrie **factorii** care ar indica posibilitatea apariției acestuia.



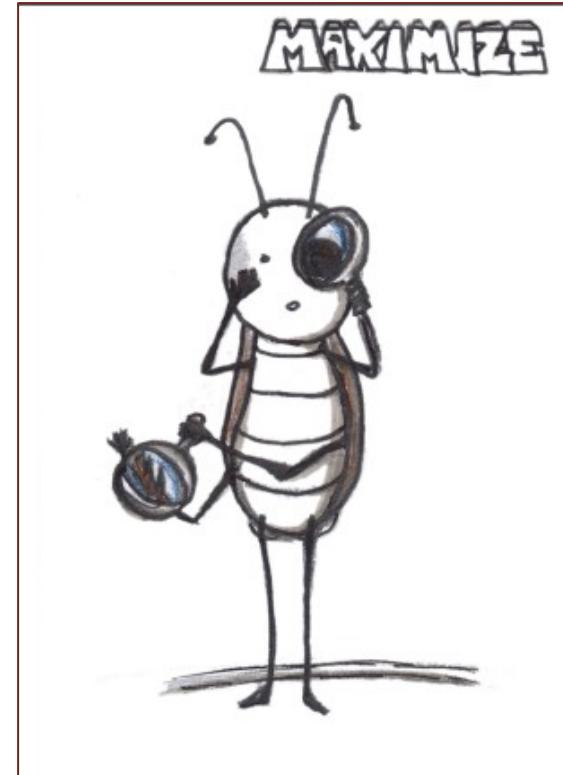
RIMGEA. Isolate

- **izolarea (engl. isolate):**
 - activitate de testare prin care se identifică **cea mai scurtă secvență de pași** necesară pentru ca bug-ul să fie reprodus și raportat într-o formă clară, astfel încât să se pună în evidență defecțiunea apărută;
 - un raport al unui bug prezintă o singură defecțiune a produsului soft.



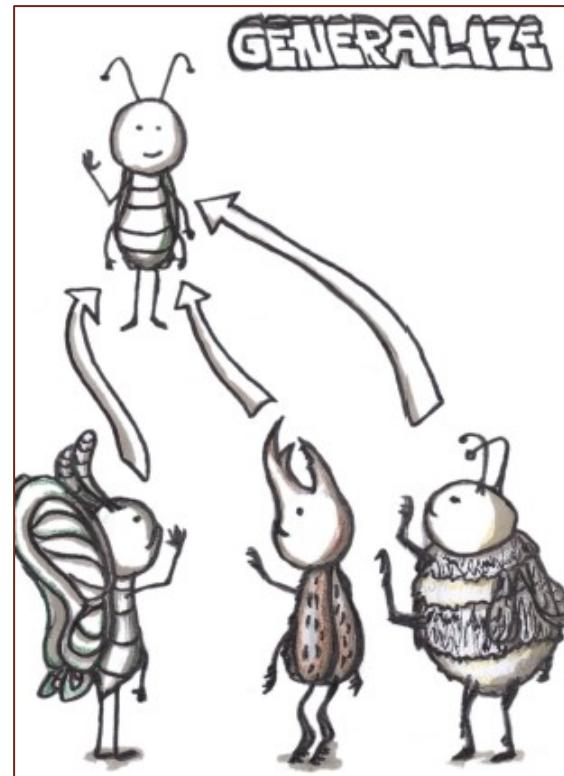
RIMGEA. Maximize

- maximizarea (*engl. maximize*):
 - activitate de testare prin care se identifică **cele mai importante (grave) consecințe** ale existenței bug-ului;
 - se recomandă prezentarea consecințelor frecvent întâlnite, determinate de prezența bug-ului, în cel mai des utilizate configurații de rulare, prelucrând date reale.



RIMGEA. Generalize

- generalizarea (*engl. generalize*):
 - activitate de testare prin care se identifică și se clasifică situațiile în care acest bug va cauza o defecțiune;



RIMGEA. Externalize

- **externalizarea (engl. externalize):**
 - activitate de testare prin care se identifică consecințele prezenței bug-ului din perspective diferite:
 - **utilizatorul propriu-zis:** Cum este afectat factorul uman?
 - **beneficiarul, clientul:** Cum este afectată reputația companiei clientului sau business-ul acestuia?
 - **dezvoltator, furnizor:** Cum este afectată reputația companiei dezvoltatorului sau business-ul acestuia?
 - **terț, competitor:** Cât de important este acest bug?



RIMGEA. And use a constructive communication

- comunicare clară și constructivă/neutră (engl. *and say it clearly and dispassionately, neutral tone*):
 - raportul unui bug conține
 - **date concrete sau speculative într-o manieră constructivă**, imparțială și corectă;
 - raportul unui bug nu conține
 - **nu se critică, nu se insultă persoane;**
 - nu se includ aspecte irelevante, care pot afecta alte persoane, doar dacă ele sunt esențiale pentru descrierea problemei și condițiilor care determină defecțiunea.



Bug de implementare vs. Bug de proiectare

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

RIMGEA. Exemple

- produs software testat:
 - Apache Open Office: <https://www.openoffice.org/>;
- tool pentru managementul bug-urilor
 - Bugzilla: <https://bz.apache.org/ooo/>;
- configurație de rulare folosită la testare și identificarea bug-urilor:
 - **sistem de operare:** Windows 10 Home, 64-bit;
 - **procesor:** Intel Core i5, 1.7GHz;
 - **memorie RAM:** 4Gb;
 - **versiune Open Office Writer:** 4.1.3.

Open Office Writer. Bug-uri investigate

- bug-uri de implementare:
 - **Bug 1.** eroare de rotunjire:
 - **Issue 120368 - Font size with decimal values don't have a consistent**
[\(https://bz.apache.org/ooo/show_bug.cgi?id=120368\)](https://bz.apache.org/ooo/show_bug.cgi?id=120368);
 - **Bug 2.** inconsistență de setare:
 - **Issue 127562 - Inconsistency on Header Height max. value when enabled first time**
[\(https://bz.apache.org/ooo/show_bug.cgi?id=127562\)](https://bz.apache.org/ooo/show_bug.cgi?id=127562);
- bug-uri de proiectare:
 - **Bug 3.** inconsistență de proiectare:
 - **Issue 126371 – Disappearing Vertical Text button in Drawing Toolbar in Writer v.4.1.1**
[\(https://bz.apache.org/ooo/show_bug.cgi?id=126371\)](https://bz.apache.org/ooo/show_bug.cgi?id=126371);

Open Office Writer. Bug 1 (1)

- **Bug 1.** eroare de rotunjire:
 - Issue 120368 - **Font size with decimal values don't have a consistent** (https://bz.apache.org/ooo/show_bug.cgi?id=120368);
 - Replicate – reproducerea bug-ului;
 - Isolate – izolarea bug-ului;
 - Maximize – maximizarea bug-ului;
 - Generalize – generalizarea bug-ului;
 - Externalize – externalizarea bug-ului;
 - And say it clear and dispassionately – atitudine neutră la raportarea bug-ului.

Open Office Writer. Bug 1 (2)

- **Bug 1.** eroare de rotunjire:
 - Issue 120368 - Font size with decimal values don't have a consistent (https://bz.apache.org/ooo/show_bug.cgi?id=120368);
- Replicate – reproducerea bug-ului;
 - **Scenariu A:**
 - Size curent: 20;
 - fereastra principală setare Size: 10.23:
 - fereastra principală: Size = 10.1;
 - fereastra Format: Size = 10.2;
 - **Scenariu B:**
 - Size curent: 20;
 - fereastra Format setare Size: 10.23:
 - fereastra principală: Size = 10.1;
 - fereastra Format: Size = 10.2;

Open Office Writer. Bug 1 (3)

- **Bug 1.** eroare de rotunjire:
 - Issue 120368 - Font size with decimal values don't have a consistent (https://bz.apache.org/ooo/show_bug.cgi?id=120368);
- Isolate – izolarea bug-ului;
 - identificarea celei mai scurte secvențe de pași prin care se evidențiază bug-ul;
 - E.g., Scenariu A, Scenariu B;

Open Office Writer. Bug 1 (4)

- **Bug 1.** eroare de rotunjire:
 - Issue 120368 - Font size with decimal values don't have a consistent (https://bz.apache.org/ooo/show_bug.cgi?id=120368);
- Maximize – maximizarea bug-ului;
 - maximizarea impactului negativ:
 - **impact negativ = inconsistență dintre cele două scenarii;**
 - întrebări:
 - unde este aplicat algoritmul de rotunjire?
 - câți/care sunt algoritmii de rotunjire folosiți?
 - eroarea de aproximare este mai puțin importantă față de inconsistență evidențiată de scenariile A și B ---> **variabila Font Size are două valori simultan 10.1 și 10.2;**

Open Office Writer. Bug 1 (5)

- **Bug 1.** eroare de rotunjire:
 - Issue 120368 - Font size with decimal values don't have a consistent (https://bz.apache.org/ooo/show_bug.cgi?id=120368);
- Generalize – generalizarea bug-ului;
 - **toate platformele au același bug:**
 - eroarea de aproximare raportată este mică;
 - puțini utilizatori folosesc size cu zecimale.

Open Office Writer. Bug 1 (6)

- **Bug 1.** eroare de rotunjire:
 - Issue 120368 - **Font size with decimal values don't have a consistent** (https://bz.apache.org/ooo/show_bug.cgi?id=120368);
- Externalize – externalizarea bug-ului;
 - utilizatorul identifică eroarea de aproximare ---> utilizatorul devine suspicios cu privire corectitudinea aproximărilor efectuate de OO Writer;
 - dacă scenariul B oferă o aproximare corectă din perspectiva utilizatorului, atunci experiența ca utilizator îi va fi afectată ---> va trebui să își amintească de fiecare dată când setează Font Size că Scenariul A nu este potrivit și să îl evite;
 - modul de abordare a acestor probleme de aproximare de către produsele concurente, e.g., Microsoft Office Word;

Open Office Writer. Bug 2 (1)

- **Bug 2.** inconsistență de setare:
 - **Issue 127562 - Inconsistency on Header Height max. value when enabled first time** (https://bz.apache.org/ooo/show_bug.cgi?id=127562);
 - Replicate – reproducerea bug-ului;
 - Isolate – izolarea bug-ului;
 - Maximize – maximizarea bug-ului;
 - Generalize – generalizarea bug-ului;
 - Externalize – externalizarea bug-ului;
 - And say it clear and dispassionately – atitudine neutră la raportarea bug-ului;

Open Office Writer. Bug 2 (2)

- **Bug 2.** inconsistență de setare:
 - Issue 127562 - Inconsistency on Header Height max. value when enabled first time (https://bz.apache.org/ooo/show_bug.cgi?id=127562);
- Replicate – reproducerea bug-ului;

Scenariu A

- 1. Open OpenOffice Writer;
- 2. Set the measurement unit to 'cm': Tools menu | Options... | OpenOffice Writer | General | Choose as Measurement Unit 'Centimeters';
- 3. Create a new document: Menu File | New | Text Document;
- 4. Set the Paper Format to A4: Format Menu | Page... | Page tab | Select from Format drop-down list;
- 5. go to Header Tab;
- 6. Enable Header;
- 7. Set Header Height to 29.7 cm;
- 8. Change focus from Hearder Height; [header height = 20.06 cm];
- 9. Repeat steps 6 and 7; [header height = 20.56 cm].

Open Office Writer. Bug 2 (3)

- **Bug 2.** inconsistență de setare:
 - Issue 127562 - Inconsistency on Header Height max. value when enabled first time (https://bz.apache.org/ooo/show_bug.cgi?id=127562);
- Replicate – reproducerea bug-ului;

Scenariu B

- 1. Open OpenOffice Writer;
- 2. Set the measurement unit to 'cm';
- 3. Create a new document;
- 4. keep the default Paper Format to Letter;
- 5. go to Header Tab;
- 6. Enable Header;
- 7. Set Header Height to 27.94 cm;
- 8. Change focus from Header Height; [header height = 18.65 cm]
- 9. Repeat steps 7 and 8; [header height = 19.15 cm]
- 10. go to Page tab again;
- 11. Set the Paper Format to A4;
- 12. go to Header Tab;
- 13. Set Header's Height to 29.7 cm;
- 14. Change focus from Header Height; [header height = 20.56 cm];
- 15. Repeat steps 12 and 13; [header height = 20.56 cm].

Open Office Writer. Bug 2 (4)

- **Bug 2.** inconsistență de setare:
 - Issue 127562 - Inconsistency on Header Height max. value when enabled first time (https://bz.apache.org/ooo/show_bug.cgi?id=127562);
- Isolate – izolarea bug-ului;
 - Scenariu A:
 - indică cea mai scurtă secvență de pași prin care bug-ul este evidențiat;
 - Scenariu B:
 - indică faptul că bug-ul apare indiferent de formatul de pagina ales inițial, i.e., A4 (scenariu A) sau Letter și apoi A4 (scenariu B);
 - indică faptul că bug-ul apare la prima activare a opțiunii Header Height;

Open Office Writer. Bug 3 (1)

- **Bug 3.** inconsistență de proiectare:
 - Issue 126371 – Disappearing Vertical Text button in Drawing Toolbar in Writer v.4.1.1 (https://bz.apache.org/ooo/show_bug.cgi?id=126371);

Se va investiga dacă OO Writer ar trebui să aibă un anumit comportament și NU dacă acest comportament este corect sau nu.

- Replicate – reproducerea bug-ului;
- Isolate – izolarea bug-ului;
- Maximize – maximizarea bug-ului;
- Generalize – generalizarea bug-ului;
- Externalize – externalizarea bug-ului;
- And say it clear and dispassionately – atitudine neutră la raportarea bug-ului;

Open Office Writer. Bug 3 (2)

- **Bug 3.** inconsistență de proiectare:
 - Issue 126371 – Disappearing Vertical Text button in Drawing Toolbar in Writer v.4.1.1 (https://bz.apache.org/ooo/show_bug.cgi?id=126371);
- Replicate – reproducerea bug-ului;
 - butonul Vertical Text:
 - disponibil când Asian Language Support din meniul Tools Options | Language Settings | Languages este activat ---> scrierea de sus în jos, pe verticală este posibilă;

Open Office Writer. Bug 3 (3)

- **Bug 3.** inconsistență de proiectare:
 - Issue 126371 – Disappearing Vertical Text button in Drawing Toolbar in Writer v.4.1.1 (https://bz.apache.org/ooo/show_bug.cgi?id=126371);
- Isolate – izolarea bug-ului;
 - Scenariu A
 - Asian Language Support nu este activat;
 - se încearcă adăugarea butonului Vertical Text din Drawing toolbar ---> butonul apare și dispare imediat;
 - dacă butonul este deja selectat, el nu apare în Drawing toolbar;
 - Scenariu B
 - Asian Language Support este activat;
 - se încearcă eliminarea butonului Vertical Text din Drawing toolbar ---> butonul dispare și apare imediat;
 - dacă butonul este deja deselectat, el apare totuși în Drawing toolbar;

Open Office Writer. Bug 3 (4)

- **Bug 3.** inconsistență de proiectare:
 - **Issue 126371 – Disappearing Vertical Text button in Drawing Toolbar in Writer v.4.1.1** (https://bz.apache.org/ooo/show_bug.cgi?id=126371);
- Externalize – externalizarea bug-ului;
 - impactul asupra utilizatorului:
 - calitatea produsului este afectată, utilizatorul devine frustrat pentru eșecul de adăugare sau eliminare a butonului Vertical Text din Drawing toolbar;
 - alte produse similare, e.g., Microsoft Office Word, nu constrâng utilizarea butonului Vertical Text doar în prezența unor setări specifice legate de limbă;
 - calitatea produsului este afectată prin forțarea utilizatorului să își aducă aminte să modifice setările de limbă pentru a realiza scrierea unui text vertical, după care să revină la setările de limbă inițiale;
 - dacă utilizatorul uită să revină la setările inițiale, acesta poate întâmpina dificultăți și la alte configurații care sunt afectate de anumite setări de limbă ---> utilizatorul nu beneficiază de caracteristicile produsului în maniera dorită;

CLASIFICAREA BUG-URILOR PE BAZA ATRIBUTELOR DE CALITATE

by Claudiu Draghia

Atribute ale calității soft și bug-uri asociate (1)

- **Funcționalitate - Functionality bug;**

- *Lasă impresia unei funcționări normale:*
 - *most human like shape;*
 - *curious with funny eyes;*
- *Impact vizual inițial pozitiv:*
 - *Dressed to impress.*



The functionality bug

source [[Draghia2023](#)]

Atribute ale calității soft și bug-uri asociate (2)

- **Testabilitate** - **CHAOS UI**;
 - *Impresionează și se deghizează:*
 - *Confident eyes;*
 - *Camouflaged;*
- Controllability;
- Heterogeneity;
- Automatability;
- Observability;
- Separation of concerns;
- Understandability;
- Isolateability.

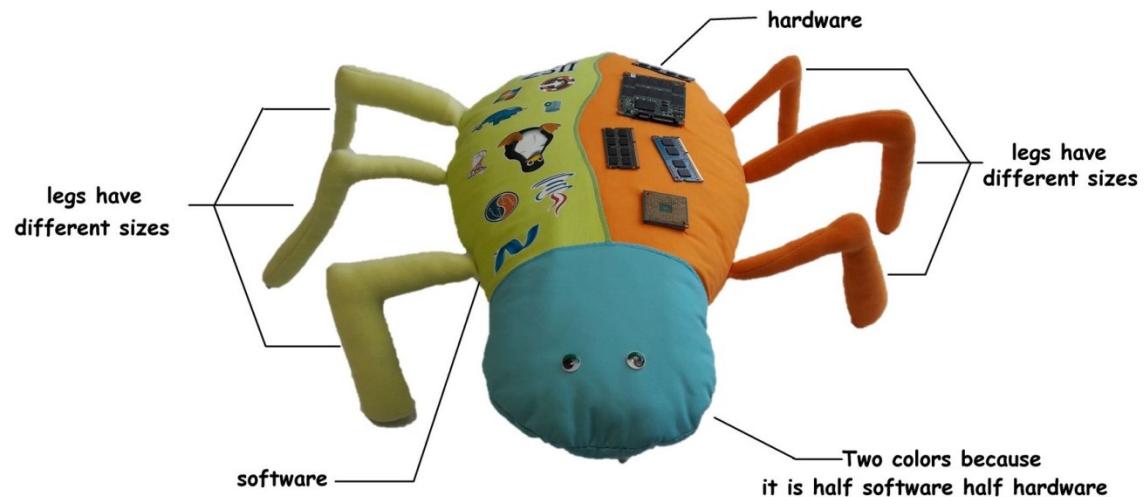


source [[Draghia2023](#)]

Atribute ale calității soft și bug-uri asociate (3)

- Performanță - **HaSo**;

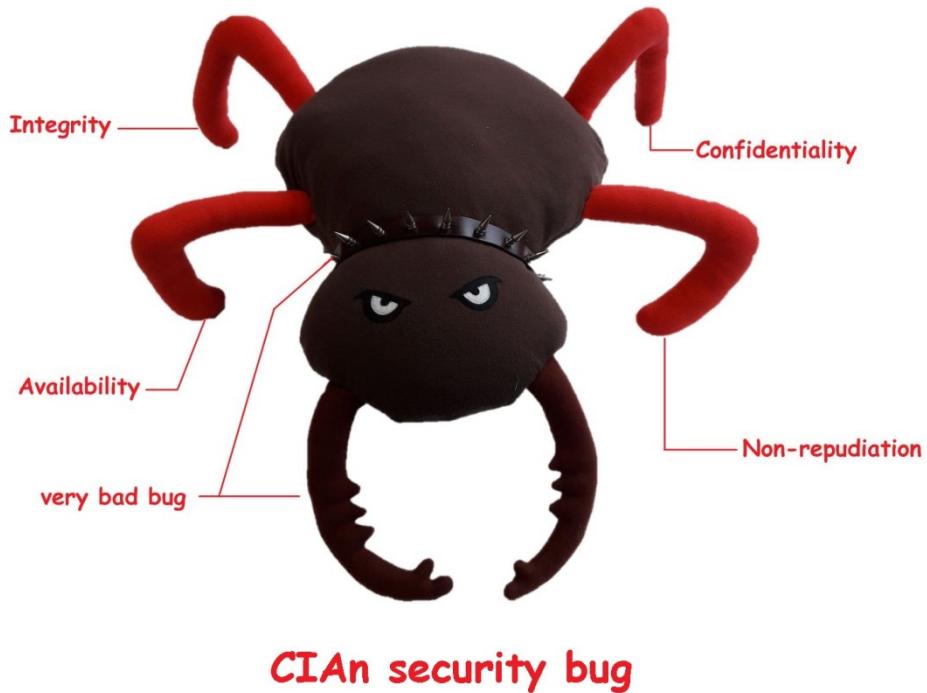
- *Bug hibrid:*
 - Hardware;
 - Software;



HaSo performance bug

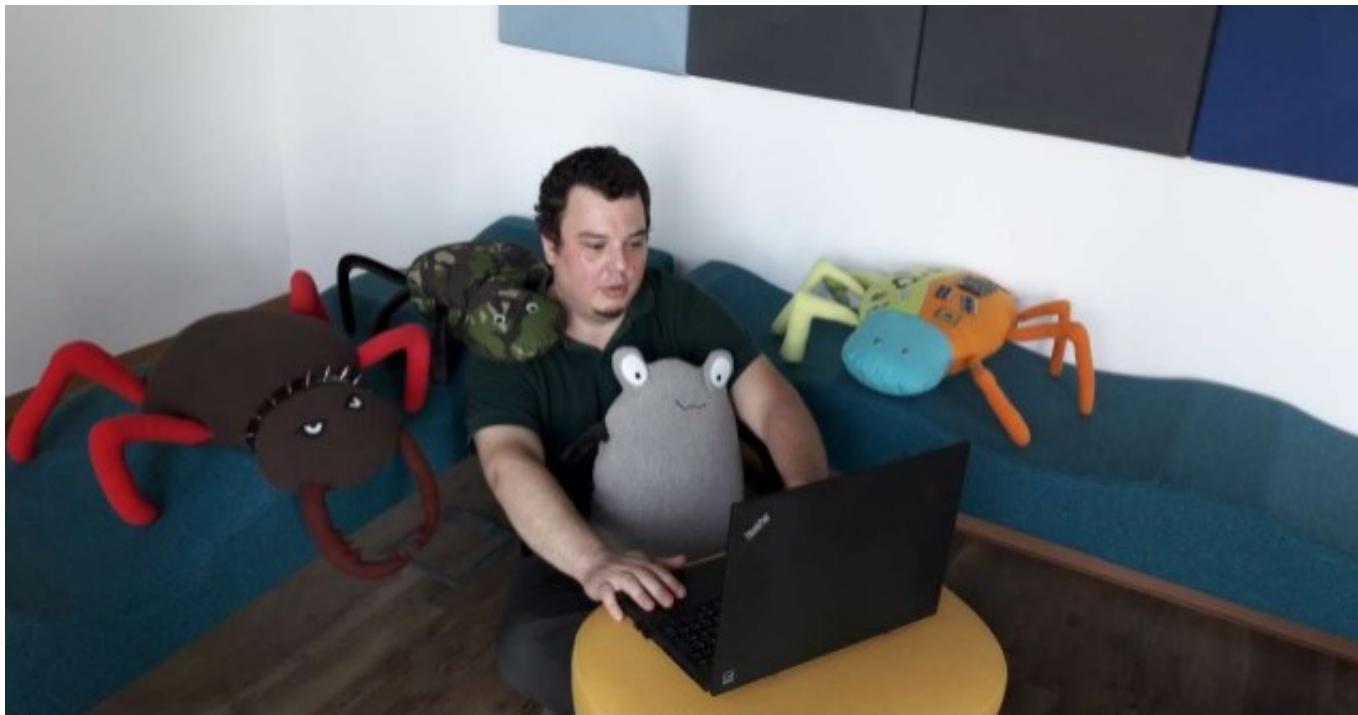
Atribute ale calității soft și bug-uri asociate (4)

- Securitate – **CIAn**;
 - *bug dezastroz*;
- Confidentiality;
- Integrity;
- Availability;
- non-repudiation.



source [[Draghia2023](#)]

Investigarea bug-urilor



source [[Draghia2023](#)]

Pentru examen...

- **Ciclul de viață al unui bug:**

- Cele două variante discutate la curs (simplu și detaliat);

- **RIMGEA**

- descrierea semnificației acronimului:

- replicate,
 - isolate,
 - maximize,
 - generalize,
 - externalize,
 - communicate it clear;

- Tipuri de bug-uri:

- Coding bug;
 - Design bug;
 - Coding bug vs design bug.

Seminar 05. A bug story (1)

- **A bug story**
 - **Termen: 10 Mai 2023, orele 12:00;**
 - **Echipe:** max. 3 studenți/echipă, i.e., echipe de forma (A, B, C), (A, B);
 - se acordă **2 puncte** de activitate pentru seminarul 5; dacă bug-ul este se regăsește printre primele 3 cele mai votate bug-uri, se acordă suplimentar **1 punct**.

Seminar 05. A bug story (2)

- **Pași de rezolvare:**
 - **Pas 1: alegerea unui produs soft;**
 - alegeti un produs soft, e.g.: MS Office Word, Google Mail, Google Sheets, OO Writer, proiectul dezvoltat la disciplina *Project Colectiv*, etc;
 - realizați o scurtă descriere a produsului soft (1-2 paragrafe) ce se va include într-un fișier.
 - **Pas 2: descrierea poveștii unui bug asociat softului ales;**
 - creați/imaginați/identificați un bug și realizați o descriere (text, 1-2 paragrafe) a acestuia, atribuindu-i diferite însușiri/caracteristici de manifestare asupra produsului (vezi însușirile bug-urilor descrise în secțiunea [Clasificarea bug-urilor](#) în **Curs09. Raportea bug-urilor**);
 - furnizați și o descriere grafică expresivă (desen, imagine) a bug-ului, alegeti un nume sugestiv;
 - descrierea/povestea bug-ului va fi inclusă în fișierul creat la **Pasul 1**;
 - în descrierea grafică se pot folosi nu doar insecte, ci și personaje (negative sau nu) din literatura universală, filme, desene animate, etc.

Seminar 05. A bug story (3)

- **Pași de rezolvare:**
 - **Pas 3: postarea poveștii bug-ului;**
 - fișierul finalizat la **Pasul 2** se postează în channel-ul **#BugStories**;
 - fișierul va conține și detaliile referitoare la componența echipei care a dezvoltat bug-ul (nume, grupă);
 - **Pas 4: votarea celei mai interesante povești/descrieri posteate;**
 - voturile se acordă individual, de către fiecare student care a participat sau nu la elaborarea poveștii unui bug;
 - un student poate vota mai multe bug-uri posteate;
 - **studenții se vor asigura că au acordat votul/rile lor până în Miercuri, 10 Mai 2023, orele 12:00;**
 - pentru acordarea punctului suplimentar se vor lua în considerare doar voturile exprimate în channel-ul **#BugStories** prin :**thumbsup**: ();
 - ca feedback, se pot adăuga și alte *reactions* la postarea poveștii unui bug, inclusiv mesaj scris în postarea asociată descrierii bug-ului.

Referințe bibliografice

- [ISTQBCertification2023] ISTQB Exam Certification, <http://istqbexamcertification.com/what-is-a-defect-life-cycle/>.
- [Patton2005] R. Patton, *Software Testing*, Sams Publishing, 2005.
- [BBST2008] Black-Box Software Testing (BBST), Bug Advocacy,
<http://www.testingeducation.org/BBST/bugadvocacy/BugAdvocacy2008.pdf>.
- [Altom2016] Levente Balint, *BLOG : RIMGEN, How Well Do you Advocate For Your Bugs?*,
<http://altom.training/blog/tag/rimgen/>.
- [Draghia2023] Claudiu Draghia, <http://bugs.brainforit.com/>.