Universitatea Politehnica București

Facultatea de Electronică, Telecomunicații și Tehnologia Informației

**Metode de testare în domeniul software**

**Aflorei Victor**

**Pâsoi Mihai**

**Jircu Andrei**

Grupa: 441A Coordonator științific:

Conf. Dr. Ing. **Ștefan Stăncescu**

**2014**

Contents

[1.Unit Testing - Testarea pe unitate (Aflorei Victor - 441A) 3](#_Toc378936435)

[1.1. Conceptul de testare pe unitate 3](#_Toc378936436)

[1.2. Testarea statică pe unitate 4](#_Toc378936437)

[1.3. Prevenirea de defecte 7](#_Toc378936438)

[1.4. Testarea dinamică pe unitate 8](#_Toc378936439)

[1.5. Debugging 10](#_Toc378936440)

[2. Testarea fluxului de control (Pâsoi Mihai - 441A) 12](#_Toc378936441)

[2.1. Introducere 12](#_Toc378936442)

[2.2. Graful fluxului de control (CFG) 13](#_Toc378936443)

[2.3. Rute în CFG.  Alegerea rutelor intr-un CFG 14](#_Toc378936444)

[2.4. Generarea  datelor de intrare folosite în testare 15](#_Toc378936445)

[3. Testare a fluxului de date (Jircu Andrei – 441A) 17](#_Toc378936446)

[3.1 Idee generală 17](#_Toc378936447)

[3.2. Anomalia fluxului de date 17](#_Toc378936448)

[3.3 Testarea dinamica a fluxului de date 19](#_Toc378936449)

[3.4. Grafuri ale fluxului de date 20](#_Toc378936450)

[4. Concluzii 21](#_Toc378936451)

[5. Bibliografie 22](#_Toc378936452)

# 1.Unit Testing - Testarea pe unitate (Aflorei Victor - 441A)

## 1.1. Conceptul de testare pe unitate

Testarea pe unitate se referă la testarea unitătilor dintr-un program în particular. Cu toate acestea, nu există un consens cu privire la definirea unei **unități**. Câteva exemple de unități frecvent înțelese sunt funcțiile, procedurile sau metodele. Chiar și o clasă într-un limbaj de programare orientat-obiect poate fi considerată ca o unitate de program.

Sintactic, o unitate de program este o bucată de cod, cum ar fi o funcție sau metodă dintr-o clasă, care este invocat din afara unității și care poate invoca alte unități de program. Mai mult decât atât, o unitate de program se presupune să implementeze o funcție bine definită furnizând un anumit nivel de abstractizare pentru implementarea funcțiilor de nivel superior. Funcția îndeplinită de către o unitate de program nu poate avea o asociere directă cu o funcție de la nivelul sistemului. Astfel, o unitate de program poate fi privită ca o bucată de cod ce implementează o funcție de nivel scăzut ("low"-level). În continuare, termenii **unitate** și **modul** îi vom folosi ca fiind echivalenți.

Acum, dat fiind faptul că o unitate de program implementează o funcție, este firesc să testăm această unitare înainte de a fi integrată cu alte unități. Astfel, o unitate de program este testată izolat, adică, într-o manieră independentă.

Se pot indentifica două motive care indică testarea unei unități în mod "stand-alone". În primul rând, erorile descoperite în timpul testării pot fi atribuite unei anumite unități și, așadar vor fi ușor de reparat. Mai mult decât atât, testarea pe unitate elimină dependențele cu alte unități de program. În al doilea rând, în timpul testării pe unitate este de dorit să se verifice că fiecare execuție distincta a unei unități de program produce rezultatul așteptat. În ceea ce privește detaliile de cod, o execuție distinctă se referă la o cale distinctă în unitate. Acest lucru necesită o selecție atentă a datelor de intrare pentru fiecare execuție distinctă. Un programator are acces direct la vectorul de intrare a unității executând o unitate de program independent. Accesul direct face mai ușoară executarea mai multor căi distincte în funcție de câte sunt dorite sau câte sunt posibile. În cazul în care mai multe unități sunt puse împreună pentru testare, un programator trebuie să genereze intrări de test în raport indirect cu vectorii de intrare a câtorva unități aflate sub test. Acestă relație indirectă, îngreunează controlul execuției de căi distincte într-o anumită unitate.

Testarea pe unitate are un scop limitat. Un programator va trebui să verifice dacă un cod funcționează corect sau nu prin efectuarea de teste la nivel de unitate. În mod intuitiv, un programator trebuie să testeze o unitate după cum urmează:

- Să execute fiecare linie de cod. Acest lucru este de dorit, deoarece programatorul trebuie să știe ce se întâmplă atunci când o linie de cod este executată. În absența unor astfel de observații de bază, surprizele de la o etapă ulterioară, poate fi costisitoare.

- Să execute fiecare predicat din unitate pentru a le evalua dacă sunt adevărate sau false separat.

- Să observe că unitatea îndeplinește funcția pentru care este destinată și să se asigure că acesta nu conține erori cunoscute. [1]

În ciuda testele de mai sus, nu există nici o garanție că o unitate testată în mod satisfăcător este corectă din punct de vedere al întregului sistem. Nu tot ceea ce este pertinent pentru o unitate poate fi testat în mod independent din cauza limitărilor acestei modalități de testare. Acest lucru înseamnă că într-o unitate de program nu pot fi găsite mai târziu erori, atunci când unitatea este integrată cu alte unități în fazele de testare de integrare si testare de sistem. Chiar dacă nu este posibil să se găsească toate erorile într-o unitate de program, este încă necesar să se asigure faptul că o unitate funcționează satisfăcător înainte de a fi folosită de către alte unități de program. Nu putem integrare o unitate de program cu erori cu alte unități din următoarele motive: (i) multe din testele ulterioare vor fi o risipă de resurse și (ii) a găsi principale cauze a eșecurilor într-un sistem integrat este o operație și mai consumatoare de resurse.

Testarea pe unitate este efectuată de către programatorul care scrie unitatea de program pentru că programatorul este familiarizat cu detaliile interne ale unității. Obiectivul programatorului este de a se asigura că unitatea funcționează cum era de așteptat. Din moment ce un programator ar trebui să construiască o unitate fără erori, un test de unitate este efectuat de către el/ea la început spre satisfacția lor și în conformitate cu cerințele altor programatori atunci când unitatea este integrată cu alte unități. Acest lucru înseamnă că toți programatorii sunt responsabili de calitatea muncii lor, care poate include atât cod nou cât și modificări la codul existent. Aici, ideea este de a împinge conceptul de calitate până la cel mai scăzut nivel al organizației și de a împuternici fiecare programator să fie responsabil pentru propria calitate. Prin urmare, este în interesul programatorului să ia măsuri preventive pentru a reduce numărul de defecte din cod. Defectele constatate în timpul testării pe unitate sunt interne, pentru grupul de dezvoltare de software și nu sunt raportate la nivel superior pentru a fi numărate în metricile de măsurare a calității. Codul sursă al unei unități nu este utilizat pentru interfațarea cu alți membri ai grupului, până când programatorul nu efectuează testarea unității.

Testarea pe unitate se desfășoară în două faze complementare:

- testarea pe unitate statică;

- testarea pe unitate dinamică.

În testarea statică pe unitate, un programator nu execută unitatea, în schimb, codul este examinat pentru toate comportamentele posibile ce ar putea apărea în timpul funcționării. Testarea statică pe unitate este, de asemenea, cunoscută sub numele de testare pe unitate bazată pe non-execuție, în timp ce testarea pe unitate dinamică este bazată pe execuție. În testarea pe unitate statică, codul fiecărei unități este validat în raport cu cerințele unității prin revizuirea codului. În timpul procesului de revizuire, potențialele probleme sunt identificate și rezolvate. [1]

## 1.2. Testarea statică pe unitate

Testarea statică pe unitate este realizată ca o parte dintr-o filosofie mai extinsă în care un produs software ar trebui să facă o fază de inspecție și de corecție la fiecare punct de referință din ciclul său de viață. La un anumit punct de referință, produsul nu trebuie să fie în forma sa finală. De exemplu, finalizarea procesului de codare este un reper, chiar dacă codarea tuturor unităților poate să nu conducă la produsul dorit. După codare, următorul punct de reper îl reprezintă testarea tuturor sau a unui număr substanțial de unități care formează componentele principale ale produsului. Astfel, înainte ca unitățile să fie testate individual prin executarea lor, acestea sunt supuse de obicei unui proces de control și corecție. Ideea din spatele procesului de revizuire este de a găsi defectele cât mai aproape de punctele lor de origine, astfel încât aceste defecte să fie eliminate cu mai puțin efort, iar produsul interimar să conțină mai puține defecte înainte ca următoarea sarcină să fie efectuată.

În cadrul testării statice pe unitate, codul este analizat prin aplicarea unor tehnici cunoscute sub numele de **inspecție** și **walkthrough**. Definiția de inspecție a fost inventată de Michael Fagan și cea de walkthrough de Edward Yourdon:

Inspecție: Este o revizuire pas-cu-pas per grup a unui produs de lucru, cu fiecare pas verificat pe baza unor criterii prestabilite.

Walkthrough: Este o revizuire în care autorul conduce echipa printr-o execuție manuală sau simulată a produsului, folosind scenarii predefinite. [1]

Indiferent dacă o revizuire se numește inspecție sau walkthrough, ea este o abordare sistematică pentru a examina codul sursă în detaliu. Scopul unui astfel de exercițiu este de a evalua calitatea software-ului în cauză, nu calitatea procesului utilizat pentru a dezvolta produsul. Review-urile despre acest tip sunt caracterizate de pregătirea semnificativă pe grupe de designeri si programatori cu diferite grade de interes în cadrul procesului de dezvoltare software. Examinarea codului poate fi consumatoare de timp. Mai mult decât atât, nici un proces de examinare nu este perfect. Examinatorii pot lua comenzi rapide, pot să nu înțeleagă adecvat produsul și poate accepta un produs care nu ar trebui să fie acceptat. Cu toate acestea, un proces de revizuire a codului bine conceput poate găsi defecte care pot fi ratate de testarea bazată pe execuție. Cheia succesului revizuirii codului este de a diviza și cuceri, adică, de a avea un examinator ce inspectează mici părți ale unității în mod independent, asigurându-se de următoarele: (i) nimic nu este trecut cu vederea și (ii) corectitudinea tuturor părților examinate ale modulului presupune corectitudinea întregului modul. Descompunerea revizuirii în etape distincte, trebuie să asigure că fiecare pas este destul de simplu astfel încât să poate fi realizat fără o cunoaștere aprofundată a celorlalți.

Obiectivul revizuirii de cod este de a revizui codul, nu de a evalua autorul codului. Un conflict poate apărea între autorul codului și revieweri iar acest lucru poate face întâlnirile neproductive. Prin urmare, revizuirea codului trebuie să fie planificată și gestionată într-un mod profesionist. Este nevoie de respect reciproc, deschidere, încredere, și schimburi de experiență în cadrul grupului. Directivele generale pentru efectuarea revizuirii codului constau în șase etape, definite în figura 1: promptitudine (readiness), pregătire (preparation), examinare (examination), reprelucrare (rework), validarea (validation), și ieșire (exit). Intrarea în etapa de pregătire este criteriul care trebuie îndeplinit înainte de începerea procesului de revizuire a codului, iar procesul produce două tipuri de documente, o cerere de schimbare (CR) și un raport.

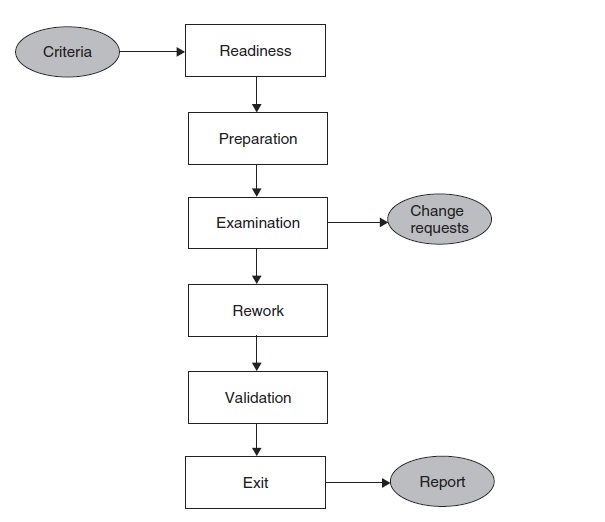


Figura 1. Pașii procesului de revizuire a codului

Aceste etape și documente sunt explicate în cele ce urmează.

Pasul 1: **Readiness -** Autorul unității se asigură că unitatea de testat este pregătită pentru revizuire. O unitate este declarată a fi gata în cazul în care îndeplinește următoarele criterii:

- Exhaustivitatea: Tot codul cu privire la unitatea care urmează să fie revizuită trebuie să fie disponibil. Acest lucru se datorează faptului că reviewerii au de gând să citească codul și să încerce să-l înțeleagă. Este neproductiv să revizuiască codul scris parțial sau codul care urmează să fie modificat în mod semnificativ de către programator.

- Minima Functionalitate: Codul trebuie compilat. Mai mult decât atât, codul trebuie să fi fost testat într-o anumită măsură pentru a se asigura că aceasta îndeplinește funcțiile sale de bază.

- Lizibilitate: Din moment ce revizuirea codului implică citirea efectivă a codului de către alți programatori, este esențial ca acest cod să fie foarte ușor de citit.

- Complexitatea: Nu este nevoie de a programa o întâlnire de grup pentru a revizui codul care poate fi ușor revizuit de către programator.

- Cerințele și Documentele de Proiectarea: Ultima versiune de specificații de proiectare aprobată sau alte descrieri adecvate de cerințele programului trebuie să fie disponibile.

Pasul 2: **Preparation -** Înainte de întâlnire, fiecare reviewer revede cu grijă pachetul de lucru. Este de așteptat ca reviewerii să citeasă codul și să înțeleagă organizarea și funcționarea acesteia, înainte de reuniunea de revizuire.

Pasul 3: **Examination -** Procesul de examinare este format din următoarele activități:

- Autorul face o prezentare a logicii procedurale utilizate în cod, a căilor care denotă calcule majore și a dependenței unității analizate de alte unități.

- Prezentatorul citește codul linie cu linie. Reviewerii pot ridica probleme în cazul în care codul este considerat a avea defecte. Cu toate acestea, problemele nu sunt rezolvate în cadrul reuniunii. Reviewerii pot face sugestii generale cu privire la modul de a repara defectele, dar este la latitudinea autorului codului dacă ia sau nu măsuri după ce se termină ședința.

- Recordkeeper-ul documentează solicitările de modificare și sugestiile de rezolvare a problemelor, dacă există.

- Moderatorul se asigură că întâlnirea rămâne concentrată pe procesul de revizuire. Moderatorul se asigură că întâlnirea face progrese la o anumită rată, astfel ca obiectivul reuniunii să fie atins.

- La sfârșitul întâlnirii, se ia o decizie cu privire la: dacă se va apela sau nu la o altă întâlnire pentru a revizui în continuare codul.

Pasul 4: **Rework -** La sfârșitul întâlnirii, recordkeeper-ul produce un rezumat al întâlnirii, care include următoarele informații:

- O listă a tuturor CR-urilor, datele de care acestea vor fi fixe, precum și numele persoanelor responsabile pentru validarea CR-urilor;

- O listă de oportunități de îmbunătățire;

- Procesul-verbal al ședinței (opțional).

Pasul 5: **Validation** - CR-urile sunt validate în mod independent de către moderator sau de o altă persoană desemnată în acest scop. Procesul de validare implică verificarea codul modificat cum este documentat în CR și asigurarea că îmbunătățirile propuse au fost implementate corect. Versiunea definitivă cu privire la rezultatul reuniunii de revizuire este distribuit tuturor membrilor grupului.

Pasul 6: **Exit** -Rezumând procesul de revizuire, se spune că este complet dacă toate au fost luate următoarele măsuri:

- Fiecare linie de cod din unitate a fost inspectată.

- În cazul în care prea multe defecte au fost găsite într-un modul, modulul este din nou revizuit după corecțiile aplicate de către autor.

- Autorul și reviewerii ajunge la un consens care, atunci când au fost aplicate corecții codul va fi potențial liber de defecte.

- Toate CR-urile sunt documentate și validate de către moderator sau de altcineva. Acțiunile de follow-up ale autorului sunt documentate.

- Un raport al reuniunii, inclusiv CR-urile sunt distribuite tuturor membrilor grupului de revizuire.

## 1.3. Prevenirea de defecte

Este în interesul programatorilor în special și în interesul companiei în general, să reducă numărul CR-urilor generate în timpul revizuirii codului. Acest lucru se datorează faptului că CR-urile sunt indicii de potențiale probleme în cod, iar aceste probleme trebuie să fie rezolvate înainte ca diferite unități de program să fie integrate. Abordarea CR-urilor necesită mai multe resurse și pot duce la amânarea proiectului. Prin urmare, este esențial să se adopte conceptul de prevenire a defectelor în timpul dezvoltării codului. În practică, defectele sunt introduse din greșeală de către programatori. Aceste accidente pot fi reduse prin luarea de măsuri preventive. Este utilă dezvoltarea unui set de directive pentru a construi cod pentru minimizarea defectelor așa cum se explică în cele ce urmează. Aceste directive se concentreză pe includerea de mecanisme adecvate în cod:

- Construirea de instrumente de diagnosticare interne, de asemenea, cunoscut sub numele de *instrumentation code*. Instrumentation code sunt utile în furnizarea de informații despre stările interne ale unităților. Aceste coduri permite programatorilor să realizeze mecanisme de urmărire built-in și mecanisme de depistare. Instrumentation code joacă un rol pasiv în testarea pe unitate dinamică. Rolul este pasiv în sensul respectării și înregistrării comportamentului intern, fără a testa în mod activ o unitate.

- Utilizarea controalelor standard pentru a detecta eventualele cazuri de condiții de eroare. Câteva exemple de detectare a erorilor din cod sunt împărțirile la zero.

- Asigurați-vă că există cod pentru toate valorile returnate, dintre care unele pot fi invalid. Acțiuni adecvate de urmărire trebuie luate pentru a se ocupa de valorile returnate invalide.

- Asigurați-vă că câmpurile de date de tip conter și buffer overflow și underflow sunt tratate în mod corespunzător.

- Oferă mesaje de eroare și texte de ajutor de la o sursă comună, astfel încât schimbările în text nu provoacă inconsecvență. Mesaje de eroare bune identifică cauzele profunde ale problemelor și pot ajuta utilizatorii în rezolvarea problemelor.

- Validarea datelor de intrare, cum ar fi argumentele, oferite funcțiilor.

- Folosirea de afirmații pentru a detecta condițiile imposibile, utilizări nedefinite de date și de comportamente nedorite ale programului. O afirmație este o declarație Boolean care nu ar trebui să fie niciodată falsă sau pot fi falsă numai în cazul în care a avut loc o eroare. Cu alte cuvinte, o afirmație este o verificare pe o condiție care se presupune a fi adevărată, dar aceasta poate provoca o problemă în cazul în care aceasta nu este adevărată.

- Lasarea afirmațiilor în cod. Le puteți dezactiva în versiunea lansată de cod cu scopul de a îmbunătăți performanța operațională a sistemului.

- Documentarea pe deplin a afirmațiilor care par a fi neclare.

- În sistemele care implică mesaje trecătoare, managementul buffer-ului este o activitate internă importantă. Mesajele primite sunt stocate într-un buffer deja alocat. Este utilă generarea un eveniment care indică disponibilitatea redusă a buffer-ului înainte ca sistemul rămâne fără buffer. Dezvoltarea unei rutine care să monitorize în permanență disponibilitatea buffer-ului după fiecare utilizare, calcularea spațiului rămas disponibil în buffer și apelarea unei rutine de tratare a erorilor în cazul în care cantitatea de spațiu din buffer-ul disponibil este prea mică.

- Dezvoltarea unei rutine de timer care numără descrescător de la un timp prestabilit până când ajunge zero sau este resetată. În cazul în care software-ul este prins într-o buclă infinită, timer-ul va expira și o excepție de tratare de rutină poate fi invocată.

- Includerea unui contor de buclă în fiecare buclă. Dacă bucla nu este executată niciodată de un număr de ori mai mic decât numărul minim posibil sau este execută de un număr de ori mai mare decât numărul maxim posibil se invoca o rutină de tratare de excepție.

- Definirea unei variabile pentru a indica ramura de decizie logică care va fi luată. Verificarea aceastei valoari după ce decizia a fost făcută și ramura din dreapta a fost presupusă luată.

## 1.4. Testarea dinamică pe unitate

Testarea pe unitate bazată pe execuție este cunoscută sub numele de la unitate de testare tetatre dinamică pe unitate. În cadrul aceastei modalități de testare, o unitate de program este de fapt executat individual așa cum o înțelegem în mod obișnuit. Cu toate acestea, această execuție diferă de execuție obișnuită în modul următor:

1. O unitate aflată sub test este scoasă din mediul său actual de execuție.

2. Mediul actual de execuție este emulat prin scrierea de cod, astfel încât unitatea și mediu emulat pot fi compilate împreună.

3. Agregatul compilat de mai sus este executat cu intrările selectate. Rezultatele unei astfel de execuții sunt colectate într-o varietate de moduri, cum ar fi observarea directă pe un ecran, logarea pe fișiere și software-ul de instrumentație a codului pentru a descoperi comportamentul din timpul rulării. Rezultatul este comparat cu rezultatul așteptat. Orice diferență între rezultatul actual și rezultatul așteaptat implică un eșec defectul este în cod.[1]

Un mediu pentru testarea dinamică pe unitate este creat prin emularea contextului unității de testat, așa cum se arată în figura 2. Contextul unui test pe unitate constă din două părți: (i) un apelant al unității și (ii) toate unitățile apelate de unitate. Mediul unei unități este emulat pentru că unitatea va fi testată în mod independent și mediul emulat trebuie să fie unul simplu astfel încât orice defect constatat în urma executării unității să poată fi atribuit numai unității aflate sub test. Unitatea apelantă este numită pilot de teste (*test driver*) și toate emulațiile unităților apelate de către unitatea aflată sub test sunt numite cioturi (*stubs*). Test driver-ul și cioturile sunt împreună numite *scaffolding*.

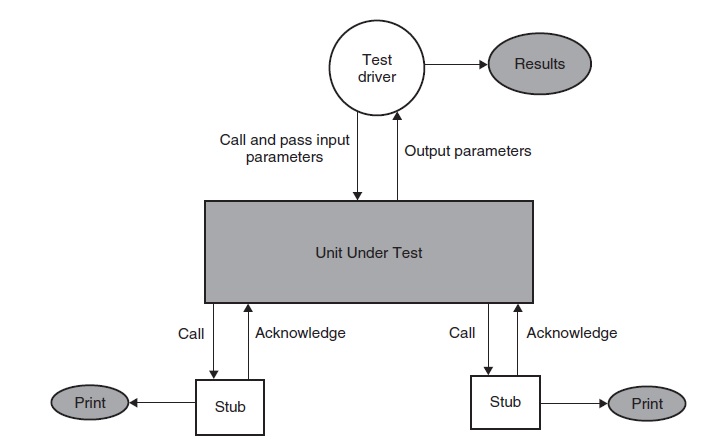


Figura 2. Mediul unui test dinamic

Funcțiile unui test driver și ale unui ciot sunt:

- Test driver-ul: Un pilot de teste este un program care invocă unitatea aflată sub testare. Unitatea aflată sub testare execută cu valorile de intrare primite de la pilot și la sfărșit returnează o valoare pilotului. Pilotul compară rezultatul real, care este, valoarea reală returnată de către unitatea aflată sub testat, cu rezultatul așteptat de la unitate și raportează rezultatul testului urmat. Pilotul de teste funcționează ca unitate principală (main) în procesul de execuție. Pilotul facilitează nu numai compilarea, dar, de asemenea, oferă date de intrare în formatul așteptat unității aflate sub test.

- Cioturile: Un ciot este un "subprogram fals" (*dummy subprogram*), care înlocuiește o unitate care este apelată de către unitatea aflată sunt test. Cioturi înlocuiesc unitățile apelate de către unitatea de testat. Un ciot efectuează două sarcini. În primul rând, aceasta arată o dovadă că ciotul a fost, întradevăr, apelat. Astfel de dovezi pot fi afișate doar prin imprimarea unui mesaj. În al doilea rând, ciotul returnează o valoare precalculată apelantului astfel încât unitatea aflată sub test își poate continua execuția.

Pilotul și cioturile nu sunt aruncate după ce testul pe unitate este finalizat. În schimb, acestea sunt refolosite în viitor în testarea de regresie a unității în cazul în care există o astfel de necesitate. Pentru fiecare unitate, ar trebui să existe un pilot de testare dedicat și mai multe cioturi după cum este necesar. În cazul în care doar un singur pilot de teste este dezvoltat pentru a testa mai multe unități, pilotul va fi unul complicat. Orice modificare a pilotului pentru a se acomoda cu schimbările de la una dintre unitățile de testat pot avea efecte secundare în testarea celelalte unități. În mod similar, pilotul de testare nu ar trebui să depindă de fișierele de date de intrare externe, dar, în schimb, ar trebui să aibă propriul set separat de date de intrare. Abordarea separată a fișierelor de date de intrare devine o alegere foarte convingătoare pentru cantități mari de date de intrare pentru test. De exemplu, în cazul în care sute de elemente de date de intrare de test sunt necesare pentru a testa mai mult de o unitate, atunci este mai bine să se creeze un fișier separat de date de intrare de test decât pentru a include același set de date de intrare de test în fiecare pilot conceput pentru a testa respectiva unitate.

Pilotul de test ar trebui să aibă capacitatea de a determina automat succesul sau eșecul unității supusă încercării pentru fiecare date de intrare de test. Dacă este cazul, pilotul trebuie să verifice, de asemenea, pierderile de memorie și probleme în alocarea și golirea memoriei. În cazul în care modulul deschide și închide fișiere, pilotul ar trebui să verifice că aceste fișiere sunt lăsate în starea așteptată (deschise sau închise) după fiecare test. Pilotul poate fi proiectat pentru a verifica valorile date ale variabilelor interne care nu ar fi în mod normal disponibile pentru verificare la integrare.

Pilotul și cioturile sunt strâns cuplate cu unitatea de testat și trebuie să însoțească unitatea de-a lungul ciclului său de viață. De fiecare dată când o unitate este modificată, programatorul ar trebui să verifice dacă trebuie sau nu să modifice pilotul și cioturile corespunzătoare. Ori de câte ori un nou defect este detectat în unitate, pilotul corespunzătoar ar trebui să fie actualizat cu un nou set de date de intrare pentru a detecta defectul și defectele similare în viitor. În cazul în care este de așteptat ca unitatea să ruleze pe diferite platforme, pilotul de testare și cioturile ar trebui să fie, de asemenea, construite pentru a testa unitatea pe platforme noi. În cele din urmă, pilotul de test și cioturile ar trebui să fie revizuite, referințe încrucișate cu unitatea pentru care acestea sunt scrise și verificate în sistemul de control al versiunii ca un produs în același timp cu unitatea.[1]

## 1.5. Debugging

Programatorul, după un program eșuat, identifică greșeala corespunzătoare și o repară. Procesul de determinare a cauzei unui eșec este cunoscut sub numele de **depanare**. Depanarea se produce ca urmare a unui test ce dezvăluie un eșec. *Myers* a propus trei abordări de depanare în cartea sa *Arta de Testare Software:*[2]

**1. Brute Force**: Abordarea brute-force pentru depanare este preferat de mulți programatori. Aici, filozofia utilizată este "let the computer to find the error".

Declarații de imprimare sunt împrăștiate prin tot codul sursă. Aceste declarații de imprimare oferă o urmă brută a modului în care codul sursă a fost executat. Disponibilitatea unui bun instrument de depanare face ca aceste declarații de imprimare să fie redundante. Un debugger dinamic permite inginerului software permite navigarea pas cu pas prin cod, observarea căror căi s-au executat și observarea modului în care valorile variabilelor se schimbă în timpul execuției controlate. Un bun instrument permite programatorului să atribuie valori câtorva variabile și să navigheze pas cu pas prin cod. Instrumentation code poate fi construit în codul sursă pentru a detecta problemele și pentru a conecta valorile intermediare ale variabilelor pentru diagnosticarea problemei. Se poate folosi o imagine de memorie după ce o defecțiune a avut loc pentru a înțelege starea finală a codului depanat. Jurnalul și imaginea de memorie sunt revizuite pentru a înțelege ceea ce sa întâmplat și cum s-a produs eșecul.

**2. Cause Elimination**: Abordarea eliminarea cauzei poate fi bine descrisă ca un proces care implică inducție și deducție. În partea de inducție, în primul rând, toate datele pertinente referitoare la eșec sunt colectate, cum ar fi ceea ce s-a întâmplat și care sunt simptomele. Apoi, datele colectate sunt organizate în ceea ce privește comportamentul și simptomele iar relația lor este studiată pentru a găsi un model pentru a izola cauzele. O ipoteză cauză este conceput, iar datele de mai sus sunt folosite pentru a dovedi sau infirma ipoteza. În partea de deducere, o listă a tuturor cauzelor posibile este dezvoltată în ordinea probabilităților și testele sunt efectuate pentru a elimina fiecare cauză în scopul scăderii probabilității. Dacă testele inițiale indică faptul că o anumită ipoteză promite, datele de test sunt rafinate în încercarea de a izola problema cum este necesar.

**3. Backtracking**: În această abordare, programatorul pornește din punct în care o greșeașă în cod a fost observată și urmărește înapoi execuția până în punctul în care a avut loc. Această tehnică este frecvent utilizat de către programatori iar acest lucru este util în programele mici. Cu toate acestea, probabilitatea de urmărire înapoi la greșeală scade pe măsură ce crește dimensiunea programului, deoarece numărul de potențiale căi înapoi poate deveni prea mare.

De multe ori, inginerii software observă alte probleme nedetectate anterior în timp ce depanează și corecteză o greșeală. Aceste defecte nou descoperite nu ar trebui rezolvate împreună cu cu defectul în cauză. Acest lucru se datorează faptului că inginerul software nu poate înțelege pe deplin partea din cod responsabilă cu noua greșeală. Cea mai bună metodă de a rezolva o astfel de situație este de a depune un CR. Un nou CR oferă programatorului o oportunitate de a discuta problema cu alți membri ai echipei, cu alți arhitecți software și poate obține aprobarea lor pentru o sugestie făcută de el. Odată ce CR-ul este aprobat, inginerul software trebuie să completeze un nou defect în baza de date de urmărire de defecte și poate începe rezolvarea. Acest proces este greoi și întrerupe procesul de depanare, dar este util pentru proiecte foarte critice. Cu toate acestea, programatorii de multe ori nu respectă acest lucru din cauza lipsei unei proceduri pe care să o pună în aplicare.

# 2. Testarea fluxului de control (Pâsoi Mihai - 441A)

## 2.1. Introducere

Două tipuri de declarații de bază într-o unitate de program sunt declarațiile de atribuire și declarațiile condiționale. O declarație de atribuire este reprezentată în mod explicit, prin utilizarea unui symbol de atribuire, "=", precumi x = 2 \* y, unde x și y sunt variabile. Condițiile programului sunt la baza declarațiilor condiționale, cum ar fi if(), bucla for(), bucla while(), şi goto. Ca exemplu, în cazul în care (x! = y), testăm inegalitatea între x și y. În absența unor declarații condiționale, instrucțiunile de program sunt executate în ordinea în care apar. Ideea de executare succesivă a instrucțiunilor dă naștere la conceptul de flux de control într-o unitate de program. Declarațiile condiţionale modifica fluxul de control secvențial şi implicit într-o unitate de program. De fapt, chiar și un număr mic de declarații condiționale poate duce la un structură flux de control complexă într-un program.

Apelurile de funcții sunt mecanisme pentru a asigura abstracție în proiectarea programului. Un apel la o funcție de program duce la controlul ce întră în funcția de apelare. În mod similar, atunci când funcția execută declarația sa de revenire, spunem că iese controlul din funcție. Deși o funcție poate avea mai multe declarații de returnare, pentru simplitate, se poate restructura funcția pentru a avea exact o revenire. O unitate de program poate fi privit ca având un punct de intrare bine definit și un punct de ieșire bine definit. Executarea unei secvențe de instrucțiuni de la punctul de intrare la punctul de ieșire a unei unităţi de program se numește o cale de program. Pot fi un număr de căi mare, chiar infinit într -o unitate de program. Fiecare cale de program poate fi caracterizată de o intrareși o ieșire de asteptat . O valoare specifică de intrare determină o cale de program specifică să fie executată ,este de așteptat ca traseul programului să realizeye calculele dorite, producând astfel valoarea așteptată de ieșire. Prin urmare , poate părea firesc să se execute cât mai multe căi de program posibile. Simpla executare a unui număr mare de căi, la un cost mai mare, poate să nu fie eficient în identificarea defectelor. În mod ideal, trebuie să se depună eforturi pentru a executa mai puține căi pentru o mai bună eficiență.

Conceptele de flux de control asupra programelor pentru calculator, căi de program, și testarea fluxului de control sunt studiate de mai multe decenii. Instrumentele sunt în curs de dezvoltate pentru a susține testarea fluxului de control. Astfel de instrumente identifică trasee de la o unitate de program bazat pe un criteriu definit de utilizator şi generează intrarea corespunzătoare pentru a executa calea selectată și a genera resturile de programe și drivere pentru a executa testul. Testarea fluxului de control este un fel de testare structurală , care este efectuată de către programatori pentru a testa codul scris de ei. Conceptul este aplicat la unități mici de cod, cum ar fi o funcție. Cazurile de testare pentru fluxul de control al testării sunt derivate din codul sursă, cum ar fi o unitate de program ( de exemplu , o funcție sau metodă ), decât din întregul program.[1]

Structural, o cale este o secvență de instrucțiuni într-o unitate de program, în timp ce, semantic, aceasta este o instanţă de execuție a unității. Pentru un anumit set de date de intrare, unitatea de program executa o anumită cale. Pentru un alt set de date de intrare, unitatea poate executa o cale diferită. Ideea de bază în testarea fluxului de control este de a selecta în mod corespunzător câteva căi într-o unitate de program și să observe dacă traseele selectate produc rezultatul scontat sau nu. Prin executarea câtorva căi într-o unitate de program, programatorul încearcă să evalueze comportamentul întregii unități de program.

## 2.2. Graful fluxului de control (CFG)

Un CFG este o reprezentare grafică a unei unități de program. Se utilizează trei simboluri pentru a construi un CFG, așa cum se arată în figura 3. Un dreptunghi reprezintă un calcul secvențial. Un calcul secvențial maximal poate fi reprezentat fie de un singur dreptunghi sau de mai multe dreptunghiuri, fiecare corespunzând la o declarație în codul sursă.[1]

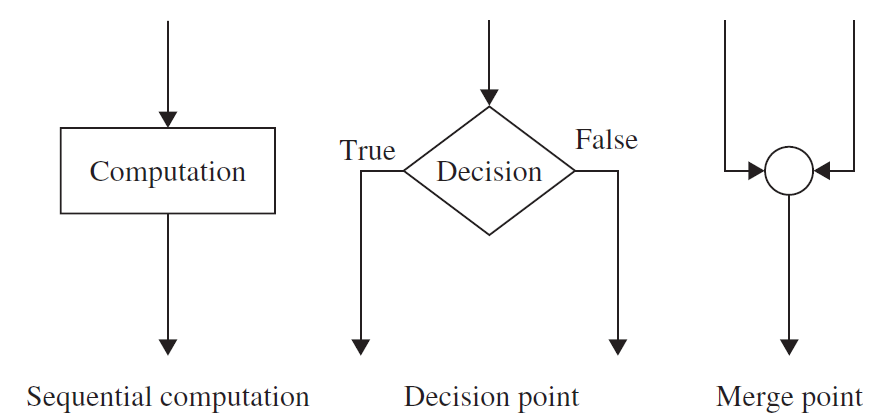


Figura 3. Simboluri într-un CFG

Etichetăm fiecare calcul și caseta de decizie, cu un întreg unic . Cele două ramuri ale unei cutii de decizie sunt etichetate cu T și F pentru a reprezenta adevărat și fals evaluări, respectiv, ale stării în caseta. Nu vom eticheta un nod de îmbinare, deoarece se pot identifica cu ușurință căile într-un CFG chiar fără a avea în mod explicit nodurile de îmbinare. Mai mult decât atât, dacă nu se menționează nodurile de îmbinare într-o cale, se va face o descriere a căii mai scurtă.

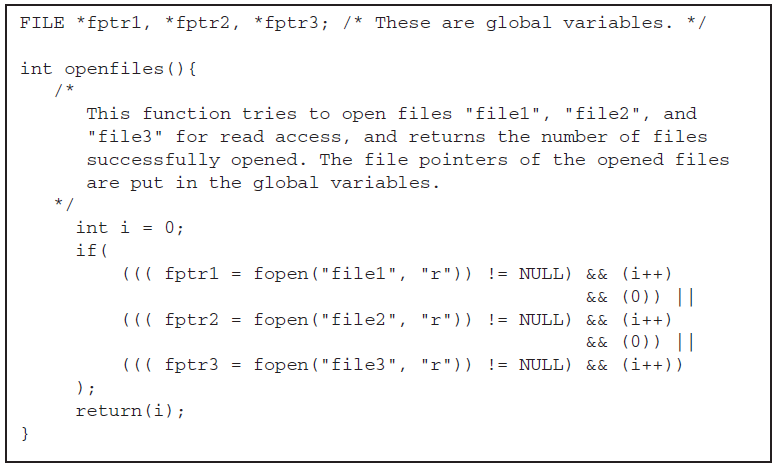


Figura 4. Funcție de deschidere a trei fișiere

Considerăm funcția openfiles prezentată în figura 4 pentru a ilustra procesul de elaborare a CFG. Funcția are trei declarații : o declarație de atribuire int i = 0; , o declarație de condiționalitate, if( ) , și o declarație return( i). Cititorul poate constata că, indiferent de evaluarea if( ), funcția efectuează aceeași acțiune, și anume, nul. În figura 5 , se va vom arata o reprezentare la nivel înalt a a fluxului de control în openfiles ( ) cu trei noduri numerotate 1 , 2 , și 3 . Fluxul grafic arată doar două căi în openfiles ( ). La o examinare mai atentă a părții de stare if(), declarația arată că există nu numai operatori booleeni și relaționali în partea de stare, dar, de asemenea, instructiuni de atribuire.

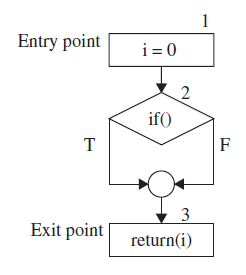


Figura 5. Reprezentare CFG de nivel înalt pentru openfiles()

## 2.3. Rute în CFG.  Alegerea rutelor intr-un CFG

Presupunem că un grafic de flux de control are exact un nod de intrare și exact un nod de ieșire pentru comoditatea de discuție. Fiecare nod este etichetat cu o valoare unică întreagă. De asemenea, cele două ramuri ale unui nod decizie sunt etichetate în mod corespunzător cu adevărat (T) sau fals (F). Suntem interesati in a identifica căi de intrare-ieșire într-un CFG. O cale este reprezentată ca o succesiune de noduri de calcul și de decizie de la nodul de intrare la nodul de ieșire. De asemenea, se specifică dacă controlul iese printr-un nod decizie, prin intermediul ramurilor sale de adevărat sau fals în timp ce-l include în cale.

**Criteriul de acoperire al tututor căilor**

Dacă sunt selectate toate căile într-un CFG , atunci se pot detecta toate defectele, cu excepția celor cauzate unor erori de cale lipsă. Cu toate acestea , un program poate conține un număr mare de căi, sau chiar un număr infinit de căi. Funcția mica openfiles() prezentată în figura 4 conține mai mult de 25 de căi. Unul nu știe dacă sau nu este o cale fezabilă în momentul selectării căii, deși numai opt dintre acestea sunt fezabile. Dacă unul selectează toate căile posibile într-un program, atunci spunem că criteriul de selecție cale al tuturor căilor a fost îndeplinită.

Să luăm exemplul openfiles(). Această funcție încearcă să deschidă trei fișiere. Funcția returnează un întreg reprezentând numărul de fișiere care le-a deschis cu succes. Un fișier este declarat a fi cu success deschis cu acces "read", dacă fișierul există. Existența unui fișier este fie "da" sau " nu". Astfel, domeniul de intrare al funcției este format din opt combinații de existență a trei fișiere. [1]

**Criteriul acoperirii declaratiilor**

Acoperirea se referă la executarea declarațiilor individuale de program și observarea rezultatului. Noi spunem că 100 % sunt acoperite în cazul în care toate declarațiile au fost executate cel puțin o dată . O acoperire completă este cel mai slab criteriu în programul de testare. Orice suita de teste care realizează acoperire mai putina pentru software-ul nou este considerat a fi inacceptabil .

**Criteriul acoperirii ramurilor**

Din punct de vedere sintactic, o filială este o margine de ieșire de la un nod. Toate nodurile dreptunghiului au cel mult o ramură de ieșire (margine).Nodul de ieșire a unui CFG nu are ramură de ieșire. Toate nodurile de tip diamant au două ramuri de ieșire. Acoperind o ramură înseamnă selectarea unui traseu care include ramura. O acoperire completă de acest tip înseamnă selectarea unui număr de căi astfel încât fiecare ramură să fie inclusă în cel puțin o cale. [2]

## 2.4. Generarea  datelor de intrare folosite în testare

În secțiunea anterioară am explicat conceptul de criterii de selecție cale pentru a acoperi anumite aspecte ale unui program cu un set de trasee. Aspectele de program pe care le-am considerat au fost toate declarațiile, adevărate și false de fiecare condiție, și combinații de condiții care afectează execuția unui traseu. Acum, după ce a identificată o cale, întrebarea este cum se vor selecta valori de intrare astfel încât atunci când programul este executat cu intrările selectate, căile alese fi executate. Cu alte cuvinte, trebuie să identificăm intrări pentru a forța executarea căilor. Se vor define cativa termeni si da un exemplu de generatoare de intrări de test pentru un traseu selectat.

1. Vector de intrare: Un vector de intrare este o colecție a tuturor entităților de date citite de rutină ale căror valori trebuie să fie fixate înainte de a intra în rutină. Membrii unui vector de intrare unei rutine pot lua diferite forme, enumerate mai jos :

• Argumente de intrare la o rutină

• Variabile globale și constante

• Fișiere

• Conținut de registre în programarea în limbaj de asamblare

• Conexiuni de rețea

• Cronometre

Un fișier este un element de intrare complex. Într-un singur caz, simpla existență a unui fișier poate fi considerată ca intrare, în timp ce într-un alt caz, conținutul fișierului este considerat a fi factor de producție. Astfel, ideea unui vector de intrare este mai generală decât conceptul de argumente de intrare ale unei funcții.

2. Predicat: Un predicat este o funcție logică evaluată la un punct de decizie.

3. Cale a predicatului: Un predicat cale este un set de predicate asociate cu o cale.

4. Interpretare predicat: Calea predicatului este compusă din elemente ale vectorului de intrare, un vector de variabile locale și constant. Variabilele locale nu sunt vizibile în afara unei funcții, dar sunt utilizate pentru a:

• deține rezultate intermediare,

• indica elementele de matrice, și

• controla iterațiide tip bucla.

5 . Expresie a căii predicat : O cale predicat interpretată se numește o expresie de cale predicat. O expresie cale predicat are următoarele proprietăți :

• Este lipsită de variabile locale și este compusă doar din elemente ale vectorului de intrare și, eventual, un vector de constante.

• Este un set de constrângeri construite din elemente ale vectorului de intrare

și, eventual, un vector de constante .

• Este o cale ce forțează valori de intrare ce pot fi generate prin rezolvarea setului de constrângeri într-o expresie cale predicat.

• În cazul în care setul de constrângeri nu poate fi rezolvat, nu există nici o intrare care poate determina calea aleasă pentru execuție . Cu alte cuvinte, calea aleasă este imposibilă.

• O cale nefezabilă nu implică faptul că una sau mai multe componente ale unui traseu expresie predicat sunt nesatisfiabile. Aceasta înseamnă că combinația totală de toate componentele dintr-o expresie cale predicat este nesatisfiabilă.[1]

# 3. Testare a fluxului de date (Jircu Andrei – 441A)

## 3.1 Idee generală

O unitate de program, cum ar fi o functie, accepta valori de intrare, executa calcule, in timp ce atribuie valori noi pentru variabile locale si globale, si in cele din urma produce valori de iesire. Prin urmare, ne putem imagina un fel de “flux” de valori de date intre variabile, de-a lungul unei cai de executie a programului. O valoare calculata intr-un anumit pas al executiei programului este de asteptat sa fie utilizata intr-o etapa ulterioara a programului.

Exista doua motive pentru testarea fluxului de date : in primul rand, o locatie de memorie corespunzatoare unei variabile de program este accesata intr-un mod dorit. De exemplu, o locatie de memorie poate sa nu fie citita inainte de a scrie in aceasta locatie. In al doilea rand este de dorit sa se verifice corectitudinea valorii de date generata de o variabila – acest lucru se observa prin verificarea tuturor valorilor ce produc rezultate dorite.

Ideea de baza de mai sus cu privire la testarea fluxului de date ne spune ca un programator poate efectua o serie de teste pe valori de date, care sunt cunoscute sub numele de flux de date. Testarea fluxului de date poate fi realizata la nivelul a doua concepte: testarea fluxului de date statice si testarea fluxului de date dinamice. Dupa cum sugereaza si numele, testarea fluxului de date statice se efectueaza prin analizarea codului sursa si nu implica executarea de cod sursa. Testarea fluxului de date statice este efectuat pentru a descoperi potentiale erori in program.

Erorile posibile din program sunt cunoscute sub numele de anomalii ale fluxului de date. Pe de alta parte, testarea dinamica a fluxului de date implica identificarea caii de program de la codul sursa, bazat pe o clasa de criteriu de testare a fluxului de date. Cititorul poate constata ca exista multe asemanari intre testarea fluxului de control si testarea fluxului de date. Mai mult decat atat, exista o diferenta esentiala dintre cele doua abordari. Asemanarile provin din faptul ca ambele abordari identifica caile de program si pune accentul pe generarea de cazuri de testare de la aceste cai de program. Diferenta dintre cele doua consta in faptul ca, testarea fluxului de control este utilizata la inceput, in timp ce criteriile de selectie de testare a fluxului de date sunt utilizate mai tarziu.

## 3.2. Anomalia fluxului de date

O anomalie reprezinta un mod anormal de a efectua ceva. De exemplu, este anormal sa atribuim succesiv doua valori pentru o variabila fara sa folosim prima valoare. In mod similar, este anormal sa utilizam o valoare a unei variabile inainte de a atribui o valoare variabilei. O alta situatie anormala este de a genera o valoare de date si sa nu o folosim niciodata. In cele ce urmeaza, vom explica cele trei tipuri de anomalii care privesc generarea si utilizarea valorilor de date. Aceste anomalii pot fi manifestari ale potentialelor erori de programare. Vom explica de ce aceste anomalii de program nu trebuie sa conduca la esecuri de program.[1]

***Anomalie de tipul I ( definirea si redefinirea )***

Sa consideram secventa partiala a calculelor prezentate in figura de mai jos, unde f1(y) si f2(z) denota functii cu intrari y si z. Putem interpreta cele doua declaratii in mai multe moduri, cum ar fi :

* Calculul efectuat de prima declaratie este inutil in cazul in care a doua declaratie efectueaza calculul propus;
* Prima afirmatie are un defect. De exemplu, primul calcul propus poate fi w = f1(y);
* A doua afirmatie are un defect. De exemplu, al doilea calcul propus poate fi v = f2(z);
* Un al patrulea tip de defect poate fi prezentat in secventa data sub forma de o declaratie lipsa intre cele doua afirmatii. De exemplu, v = f3(x) poate fi afirmatia dorita ce ar trebui sa se afle intre cele doua afirmatii deja existente.

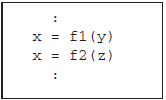


Figura 6. *Secventa a calculelor ce arata o anomalie a fluxului de date*

Programatorul face interpretarea dorita, desi acesta poate interpreta cele 2 afirmatii in unul din cele patru moduri explicate mai sus. Cu toate acestea, se poate spune ca exista o anomalie a fluxului de date in aceste 2 afirmatii, ce indica ca este nevoie de o examinare pentru a elimina orice confuzie pentru persoanele ce citesc codul respectiv.

***Anomalie de tipul al II-lea (nedefinit dar referit)***

Un al doilea tip de anomalie de flux de date apare cand utilizam o variabila nedefinita intr-un calcul, cum ar fi **x = x – y – w**, unde variabila **w** nu a fost initializata de catre programator si acesta avea intentia sa foloseasca o alta variabila initializata, sa zicem y, in locul variabile w. Oricare ar fi intentia reala a programatorului, exista o anomalie in folosirea variabilei w, iar pentru eliminarea anomaliei trebuie sa initializeze variabila w sau sa inlocuiasca variabila w cu variabila dorita.

***Anomalie de tipul al III-lea (definit dar fara referinta)***

Un al treilea tip de anomalie a fluxului de date este de a defini o variabila si apoi sa o redefinim fara sa o folosim in vreo secventa de calcul ulterioara.

De exemplu, sa consideram afirmatia x = f(x,y), unde o noua valoare este atribuita variabilei x. Daca valoarea lui x nu este utilizata in niciun calcul ulterior, atunci va aparea aceasta forma de anomalie ce se numeste “definit dar fara referinta”.

J.C. Huang a introdus idea de “stari” ale variabilelor de program pentru a identifica anomaliile fluxurilor de date.



Figura 7. Diagrama de tranzitie intre stari

De exemplu, initial, o variabila poate ramane in starea nedefinitia (U), ceea ce inseamna ca doar o locatie de memorie a fost alocata pentru variabila dar nicio valoare nu a fost inca atribuita. La un moment ulterior, programatorul poate efectua un calcul pentru a defini (d) variabila – aici este starea cand variabila se muta la “definita dar fara referinta” (D). Mai tarziu programatorul poate face referinta la variabila (r), si se va deplasa in starea de “definite si cu referinta” (R). Variabila ramane in starea R atata timp cat programatorul continua sa faca referinta valorii variabilei. Daca programatorul atribuie o noua valoare variabilei, atunci aceasta se intoarce la starea D ( definite dar fara referinta) .

Cu toate acestea, programatorul poate face greseli. Acesta poate lua actiuni gresite in timp ce o variabila este intr-o anumita stare. De exemplu, daca o variabila este in starea U (starea nedefinita) si programatorul citeste variabila (r), variabila se deplaseaza in starea A ( anormal). Starea anormala a unei variabile reprezinta faptul ca o anomalie de programare s-a produs.

Prezenta unei anomalii de flux de date intr-un program nu inseamna neaparat ca executarea programului va duce la un esec. O anomalie a fluxului de date inseamna ca programul poate esua, si , prin urmare, programatorul trebuie sa investigheze cauza anomaliei.

## 3.3 Testarea dinamica a fluxului de date

In procesul scrierii codului , un programator manipuleaza variabilele, in scopul de a obtine efectul dorit de calcul. Manipularea variabilelor apare in mai multe moduri, cum ar fi de initializare a unei variabile, atribuirea unei valori noi pentru variabila, calcularea unei valori a unei alte variabile folosind valoarea variabilei si controlul fluxului pentru executia programului.

O variabila poate fi folosita intr-un predicat,care este o conditie pentru a alege un flux adecvat de control. Testarea fluxului de date implica selectarea traseelor de intrare-iesire,cu obiectivul de a acoperi definirea si utilizarea datelor, de obicei cunoscuta sub numele de criteriu de testare a fluxului de date. In urma ideilor de mai sus avand in vedere testarea fluxului de date putem sa dam cateva indicatii:

* Desenati un flux de date dintr-un program
* Selectati unul sau mai multe criterii de testare a fluxului de date
* Identificarea cailor din graful fluxului de date ce indeplinesc criteriile de selectie[3]

## 3.4. Grafuri ale fluxului de date

*Graful de dependenta a datelor ( DDG – Data Dependency Graph)*

Exemplu :

integer x, y, z ;

input x, y;

z = y+x; [3]

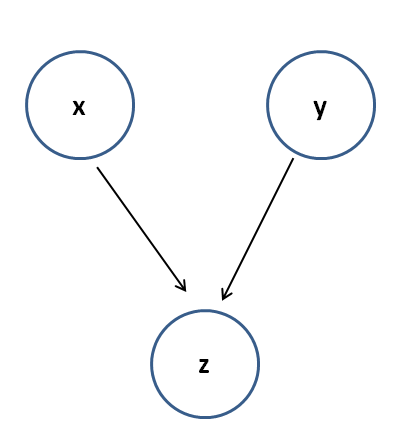


Figura 8. Graful de dependenta a datelor

* Nodurile reprezinta elemente de date
* Link-urile ( sagetile) reprezinta fluxul de x si y pentru z sau depedenta de z pe x si y

**Caracterizarea grafului de dependenta a datelor (DDG)**

Ne intereseaza cel mai mult relatia D-U (definirea datelor – folosirea datelor) cand vrem sa facem o testare a fluxului de date.

Datele folosite in predicatul nodurilor pentru P-utilizare.

Consideram un segment :

input w;

if w **≥** 3 then

z = x;

else z = y;

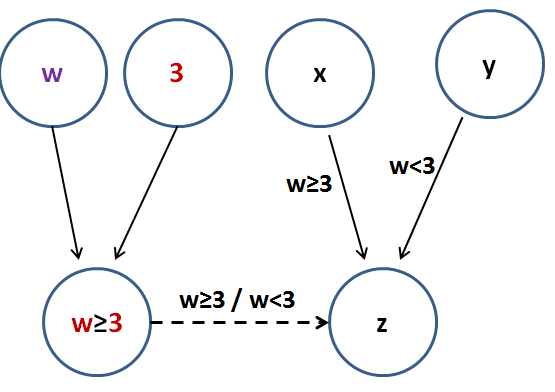


Figura 9. Caracterizarea grafului de dependenta a datelor

Retineti ca elementul de date, w este folosit in principiu pentru P-utilizare.

De retinut ca aceasta constanta 3 este de asemenea utilizata pentru P-utilizare

Sageata punctata descrie relatia P-utilizare [5]

# 4. Concluzii

(Lucru în echipă)

Din punctul de vedere al dezvoltatorului, testarea pe unitate este cea mai importanta clasă de testare pentru programul în curs de dezvoltare. Este tipul cel mai fundamental de testare, asigurând programtorii că codul respectiv îndeplinește cerințele de proiectare la cel mai scăzut nivel. În ciuda faptului că dezvoltatorii sunt mult mai preocupați de funcționarea corectă aprogramului decât cu spargerea acestuia în unități (module), testarea pe unitate în timpul dezvoltării oferă o bună stare de spirit pentru dezvoltarea ta ca un programator matur, eficient.

Testarea pe unitate se dovedește fiind o tehnică foarte valoroasă pentru a valida în mod constant software-ul în curs de dezvoltare (testare de regresie) și permite o automatizare într-o proporție cât mai mare a întregului proces de testare. Cu toate acestea, se ține cont de faptul că nu trebuie să se concentreze numai asupra testării pe unitate, existând mai multe teste care sunt importante, cum ar fi testele de uzabilitate, testele de performanță, testele de securitate. De asemenea, trebuie ținut cont de faptul că o suita de teste de lucru de 100% nu înseamnă că software-ul este absolut bug-free.

Noțiunea de cale într-o unitate de program este un concept fundamental. Presupunând că o unitate de program este o funcție, o cale este o secvență de instrucțiuni executabilă de la începutul executării funcției la declarația revenire în funcție. Dacă nu există nici o condiție ramificare într-o unitate de program, atunci există doar o singură cale în funcție. În general, există multe condiții ramificate într-o unitate de program și, astfel, există numeroase căi. O cale diferă de altă cale prin cel puțin o instrucțiune. O cale poate conține una sau mai multe bucle. Prin urmare, o cale este de lungime finită în termeni în raport cu numărul de instrucțiuni pe care le execută. Se poate avea o reprezentare grafică a unității de program, numit un graf de control, pentru a capta conceptul de flux de control în unitatea de program.

Fluxul de date intr-un program poate fi vizualizat prin luarea in considerare a faptului ca un program unitate accepta date de intrare, transforma datele de intrare, printr-o secventa de calcule, si , in final produce datele de iesire. Sunt trei actiuni fundamentale asociate cu variabila precum , nedefinire ( u ), definire (d) si referinta (r). O variabila este nedefinita implicit atunci cand este creata fara sa i se atribuie o valoare. Am explicat ideea de stari ale variabilelor, precum nedefinit (U), definit( D), referinta (R) si anormal (A) prin luarea in considerare a celor 3 actiuni fundamentale asociate cu variabila. Starea A reprezinta faptul ca variabila a fost accesata intr-un mod care provoaca anomalie a fluxului de date. Actiunile individuale pe o variabila nu provoaca anomalii ale fluxului de date. In schimb anumite secvente de actiuni duc la anomalii ale fluxului de date, precum : dd ( definire si redefinire) , ur (nedefinit dar referit) si du (definire dar fara referinta).

O variabila continua sa ramana in stare anormala (A) indifferent de actiunile ulterioare de indata ce intra in aceasta stare. Prezenta anomaliei fluxului de date in program nu va conduce la esuarea programului. Programatorul trebuie sa investigheze cauza anomaliei si sa modifice codul pentru a o putea elimina. De exemplu : o afirmatie ce lipseste din cod poate cauza o anomalie dd (definire si redefinire ), iar in acest caz programatorul trebuie sa scrie un cod nou.

Conceptul de testare a fluxului de date ne da noi criterii de selectie pentru a alege mai multe cai de program pentru a putea fi testate decat putem alege cu ajutorul testarii control flow.

# 5. Bibliografie

[1] - *Software Testing and Quality Assurance* – Theory and Practice -Kshirasagar Naik, Priyadarshi Tripathy

[2] - *The Art of Software Testing* - G. Myers.

[3] - *Selecting software test data using data ﬂow information -* S. Rapps and E. J. Weyuker

[4] - [http://cse.spsu.edu/ftsui/images/SWE6673\_2013\_Chapter\_11\_Data Flow Testing2.ppt](http://cse.spsu.edu/ftsui/images/SWE6673_2013_Chapter_11_Data%20Flow%20Testing2.ppt)

[5] - <https://ece.uwaterloo.ca/~snaik/MYBOOK1/Ch5-DataFlowTesting.ppt>

[6] - *Design and Code Inspections to Reduce Errors in Program* Development - M. E. Fagan - IBM Systems Journal

[7] - *Graph Theoretic Constructs for Program Control Flow Analysis* - F. E. Allen and J. Cocke.

[8] - *Data Flow Analysis in Software Reliability* - L. D. Fosdick and L. J. Osterweil.

Figurile–*Software Testing and Quality Assurance* – Theory and Practice (Kshirasagar Naik,Priyadarshi Tripathy)