FACULTATEA DE ELECTRONICA, TELECOMUNICATII SI TEHNOLOGIA INFORMATIEI

Implementarea, testarea, verificarea și validarea produselor software

Coordonator,

Conf. Dr. Ing. Ștefan Stăncescu

Lecu Radu Șerban

Tică Andra Maria

Vidrașcu Mihai

442A

-2011-

Cuprins

Bibliografie

5. Tipuri de teste software — Vidrașcu Mihai
   - 5.1. Testare prin metoda cutiei negre (Black-box)
   - 5.2. Testare white-box
   - 5.3. Testare gray-box
   - 5.4. Testarea funcțională
   - 5.5. Testarea nefuncțională
     - 5.5.1. Testare de compatibilitate
     - 5.5.2. Testare de anduranță
     - 5.5.3. Testare de incărcare
     - 5.5.4. Testare de localizare
     - 5.5.5. Testarea de performanță
     - 5.5.6. Testare de securitate
     - 5.5.7. Testare de utilizabilitate
   - 5.6. Concluzii

   Bibliografie

IV. Verificarea și validarea produselor software — Tică Andra Maria
1. Introducere
2. Recenzii
3. Verificări formale
   - 3.1. Model checking
   - 3.2. Inferenţa logică
   - 3.3. Derivarea de program
   - 3.4. Verificare prin demonstrare de teoreme
4. Concluzii
5. Bibliografie

V. Concluzii

Implementarea, testarea, verificarea și validarea produselor software

I. Etapele de realizare ale produselor software

Lecu Radu Șerban

1) Introducere

Dezvoltarea unui produs software presupune etape de analiză, proiectare, scriere, testare, debugging și mentenanță a codului sursă al programelor.

Dezvoltarea unui produs software poate fi folosită pentru a face referire la programarea calculatoarelor, însa într-un sens mai extins reprezintă totalitatea activităților de realizare ale unui produs software (= programarea calculatorarelor) ideal într-un proces planificat și structurat.

Scopul final este de a obține o soluție software eficientă și care poate evolua în timp. [4]

2) Etapele de realizare ale unui produs

Implementarea, testarea, verificarea și validarea produsului software sunt cele mai importante etape de realizare ale unui produs software. Pentru înțelegerea lor este însă nevoie de cunoasterea tuturor etapelor. Deși sunt considerate etape separate, intre ele există o puternică relație de interdependență.

2.1) Analiza cerințelor

Analiza cerințelor este prima etapă a ciclului de realizare a unui produs în care se stabilesc cerințele aplicației, pornind de la cerințele utilizatorului final, se identifică funcțiile viitorului produs software precum și datele implicate. Această etapă răspunde la întrebarea ce se va realiza prin dezvoltarea produsului software.[17]

2.2) Proiectarea arhitecturii software

Proiectarea reprezintă acea etapă a ciclului de realizare a unui produs în care se stabilește modul de realizare a cerințelor identificate în etapa de analiză, adică trebuie să raspundă la întrebarea cum se vor realiza aceste cerințe atât la nivel global cât și la

nivel de detaliu. Această etapă pornește deci de la cerințele și specificațiile definite anterior și continuă cu detalierea și transformarea acestora până la definirea structurii unei soluții care să fie reprezentată folosind un limbaj grafic, textual sau mixt. Proiectul astfel obținut trebuie să poata fi utilizat mai departe la construirea sau elaborarea produsului software (codificare, testare, integrare).[17]

2.3) Implementarea codului

Implementarea codului reprezintă scrierea textului folosind formatul și sintaxa unui limbaj de programare ales.

Codul este special conceput pentru a facilita munca unui programator, astfel oferin posibilitatea acestuia să specifice operațiile ce vor fi realizate de către calculator.

Odată ce un program a fost realizat el va fi convertit în cod binar numit cod mașină, care poate apoi fi citit si executat.[5]

2.4) Compilarea si interpretarea

Compilarea reprezintă o metodă prin care calculatorul convertește codul sursă scris într-un limbaj de programare (de cele mai multe ori un limbaj de nivel înalt cum ar fi c++, Java) într-un limbaj de nivel jos (cod mașină sau assembler). În urma realizării acestei operații se obține un fișier executabil ce poate fi înțeles și rulat de către mașina sau mașinile pentru care a fost conceput.[6]

Interpretorul reprezintă o metodă prin care mașina convertește codul sursă în cod mașină la momentul în care acesta este rulat. Interpretorul poate fi un program care fie execută codul sursă direct, fie transformă codul inițial într-un cod intermediar, iar acesta va fi încărcat pentru execuție, fie interpretează codul care anterior a fost compilat de către un compilator care face parte din sistemul de interpretare.[7]

Fiecare din cele 2 variante are avantajele si dezavantajele sale. Spre exempul un compilator este mai rapid în momentul execuției, însa interpretorul folosește principiul mașinii virtuale care oferă avantajul siguranței datelor.

2.5) Documentarea

În general documentarea se referă la procesul de a oferi dovezi.

Documentarea software sau documentarea codului sursă reprezintă text scris care exprimă modul de implementare al programului, modul de folosire al acestuia, informații legate de testarea programului, modificări aduse unei noi versiuni sau orice alte informații legate de programul respectiv.[8]

Există diferite tipuri de documente:

- Documente de cerințe software;
- Documente de arhitectură și design;
- Documente tehnice;
- Documente ale utilizatorului.

2.6) Integrarea

Integrarea se referă la faptul că un modul poate fi integrat în interiorul unui alt modul mai complex, sau că un modul mai complex poate fi realizat din mai multe module simple.

Prin integrare, datele sau informațiile oferite la ieșire de primul modul pot fi folosite de către un al doilea modul ca valori de intrare cu condiția ca ambele module să respecte acelasi format la ieșire, respectiv intrare. [9]

2.7) Testarea software

Testarea software reprezintă o investigatie dirijată în scopul de a obține informații legate de calitatea produsului software realizat în etapele anterioare.

2.8) Debugging

Debugging-ul reprezintă o metodă prin care se detectează și se elimina bug-urile (erorile) unui program (sau, general vorbind, în orice componentă electronică), în scopul de a-l face să funcționeze pe măsura așteptărilor.

Aparent debug este echivalent cu testarea, însă prin termenul de debug se intelege testarea, descoperirea cauzelor de eroare și rezolvarea acestor erori.

Debuggingul este puternic dependent de modul în care codul este implementat. Principala problemă este aceea că orice modificare adusă codului poate să ducă la modificarea funcționalității întregului program.

Astfel este foarte important modul în care codul este realizat pentru a facilita rezolvarea unor astfel de buguri. De aceea este necesară respectarea unor reguli de scriere a liniilor de cod cum ar fi: coupling, încapsularea datelor, cohesion. [10]

2.9) Verificarea și validarea software

Verificarea asigură că produsul e construit în concordanță cu cerințele, specifcațiile și standardele specificate. Validarea asigură că produsul va fi utilizabil pe piață.

2.10) Mentenanța

Odată lansat pe piață un produs software nu înseamnă că este lipsit de erori. Mai mult, deși produsul este funcțional, pot fi aduse înbunătățiri ulterioare.

Astfel dupa apariția versiunii inițiale procesele de implementare, testare, debug pot fi continuate. [11]

2.11) Specificarea

Specificarea (spec) reprezintă un set explicit de cerințe ce trebuiesc satisfacute de un material, produs, sau serviciu. Astfel putem spune că specificarea reprezintă un standard.

II. Implementarea produselor software

1) Introducere

Implementarea reprezintă realizarea unei aplicații sau execuția unui plan, unei idei, unui model, etc.

În știința calculatoarelor implementarea reprezintă realizarea unui algoritm sub formă de program.

Există numeroase aplicați care pot utiliza diferite standarde. Spre exemplu, la browser-ele web este de recomandat să conțină implementări ale World Wide Web Consortium, iar sistememele de dezvoltare conțin implementări ale limbajelor de programare.

Este cunoscut faptul că prezența unuia sau a mai multor utilizatori în cadrul tuturor etapelor de realizare a produsului este una benefică. De exemplu prezența utilizatorului la realizarea designului sistemului, li se oferă posibilitatea să modeleze sistemul în conformitate cu prioritățile lor și li se oferă posibilitatea să adapteze produsul final conform cerințelor lor. În plus există o probabilitate ca să reacționeze mult mai ușor la schimbări.

Implementarea unui produs software reprezintă munca în echipă a unui numar mare de dezvoltatori de programe de multe ori aflați în locații diferite (ex: tări diferite). Astfel este necesară folosirea unor metode de implementare a produsului software. O metodă de implementare a produsului software reprezinta o abordare sistematică structurata pentru a integra efectiv un serviciu software de bază sau o componentă în structura întregului produs. [5]

2) Scopul

Codul implementat are 2 roluri ce trebuie avute în vedere

- pentru a putea fi rulat de sistemul de calcul pentru care a fost destinal;
- pentru a fi inteles de programatorii care citesc codul, inclusiv cel care l-a conceput inițial.

3) Codul sursă

Codul sursă reprezintă reprezintă textul scris folosind formatul și sintaxa unui limbaj de programare ales.

Codul este special conceput pentru a facilita munca unui programator, astfel oferind posibilitatea acestuia să specifice operatiile ce vor fi realizate de către calculator. El poate fi văzut ca o modalitate prin care utilizatorul informează calculatorul ce și cum are de făcut în scopul manipulării anumitor date.

Odată ce un program a fost realizat el va fi convertit în cod binar numit cod mașină, care poate apoi fi citit si executat. Prin scrierea de cod utilizatorul informează un compilator sau interpretor cum să realizeze codul binar folosit pentru rularea programului.

Majoritatea aplicațiilor sunt distribuite într-un format de tip executabil, care nu conține cod sursă. Utlizatorul nu trebuie să cunoască modul cum a fost implementat programul, ci numai ce face programul. În caz contrar acesta ar putea modifica codul sursă sau înțelege cum funcționează.[12]

3.1) Scopul

Codul sursă este în primul rând folosit pentru a comunica unui sistem de calcul ce are de făcut.

Un alt scop este acela ca odată realizat un cod sursă într-o aplicație cu o anumită funcție, acesta să poată fi utilizat în multiple alte situații ale aceluiași program sau a unui alt program care care are de realizat aceiași funcție.

Această tehnică este folosită de către programatori atât pentru economisirea de timp cât și pentru învătarea de tehnici noi de implementare a unui program.

3.2) Organizarea

Codul sursă al unui program poate fi conținut de către un singur fișier sau de către mai multe fisiere.

Deși este de dorit evitarea acestei situații, în anumte cazuri este necesară scrierea codului sursă în limbaje de programare diferite. Spre exemplu limbajul de programare C poate conține și linii de cod scrise în limajul assembler.

3.3) Calitatea

Calitatea unui produs software este data de modul în care acesta este implementat și are implicații puternice în care ulterior, un alt programator poate înțelege testa și modifica codul inițial.

Calitatea produsului software este dată de anumite caracteristici care au fost prezentate mai sus. Pentru ca un produs să fie considerat de calitate sunt necesare respectarea anumitor reguli de implementare.

3.4) Licențierea

Un produs software odată realizat poate fi de 2 tipuri: free software sau proprietary software.

Prin free software se întelege acel software care poate fi folosit, distribuit, modificat, sau studiat de către oricine fără a fi nevoie să plătească pentru el.

În al doilea caz codul sursă este ținut secret, utilizatorul putând să aibă numai posibilitatea de utilizare a lui numai pe bază de licență.

4)Calitatea produselor software

Un produs software pentru a putea fi considerat funcțional este suficient să îndeplinească cerințele de realizare.

Un produs software pentru a putea fi considerat de calitate pe lângă faptul că este funcțional el trebuie să respecte și o serie de alte cerințe.

Nu există produse software lipsite de erori. Un produs nu este finalizat odată cu lansarea sa pe piată deoarece ulterior pot fi oricând adăugate funcționalități noi.

Un produs software trebuie astfel conceput încât să poată fi înțeles, testat, corectat și modificat cu ușurință și cu riscuri cît mai mici de generare de noi erori.

Produsele software sunt caracterizate de o serie de calități care au implicații atât în ceea ce privește implementarea codului sursă al produsului, cât și al altor operații cum ar fi testarea, verificarea, validarea, mentenanța, debugging, integrarea.

Caracteristicile necesare pordusului software pentru a putea fi considerat de calitate sunt:

4.1) Corectitudinea (reliability)

Prin corectitudine se exprimă cât de des rezultatul dat de program este corect. Reliability se referă la corectitudinea algoritmilor. Scopul este de a minimiza greșelile de program datorate managementului resurselor și erorilor logice. [4]

4.2) Robustețe (robustness)

Prin robustețe se intelege cât de bine reușeste programul să anticipeze probelemele apărute în cadrul manipulării datelor. Acesta se referă la detecția unor situații cum ar fi date incorecte, nepotrivite, distruse, nedisponibilitatea lor la un moment dat, probleme legate de indisponibilitatea de memorie, de servicii oferite de sistemul de operare, conexiune prin rețea sau probleme datorate utilizatorului și intrărilor introduse de acesta. [4]

4.3) Utilizabilitate (usability)

Utilizabilitatea se referă la ușurința cu care o persoană poate folosi un anumit program în scopul rezolvarii problemelor pentru care a fost realizat programul, sau în anumite cazuri și a unor probleme neanticipate. Aceasta implică o gamă largă de elemente atât de tip software ce țin de aspectul grafic al aplicației și de controlul produsului software, cât si de partea hardware a mașinii de calcul pe care rulează aplicația.

Elementele de tip grafic se referă la influența aspectului grafic al programului asupra utilizatorului, la intuitivitatea și coerența programului, iar celelalte elemente se referă la funcționalitatea programului adică la probleme care duc la stări de asteptare mult prea mari datorate implementării sau chiar la blocarea programului. [4]

4.4) Portabilitate (portability)

Prin portabilitate se înțelege gama de sisteme hardware și sisteme de operare pe care se poate interpreta sau compila și rula programul. Aceasta se referă la adaptabilitatea facilităților oferite de program la multitudinea de sisteme hardware și sisteme de operare. [4]

4.5) Mentenabilitate (maintainability)

Această proprietate se referă la ușurința cu care programul poate fi modificat de către programatorii prezenți și viitori cu scopul de a corecta buguri, de a face modificări, de a reliza facilități suplimentare sau de a-l adapta la alte sisteme hardware sau sisteme de operare. În acest caz trebuie luat în calcul modul cum a fost proiectat și implementat produsul.[4]

4.6) Eficiență/ performanță (efficiency/performance)

Prin aceste caracteristici se înțelege cantitatea de resurse folosite de către program. Cu cât sunt folosite mai puține resurse cu atât eficiența este mai mare. Aceastea includ distribuirea corectă de resurse cum ar fi ștergerea de fișiere temporare sau de date stocate inutil în memorie, folosirea inutilă a rețelei.

O importanță deosebită în acest caz o are și structura hardware. De multe ori un program este conceput în anumite scopuri pe baza cărora se alege și configurează sistemul de calcul pe care va funcționa programul respectiv. Astfel structura acestui sistem trebuie astfel aleasă încât costurile de cumpărare ale hardului să fie orientate către cresterea performanței produsului software prin creșterea performanțelor componentelor hardware folosite mai intens (ex: în cazul serverelor folosite intens pentru stocarea de date să se pună mai mult accentul pe viteza de stocare a datelor decât pe cea de procesare). Astfel componeta software este în strânsă legătură cu cea hardware. [4]

Alte caracteristici:

4.7) Lizibilitatea codului sursa

Prin lizibilitatea codului sursă se înțelege usurința cu care scopul, fluxul de informație și operațiile codului sursă pot fi înțelese de către un programator. Aceasta are implicații asupra unor calități menționate mai sus cum ar fi portabilitatea, utilizabilitatea sau mentenabilitatea.

4.8) Complexitatea algoritmilor

Complexitatea algoritmilor se referă la alegerea dintr-o mulțime de algoritmi care au aceiași funcționalitate însă care pot fi mai eficienți sau nu. Eficiența se poate face pe diferite criterii: timp, resurse folosite, dificultate de implementare.

Programatorii experimentați au sarcina de a alege cel mai eficient algoritm corespunzător aplicației pe care o implementează.

5) Limbajul de programare

Codul sursă este diferit în funcție de limbajul de programare ales. În plus putem avea diferite limbaje de programare folosite în cadrul aceluiași proiect în funcție de aplicația dorită. Există limbaje de programare procedurale și limbaje de programare obiect-orientate. Putem folosi în cazul aplicațiilor Web limbaje de programare destinate browser-elor (Web development): JSP, JavaScript, HTML. În cazul serverelor de multe ori trebuie stocată o cantitate mare de informație, caz în care vom folosi aplicații si limbaje de programare destinate realizării și gestiunii unor baze de date: SQL, Oracle.

Spre exemplu, putem folosi o aplicație care foloseste o interfață web scrisă în JSP care folosește diferite clase Java și cu ajutorul căreia se conectează la un server ce are o bază de date realizată folosind MySQL

5.1) Compilarea si interpretarea

În principiu, limbajele de nivel înalt pot fi grupate în 2 categorii: compilatoare și interptretoare. În realitate există rar programe care pot fi asociate exclusiv uneia din cele 2 categorii.

Compilarea reprezintă o metodă prin care calculatorul convertește codul sursă scris într-un limbaj de programare numit sursă (de cele mai multe ori un limbaj high-level cum ar fi c++, Java) într-un limbaj de calculator numit limbaj țintă (cod mașină sau assembler). Sursa inițială se numește cod sursă iar rezultatul cod obiect. În urma realizării acestei operații se obține un fișier executabil ce poate fi înțeles și rulat de către mașina sau mașinile pentru care a fost conceput.

Un program care face operația inversă poată numele de decompilator.

Comilatoarele oferă posibilitatea de dezvoltare a unor porgrame care sunt independente de mașina pe care acestea sunt rulate. În cazul programelor scrise în assembler codul trebuia modificat dacă era schimbată mașina pe care era rulat programul.

Un compilator pur este folosit numai în cazul limbajelor de tip low-level, deoarece este mai natural și mai eficient. [6]

Interpretorul reprezintă o metodă prin care mașina convertește codul sursă în cod mașină la momentul în care acesta este rulat.

Interpretorul poate fi:

- un program care execută codul sursă direct

- un program care transoformă codul inițial într-un cod intermediar, iar acesta va fi încărcat pentru execuție

- un program care interpretează codul care anterior a fost compilat de către un compilator care face parte din sistemul de interpretare.

Principalul dezavantaj al unui interpretor este că acesta este mult mai lent. Spre exemplu un program interpretat poate fi de 10 ori mai lent decât cel compilat. De aceea nu este niciodată folosit un interpretor pur. În schimb precompilarea programului si apoi interpretarea noului cod la rulare reduce puternic timpul de rulare, păstrând avantajele oferite de interpretare.[7]

5.2) Limbaje procedurale vs limbaje obiect-orientate

Printr-o procedură se înțelege o rutină, subrutină, metodă sau funcție care conține o serie de pași ce trebuiesc parcurși. Orice procedură poate fi apelată la orice moment de timp, în timpul execuției programului respectiv, inclusiv oferindu-se posibilitatea ca o procedură să se autoapeleze.

Scopul programelor procedurale este de a împărții funcția acestuia într-o colecție de variabile, structuri de date și subrutine.[13]

În cazul programării obiect-orientate, se folosesc obiecte, care interacționează în scopul de a realiza o anumită funcție.

Principalul avantaj al programării obiect orientate este acela că un program este mult mai usor de organizat. Un limbaj de programare procedural poate fi văzut ca o clasă care conține un singur obiect. În cazul programării obiect orientate acest obiect poate fi divizat în mai multe obiecte. După aceea putem introduce interefețe prin care obiectele pot interacționa.

Programarea obiect orientată oferă avantaje cum ar fi încapsularea datelor, moștenire, polimorfism, date abstracte care duc la simplificarea codului și a complexității acestuia, în același timp introducând avantajul securității ridicate ale datelor. [14]

5.3) Alte limbaje de programare folosite (limbaje de programare pentru aplicații web și limbaje de programare folosite pentru baze de date).

Inițial calculatoarele au fost realizate pentru a funcționa independent. Ele rulau programe simple care procesau cantități mici de informație. După apariția internetului a fost necesară extinderea acestor limbaje pentru a oferii posibilitatea de comunicare între calculatoare. Aplicațiile web și stocarea unor cantități foarte mari de informație stau la baza acestei dezvoltări.

Apariția browserelor web a dus la realizarea de limbaje de programare folosite de acestea. Au fost astfel realizate limbaje de programare cum ar fi HTML, JSP, JavaScript, PHP.

Prin Web development se înțelege procesul de realizare a unui site web. Acesta presupune aceleași etape ca in cazul orcărui alt tip de program.

Aplicațiile web au fost destinate a fi folosite de un număr mare de utilizatori din întreaga lume. Astfel a fost necesară realizarea unor servere pe care să fie stocate informațiile. Stocarea poate fi realizată folosind baze de date.

Bazele de date reprezintă o modalitate prin care o cantitate mare de informație poate fi organizată pentru ca datele să fi stocate, citite și modificate independent. De cele mai multe ori bazele de date oferă atât posibilitate de realizare și gestionare grafică, cât și sub formă de linii de cod.

6) Reguli de scriere a unui cod de calitate

Pe lângă faptul că programul trebuie bine documentat, un programator trebuie să aibă în vedere că scrierea unui program este numai una din multiplele etape de realizare ale unui program software profesional, iar programarea unui software complex presupune munca în echipă a unui numar mare de oameni.

Când un program complex este realizat există mai mulți programatori care lucrează la acel proiect. De multe ori un program este realizat de o anumită echipă de programatori, iar la un moment dat anumiți membrii sau chiar întreaga echipă trebuie să preia proiectul pentru a-l finaliza. Noii programatori care se ocupă de proiect trebuie să înteleagă liniile de cod scrise pentru a putea finaliza proiectul. Astfel codul trebuie astfel realizat încât să fie usor de inteles de către alte persoane. Trebuie realizat un program cu o complexitate cât mai scăzută.

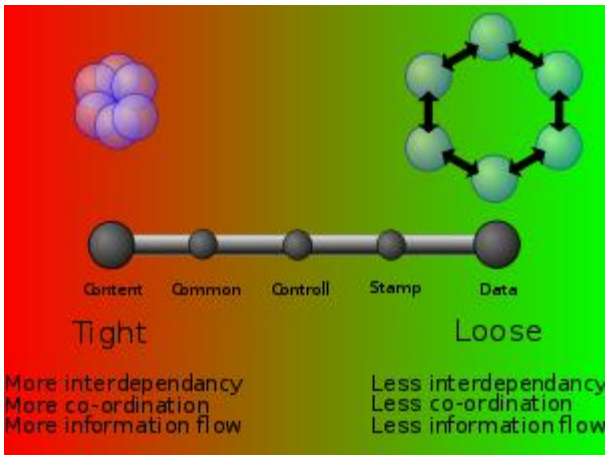
Un alt lucru ce nu trebe neglijat este că odată realizat un program, sau un modul al unui program cu o anumită fucționalitate, cu cât este mai bine organizat, cu atît poate fi mai usor integrat în cadrul altor aplicații care necesită aceiași funcționalitate fără sau cu un număr mic de modificări necesare.

În plus după etapa de implementare a codului urmează etape de debugging, testare, verificare, validare, mentenantă, etc. care sunt puternic dependente de modul în care este scris codul

Termenul de coupling (cuplarea) se referă la gradul de dependența dintre 2 subsisteme, adică la modul în care fiecare modul al unui porgram se bazează pe modul de funcționare al unui alt modul al aceluiași program sau al altuia și vice versa. [15]

Coupling poate fi definit ca fiind cantitatea de informație pe care un subsistem o are despre alt subsistem

Se disting 7 tipuri de coupling care sunt prezentate în figura de mai sus:



Modelul conceptula al couplingului (wikipedia)

-Content coupling (Tight coupling) - un modul se bazează pe modul de funcționare al altuli modul (ex. accesul la datele din acel modul).

- Common coupling - două module împart aceiași dată.

- Controll coupling - 2 module folosesc date, protocoale, sau interfață care au același format.

- Stamp coupling - 2 module folosesc același set de flaguri (steaguri) astfel influențându-se unul pe celălalt.

- Data coupling - 2 module împart aceleași date care sunt transmise de la unul la celălalt.

- Message coupling - comunicarea dintre cele 2 module se face pe bază de mesaje sau transmiterea de parametrii.

- No coupling - modulele nu comunică unul cu celălalt.

Micșorarea gradului de coupling se poate face prin interfețe. Interfața reprezintă un contract prin care un modul cunoaște numai informațiile oferite prin contract de celălalt modul.

Coupling este în general pus în contrast cu cel de cohesion (coesiune). Dacă coupling se referă la modul în care 2 module sunt legate între ele, coesiunea se referă la modul de implementare al unui singur modul.

Termenul de coesiune se referă la modul în care un modul are un scop bine definit.[16]

Se disting 7 tipuri de coesiune:

- Coincidental cohesion (lowest) - apare când părți ale unui modul sunt grupate arbitrar, singura relație dintre ele fiind că au fost grupate.

- Logical cohesion - apare când părți ale unui modul sunt grupate deoarece ele din punct devedere logic fac același lucru.

- Temporal cohesion - apare când părți ale unui modul sunt gurpate deoarece ele sunt rulate cu aproximație în aceiași perioadă de timp.

- Procedural cohesion - apare când părți ale unui modul sunt gurpate deoarece ele urmăresc aceiași secvență de instrucțiuni.

- Communicational cohesion - apare când părți ale unui modul sunt gurpate deoarece folosesc aceleași date.

- Sequential cohesion - apare când părți ale unui modul sunt gurpate deoarece o ieșire a unei părți este intrare a altei părți.

- Functional cohesion - apare când părți ale unui modul sunt gurpate deoarece ele contribuie la o singură funcție bine definită.

De ce este important să folosim coupling și cohesion?

Cei 2 termeni fac referire la complexitatea și modul în care este structurat un program. Principiul aplicat este „Divide et impera” (Dezbină și cucerește). În primul rând prin coeziune se divid module complexe în mai multe module mai simple cu funcționalități bine definite. În al doilea rând se micșorează gradul de coupling cât mai mult posibil astfel încât singurele dependențe între module să fie cele necesare (ex: când 1 modul este dependent de functionarea altuia putem fi nevoiți să folosim stamp coupling în loc de no coupling).

7) Documentarea codului

Fiecare etapă a realizarii produsului software presupune realizarea unei documentații corespunzătoare. Simpla implementare a codului nu este suficientă deoarece complexitatea unui program este foarte mare. Astfel atât utilizatorul cât și cei ce se ocupă de etapele urmatoare au nevoie de informații suplimentare, care să usureze munca acetora.

O altă problemă este aceea că un program presupune utilizarea mai multor limbaje de programare. Există programatori care nu cunosc unele limbaje folosite. Astfel pentru înțelegerea intregului program de către aceștia pot fi folosite alte metode. O variantă este scrierea de pseudocod. Realizarea acestor documente presupune și utilizarea unor forme grafice cum ar fi organigramele sau schite UML.

7.1) Pseudocod

Un limbaj pseudocod este o scriere intermediară, menită să simplifice scrierea unui algorit într-un limbaj de programare și să ajute la realizarea clarității algoritmului în timp scurt.

Pseudocodul poate fi vazut ca un limbaj de programare, menit nu pentru a fi introdus pe un calculator, ci utilizat ca o forma simplificată de scriere a unui algoritm folosit în cadrul unui program.

Pseudocodul este un limbaj apropiat de limbajul uman, astfel putând fi utilizat de către orice programator.

Sintaxa generală a pseudocodului este următoarea:

algorithm <nume_algoritm>

<declarare_variabile> {tipul variabilelor}

<declarare_proceduri> {definirea de proceduri/functii}

begin

<procesul_de_calcul_P> {instructiunile algoritmului}

end

Elemente lexicale ale limbajului:

- Cuvinte cheie - cuvinte care au un anumit înțeles și se folosesc numai într-un anumit context care sunt utilizate pentru a defini o anumită instrucțiune.

- Identificatori - nume de variabile (constante, variabile sau funcții) care sunt folosite pentru aplelarea acestora, în timpul desfașurării procedeului de calcul

- Expresiile - forme lexicale, asemănătoare operațiilor matematice, alcatuite din operatori și operanzi

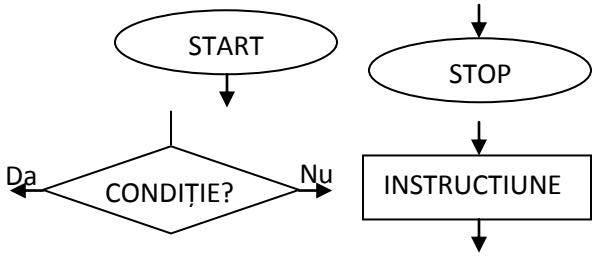
7.2 Organigrame

În locul scrierii de pseudocod se poate opta pentru folosirea unei reprezentări grafice a algoritmului numită organigramă.

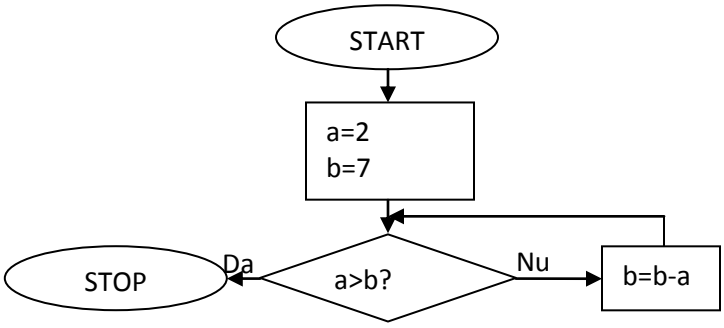
În cazul dezvoltării unui program software, organigrama este un grafic aparținând unui program sau algoritm destinat a fi implementat pe o mașină de calcul.

Elementele principale ale unei organigrame sunt:

- blocurile de start si stop
- blocul de scriere a unei instrucțiuni sau secvențe de instructiuni
- bloc de decizie
- legături intre blocuri (realizate printr-o săgeată de legatură)



Exemplu:



7.3 UML

UML este un limbaj folosit pentru a defini un produs software din punct de vedere al entităților participante, al fluxului de operații și al relațiilor dintre entitățile participante.

Un model este o abstractizare a unui sistem, într-un anumit context, pentru un scop bine precizat. Un model captează conceptele relevante ale acelui sistem (din punct de vedere al scopului pentru care a fost conceput), ignorând aspectele neesenţiale. In cursul etapelor de dezvoltare a aplicaţiilor software (analiza, proiectare, implementare, testare) se creează mai multe modele ale produsului, care captează diferite aspecte ale acestuia (structura, comportarea, instalarea, distribuirea, utilizarea).

Pentru modelarea sistemelor complexe (În special a sistemelor software) au fost concepute un număr mare de modalitaţi (tehnologii) şi limbaje care sa sustină aceste metodologii. Unificarea acestora s-a impus din necesitatea de a facilita schimburile de idei si proiecte si a fost susținută de personalități marcante din domeniul ingineriei software (*Software Engineering*), iar rezultatul l-a reprezentat limbajul UML.

UML (*Unified Modeling Language*) *este un limbaj pentru construirea, specificarea, vizualizarea şi documentarea modelelor*. El a fost conceput și standardizat pentru modelarea sistemelor software, dar poate fi folosit și în alte domenii de proiectare (hardware, construcții etc.).

Limbajul UML definește mai multe tipuri de diagrame (*modele*) care pot fi folosite pe parcursul oricărei etape (faze, iteratii) de proiectare (software sau alte domenii), oferind mai multe vederi ale sistemului dezvoltat. In UML sunt definite semantica (intelesul) și notațiile acestor diagrame, care pot fi folosite într-un mod foarte flexibil, în funcție de necesitătile particulare de dezvoltare, fără să se prevada nici o restricţie asupra modului de utilizare a diagramelor

La definirea cerințelor si la dezvoltarea unui sistem complex colaboreaza mai multe categorii de specialiști și fiecare dintre aceste categorii sunt interesate de aspecte diferite ale sistemului dezvoltat. Fiecare dintre categoriile de specialişti beneficiăza de cel puţin o vedere a sistemul reprezentata sub formă unei diagrame UML.

Cele mai importante diagrame specificate în limbajul UML sunt: diagrame ale claselor, diagrama secvențelor, diagrama de colaborare, diagrame de stare, diagrame de utilizare.

Diagrama claselor este partea esentiala a limbajului UML, în special in proiectele dezvoltate în abordarea obiect orientată. Diagrame ale clasele se dezvoltă atat în etapele de analiză cat si in etapele de proiectare, cu diferite niveluri de detaliere si ele reprezinta modelul conceptual al sistemului, care stabilește *structura* sistemului.

Diagramele de colaborare, *diagramele secventelor* si *diagramele de stare* reprezinta comportarea sistemului, in diferite faze de executie. Se folosesc In principal in proiectare si implementare.

Diagramele de cazuri de utilizare (*Use Case*) sunt diagrame de descriere a comportării sistemelor din punct de vedere al utilizatorului. Aceste diagrame sunt simplu de înțeles, astfel încat cu ele pot lucra atât dezvoltatorii (proiectantii sistemelor) cît și clienții acestora. Se folosesc în special în etapa de analiză, dar și în etapele de proiectare și testare.

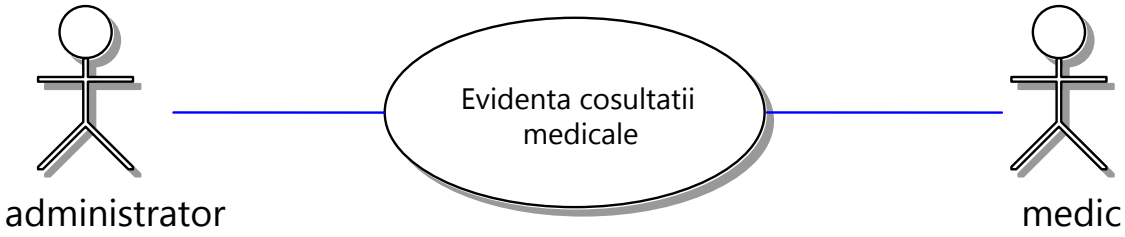
Mai există și alte diagrame: *diagrama componentelor*, *diagrame de instalare* (deployment), *diagrama pachetelor* (package).

Toate aceste diagrame se reprezintă folosind elemente ale limbajului (*simboluri*) care sunt figuri geometrice simple (linii, dreptunghiuri, ovale). Exista unele simboluri care se pot folosi în orice tip de diagramă (de exemplu simbolurile de documentare), iar alte simboluri sunt specifice diagramei în care se folosesc. Mai mult, unele simboluri (de exemplu o linie cu sageata la capat) poate avea diferite semnificații, în funcție de diagrama în care se folosesc.

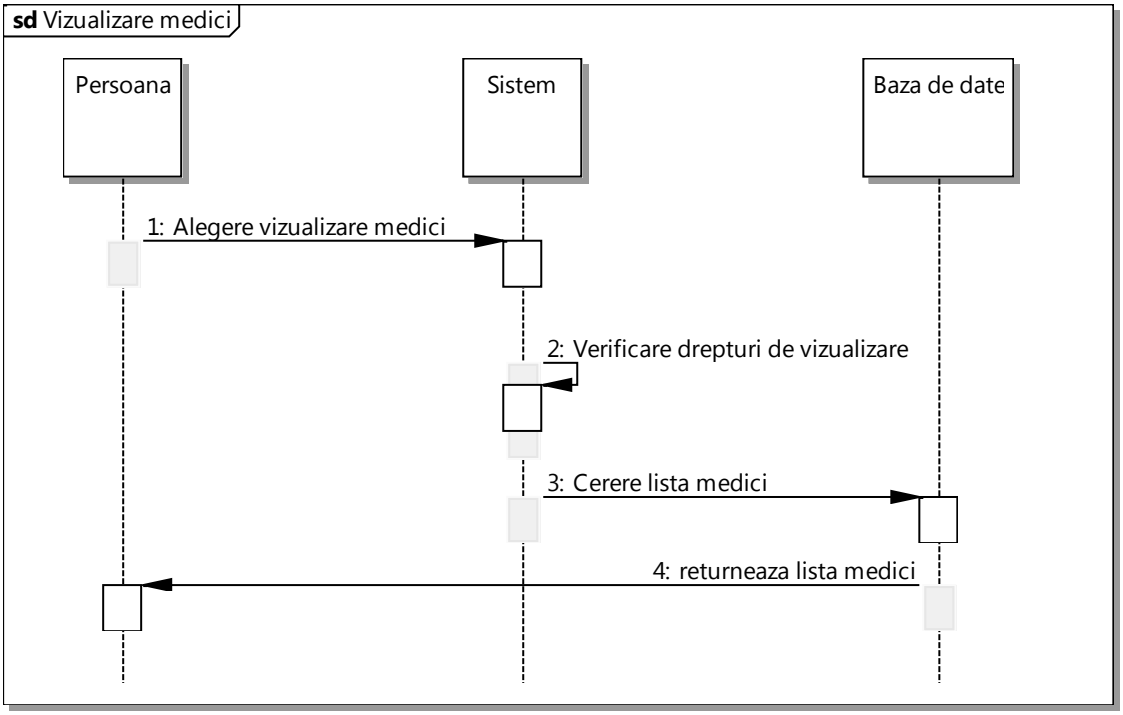
Există posibilitatea de a genera cod pe baza unui proiect UML, pentru diferite limbaje de programare: c#, java, folosind unelte de generare de cod. Astfel se poate accelera faza de implementare de cod, oferind avantajul eliminării erorilor de scriere a codului.

Exemple:

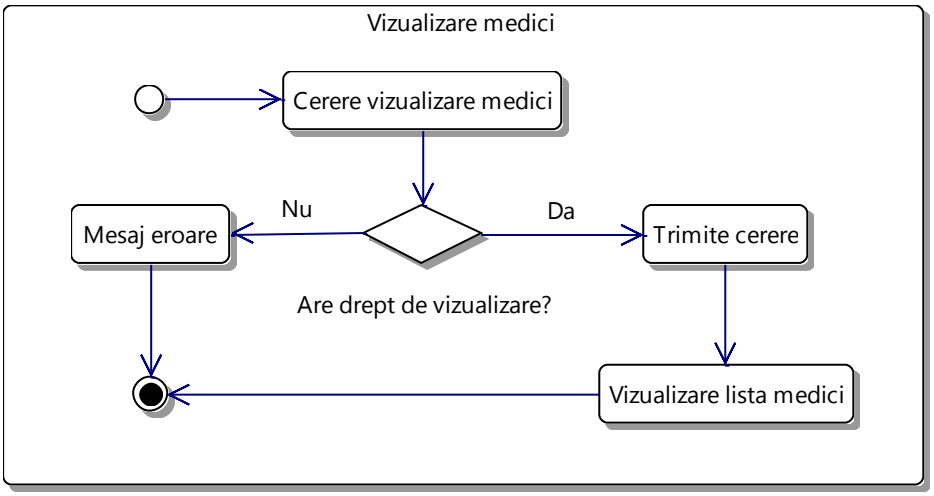
- Diagrama de context static



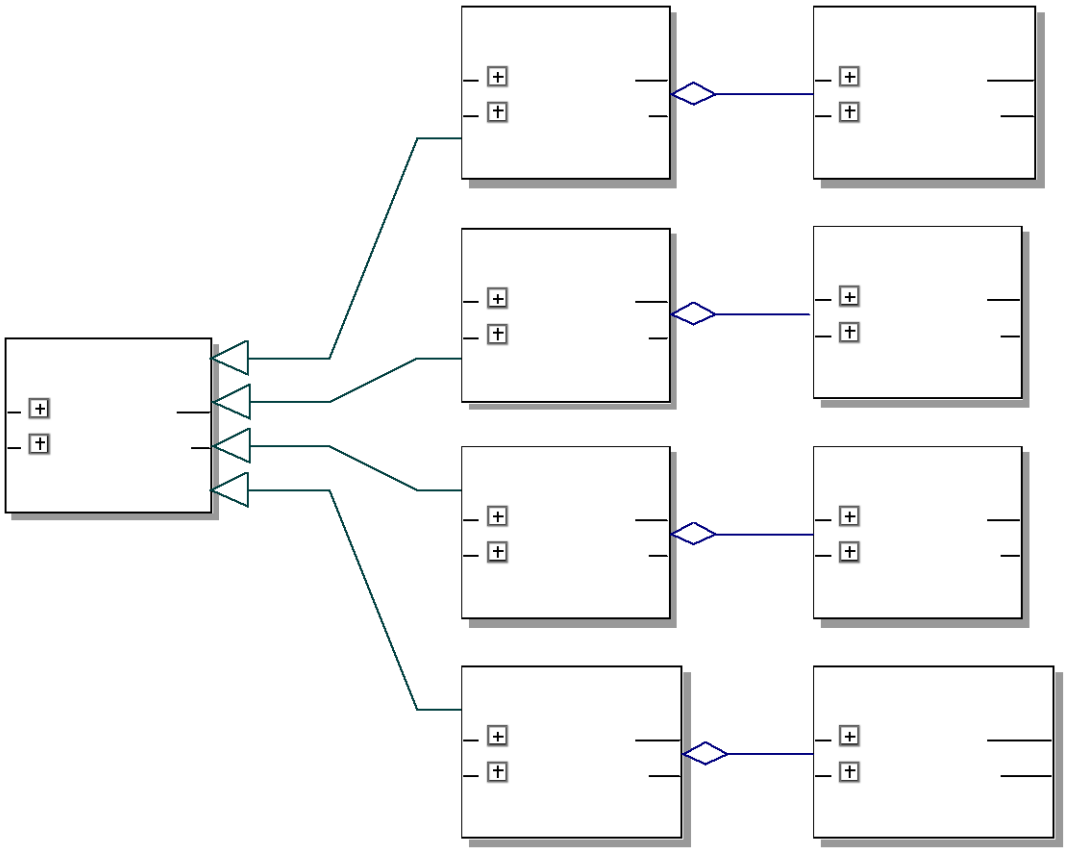
- Diagramă de secventă:



- Diagramă de activitate:



- Diagramă a claselor:



8) Concluzii

Implementarea unui produs software este o etapă dificilă prin faptul că în cadrul programelor complexe poate fi realizată numai de către un grup de oameni. Astfel se pune accentul pe lucrul în echipă.

Implementarea unui produs software reprezintă etapa de scriere efectivă a codului. Codul reprezintă atât un limbaj de comunicare între programator și calculator cât și unul folosit între mai mulți programatori. Astfel codul trebuie astfel realizat și organizat încât să poată fi înțeles și modificat sau utilizat de către un alt programator care ulterior va citi codul.

Există un număr mare de limbaje de programare. Alegerea limbajului de programare folosit este dependentă de caracteristicile programului. Deși este de preferat să fie folosit un singur limbaj de programare în cadrul realizării unui produs de cele mai multe ori este imposibil deoarece funcțiile necesare produsului nu pot fi acoperite de către un singur limbaj de programare.

Calitatea codului este esențială. Nici un cod nu este lipsit de erori. Astfel prin respectarea unor reguli elementare de scriere a codului se permite corectarea cu usurință a bug-urilor fără a duce la generarea de erori noi. În plus complexitatea este redusă fiind mult mai ușor de înțeles, modificat iar erorile sunt mai usor de detectat. În

al treilea rând modificarea codului pentru realizarea de noi versiuni este mult mai usoară, fără a fi necessare modificări majore ale vresiunii curente.

Pentru a simplifica munca celor ce preiau programul în cadrul următoarelor etape, documentarea are un rol important. Aceasta presupune cunoasterea si utilizarea diferitelor moduri de reprezentare a liniilor de cod, pseudocod, organigrame, UML.

III.Testarea produselor software

1) Introducere

Testarea software reprezintă o investigație dirijată în scopul de a obține informații legate de calitatea produsului software realizat în etapele anterioare. Testarea reprezintă rularea programului în scopul de a descoperi bug-uri software.

Testarea este o etapă puternic dependentă de contextul operațional în care programul va fi folosit. Testarea produselor software reprezintă o etapă importantă în dezvoltarea produsului software deoarece, 40-50 % din eforturi sunt directionate în această direcțe, mai ales pentru produsele software care necesită un grad mare de siguranță.

O analogie interesantă este similaritatea dintre testarea software și pesticide, care poartă numele de Pesticide Paradox. Orice metodă este folosită pentru găsirea si eliminarea gândacilor va lăsa în urmă anumite categorii de gîndaci care sunt imuni la acea metodă. Astfel orice metodă este folosită pentru detecția si corecția erorilor nu duce la dispariția în totalitate a tuturor erorilor. Din acest motiv defectele software sunt denumite bugs iar testarea și rezolvarea acestor defecte poată numele de debugging. Complexitatea și varietatea erorilor este atât de mare încât este imposibil minții umane să rezolve toate aceste erori. Eliminarea erorilor este atat de dificilă încât în timpul necesar pentru eliminarea tuturor acestora apar noi erori.

În urma testării se poate, sau nu, confirma faptul ca produsul software supus testării corespunde cerințelor care au dus la crearea acestui produs și ca functionează corespunzător asteptărilor.

Este evident că deși testarea reprezintă o etapă separată. Ea este puternic dependentă de etapa de dezvoltare și are la bază metodele de dezvoltare ale produlsului software.

Un produs software este diferit de orice alt sistem fizic cu intrări și ieșiri. Un produs software nu suferă defecte datorate trecerii timpului (ex: coroziunea în cazul unor componente electrice ale unui circuit). Singurele tipuri de defecte pe care le poate suferii un produs software sunt date de proiectare și implementre. Un produs software odată realizat, nu va mai suferi modificări decît în urma realizării unei noi versiuni a acelui program. Astfel, dacă un produs este declarat funcțional momentul în care are loc finalizarea testării el va fi funcțional la orice alt moment de timp ulterior.

Principala cauză a erorilor unui produs software este dată de complexitatea acestuia. Oamenii au o capacitate limitată de procesare în cazul unei complexități ridicate. Din acest motiv defectele de proiectare sunt inevitabile. De aceea este imposibil de conceput un produs software care sa nu conțină erori și nu există nici o modalitate de testare a produsului software care să elimine sau cel puțin să detecteze toate aceste defecte.

Există situații când rezolvarea unei erori să duca la apariția unora noi. Astfel, dacă este detectată o anumită eroare în faza de testare, se poate încerca remedierea ei. Este totuși posibil ca acea remediere să nu ducă la rezolvarea erorii, sau chiar să ducă la apariția unora noi. După realizarea corecției trebuie reluată faza de testare pentru a avea siguranța că modificarea realizată a dus la rezolvarea problemei fără introducerea unora noi. Este important de înțeles că orice modificare adusă unui program software trebuie însoțită de un set de teste care să asigure funcționalitatea noii versiuni înainte de a fi dată utilizatorilor. Cea mai mică modificare poate duce chiar la schimbarea totală a modului de funcționare a produsului software.

O altă situație este în cazul în care eroarea apare pentru un număr foarte restrâns de valori de intrare. Acesta poate fi un lucru favorabil dacă eroarea este neglijabilă (ex: deși nu se obține rezultatul dorit pentru acel domeniu de valori, apariția acestor erori nu duce la pierderi seminificative). În același timp, poate fi și un lucru nefavorabil deoarece această eroare este greu de detectat și e posibil să nu fie detectată în faza de testare. Deși probabilitatea de apariție a acestei erori este mică, în anumite situații efectele apărute în urma acestei erori pot fi devastatoare (ex: costuri mari, pierderi de vieți umane).

Testarea nu poate oferii o garanție de 100% că produsul funcționează corect în orice condiții ci numai în acele condiții pentru care a fost testată. Astfel alegerea intrărilor folosite este esențială pentru acoperirea unei game cât mai largi de posibile erori.

Informația obținută în urma testării este esențială pentru remedierea defectului respectiv. În momentul în care se detectează o eroare, este posibil să nu se cunoască și cauza care a generat acea eroare. Programatorul trebuie ca după detecția erorii să identifice pe baza acesteia instrucțiunea sau secvența de instrucțiuni care a dus la apariția erori.

În unele situații erorile pot să apară, nu din cauză de cod greșit, ci datorată unor lacune, neclarități la nivel de cerințe sau cerințe incomplete. Acestea pot duce la erori care apar încă din faza de proiectare de către designerul programului. Acestea poartă numele de cerinte non-funcționale.

O altă cauză importantă care poate duce la apariția unei erori este aceea de compatibilitate, ce poate fi cauzată atât de inteacțiunea lor cu alte aplicații, cu diferite sisteme de operare sau prin rularea acestora pe sisteme hardware diferite, cât și din

cauza nonconformărilor ce apar de la o versiune la alta într-un proces incremental de dezvoltare al produsului.

Dacă de exemplu produsul software a fost testat numai pe un singur sistem de operare se poate oferii siguranța în urma testătii produsului ca acesta funcționează corect numai pentru acest sistem de operare. Similare este și în cazul testarii produsului software pe o singură arhitectură hardware.

De la o versiune la alta a unui singur program, deși versiunea nouă nu are erori de funcționare, aceasta a suferit modificări care au dus la incompatibilitatea cu alte programe cu care versiunea anterioară putea interacționa fără probleme. Foarte posibilă este și apariția incompatibilității dintre 2 subdiviziuni ale aceluiași program.

2) Scopul testării

Ideal scopul testării este de a detecta și elimina toate erorile dintr-un program. Acest lucru este inposibil.

În realitate, scopul principal este detecția erorilor, corecția celor care pot fi eliminate, minimizarea acelora care nu pot fi eliminate (rezolvarea parțială a erorii, sau remedierea acelei erori deși se știe că vor apărea erori noi dar care pot fi ignorate) sau ignorarea ei în cazul în care eroarea nu poate fi eliminată sau costul remedierii erorii este mai mare decât costul datorat ei.

De asemenea scopul testării este acela de a oferi utilizatorului siguranța că produsul este conform cerințelor.

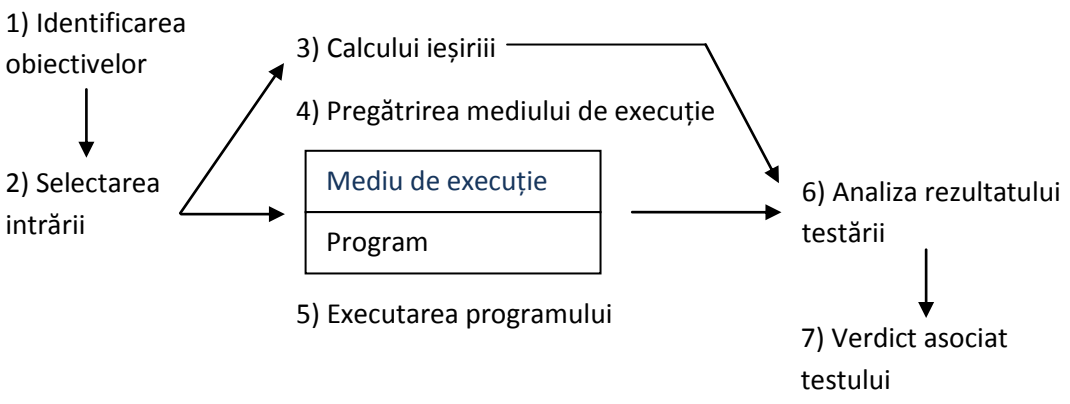
În general scopurile testării performanțelor produsului software sunt:

- cerceterea calității produsului software
- testarea si validarea acestuia.

Deoarece produsele software sunt folosite în aplicații critice, urmarile unui bug pot si dezastroase. Într-o lume în care aproape orice produs are și o componentă software testarea produselor software în scopul creșterii calității lor este esențială. ținând cont și de imperfecțiunea naturii umane, a fost necesară realizarea de tehnici, algoritmi și unelte pentru testarea produselor software.

3) Realizarea unui test software

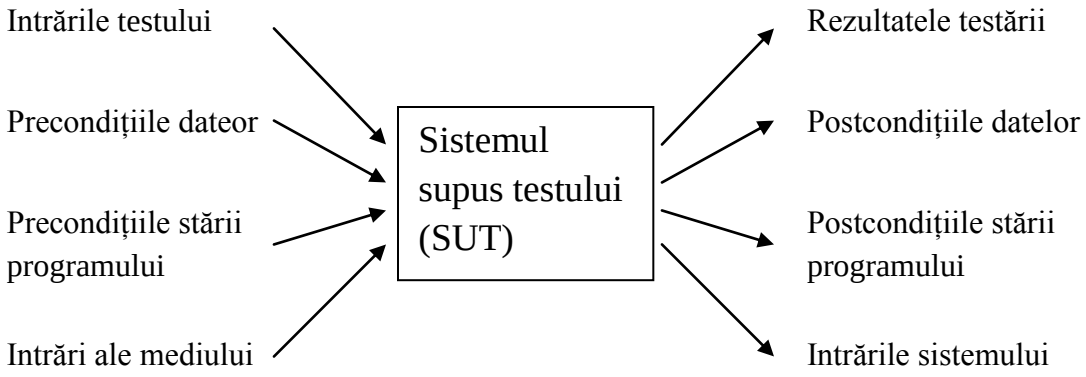
Pentru testarea unui produs trebuie realizată o secvență de activitate de testare.



4) Automatizarea testării

Scopul testării automate este de a minimiza cantitatea de muncă manuală în executara testului și acoperirea unei game mai mari de valori pentru a face testul prin aplicarea unui număr mai mare de teste. Testara automată are un impact major asupra metodelor de testare concepute și al uneltelor folosite pentru testare. Prin testarea automată se detectează blocările programelor și operațiile curente oferind informații de diagnosticare.

Un model de testare poate fi reprezentat astfel:

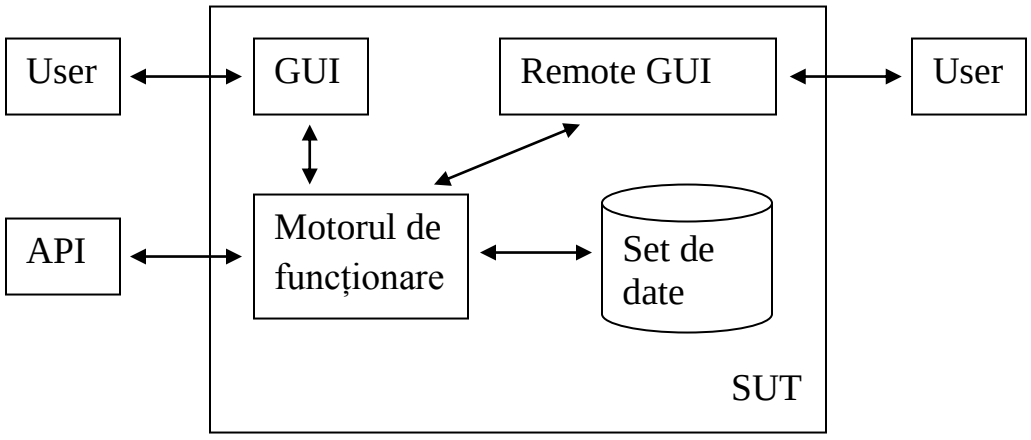


Un tester poate să identifice componentele unui program cum ar fi GUI (Graphic user interface) și poate deasemenea realiza anumite funcționalități cum ar fi calcule aritmetice, concatenări de stringuri, sau integrări de baze de date.

Sistemul supus testului (SUT) joacă un rol important în arhitecura uneltelor de test. Sistemul de test necesită introducerea de valori de intrare care pot fi alese numai în urma cunoașterii modului de funcționare al SUT-ului.

În cazul testării manuale utilizatorul aplică valori la intrare. La iesirea SUT-ului vom avea valori de ieșire care sunt comparate cu valorile dorite.

Un SUT poate fi reprezentat astfel:



Pornind de la această structură putem proiecta un sistem de testare astfel încât utilizatorul este inlocuit de unealta de testare. Acesta introduce valorile de intrare pentru fiecare intrare de test definită și preia valorile de ieșire corespunzătoare. Prin compararea valorilor de ieșire obținute de la SUT cu cele dorite se obțin rezultatele testării.[18]

Bibliografie

- [1]Head First Software Development - Dan Pilone & Russ Miles- editura O'REILLY
- [2]Software Development and Professional Practice
- [3]SCJP (Sun Certified Programmer for Java 6) - Kathy Sierra & Bert Bates editura Mc Graw Hill
- [4]http://en.wikipedia.org/wiki/Computer_programming
- [5] http://en.wikipedia.org/wiki/Implementation
- [6] http://en.wikipedia.org/wiki/Compiler
- [7] http://en.wikipedia.org/wiki/Interpreter_(computing)
- [8] http://en.wikipedia.org/wiki/Documentation
- [9] http://en.wikipedia.org/wiki/Digital_integration
- [10] http://en.wikipedia.org/wiki/Debugging
- [11] http://en.wikipedia.org/wiki/Software_maintenance
- [12] http://en.wikipedia.org/wiki/Source_code
- [13] http://en.wikipedia.org/wiki/Procedural_language
-[14] http://en.wikipedia.org/wiki/Object-oriented_programming
-[15] http://en.wikipedia.org/wiki/Coupling_(computer_programming)
- [16] http://en.wikipedia.org/wiki/Cohesion_(computer_science)
- [17] http://www.progapl.ase.ro/ps-id/Cap%205-1_4%20Metode%20de%20proiectare%20clasice.pdf
- [18] http://www.testingeducation.org/course_notes/hoffman_doug/test_automation/auto8.pdf
-[19] http://education.yahoo.com/reference/encyclopedia/entry/computer-prog
- [20] http://searchsoa.techtarget.com/definition/source-code
- [21] http://c2.com/cgi/wiki?WhatIsSoftwareDesign
- [22] http://www.inventoryops.com/software_selection.htm
-[23] http://www.shiva.pub.ro/PDF/TEST/Testarea%20software%20si%20asigurarea%20calitatii%20-%20curs2.pdf
- [24] http://www.chillarege.com/authwork/TestingBestPractice.pdf

- [25] http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/Presentations/PDF/ch23.pdf

Mihai Vidrașcu

5. Tipuri de teste software

5.1. Testarea prin metoda cutiei negre (black box)

Aceasta metodă testează funcționalitatea aplicației, nu abordează deloc codul și procesele interne ale aplicației. Tester-ul nu trebuie să știe decat ceea ce trebuie să faca programul, nu și cum face. Se urmărește „funcția de transfer" a programului: se dau date de intrare și se verifică datele de ieșire, fără a se știi cum se face prelucrarea lor. Cazuri de testare se construiesc în jurul cerințelor și specificațiilor. Aceste teste pot fi funcționale (cele mai multe) sau nefuncționale. Aceasta metodă se poate aplica la toate nivelele de testare: de unitate, de integrare, de sistem și de acceptare. [8]

Deci se testează:

- Funcționalitatea
- Validarea datelor de intrare (să nu se accepte date de intrare neconsistente, ca"-1" la data nașterii)
- Datele de ieșire
- Trecerea dintr-o stare în alta a sistemului (de exemplu la programul de instalare, componentele unei aplicații se instalează intr-o anumita ordine, sau la realizarea unei conexiuni).[3]

5.2. Testarea white-box

Testarea prin metoda cutiei albe (sau cutiei de sticlă/transparenta), sau testarea structurală este o tehnică de testare care verifică structura internă și modul de prelucrare a datelor (În opozitie cu testarea black-box descrisa mai sus). Pentru a efectua o astfel de testare, evaluatorul trebuie să știe cum a fost făcut programul și să aiba experienta în programare. Testerii introduc elemente de testare în cod care îi ajuta să determine dacă instrucțiunile s-au executat corect pânăîn punctele introduse. Este similar cu testarea unui circuit electronic în care se masoara din loc în loc parametrii pentru a evalua funcționarea corecta.

Se poate aplica la nivelul de unitate, de integrare și de sistem, dar cel mai des se folosește la nivelul unității. Deși astfel se descopera multe erori și probleme, nu se detectează partile neimplementate sau specificații și cerințe lipsa.[8], [5]

În categoria cutiei albe sunt incluse:

- Testarea fluxului de control
- Testarea cailor pe care le poate lua programul
- Testarea căii de date
- Testarea tratării corecte a erorilor [3]

5.3. Testarea gray-box

Testarea prin metodă cutia cutiei gri presupune ca testerul să cunoască algoritmii și structurile de date din interiorul programului, dar să testeze ca un utilizator (ca la black-box). Nu are nevoie de acces la codurile sursa. Manipularea datelor de intrare și de ieșire nu se califică în testarea gray-box, intrucât acestea se afla inafară sistemului (cutiei), insa modificarea structurii de date se incadreaza în aceasta categorie, pentru că, în mod normal, un utilizator nu ar putea schimba datele dinafara sistemului.

Testarea gray-box poate include și ingineria inversa pentru a determina, de pildă, valorile maxime și mesajele de eroare.

Cunoscand conceptele din spatele produsului software, un tester poate face alegeri mai bune și în cunostinta de cauza în ceea ce priveste tipurile de teste care trebuie efectuate. În general, unui astfel de tester i se permite pregătirea unui mediu de test propriu (de exemplu, sa adauge propriile date intr-o baza de date și apoi să creeze propriile scripturi SQL pentru a vedea răspunsurile).

În concluzie, testarea grey-box implementează teste inteligente, bazate pe cantități limitate de informație.[8], [3]

5.4. Testarea funcțională

Testarea funcțională se aplică pentru a verifică dacă un produs software se comporta și funcționează corect, conform specificațiile din proiect. O specificare funcțională este o descriere a comportamentului asteptat de la program. Indiferent de ce formă o ia, formală sau informală, specificarea funcțională este cea mai importantă sursa de informații pentru proiectarea testelor. Crearea de cazuri de testare din specificarile de program se numește testare funcțională.

Testarea funcțională, sau, mai precis, proiectarea funcțională de cazuri de testare, incearca să raspunda intrebarii: „Face programul ceea ce trebuie ?” , considerând numai specificațiile programului, nu și designul lui sau structurea de implementare. Fiind bazată pe specificațiile de program și nu pe cod, testarea funcțională se mai numește și testare black-box (metodă cutiei negre). Testarea funcțională este în general tehnica de baza pentru proiectarea de cazuri de testare. Proiectarea de teste poate incepe ca parte a procesului de specificare a cerințelor și poate continua prin fiecare nivel de proiectare și de interfata a specificațiilor; este

singura metodă de testare care se poate aplică atat de devreme și atat de larg. În plus, testarea funcțională este eficientă în detectarea unor clase de defecte care de obicei trec de testarea white-box (sau glass box) sau de testarea bazată pe defecte (detaliate în capitolele urmatoare).

Tehnicile de testare funcțională se pot aplică pentru orice descriere a comportamentului programului, de la descrierea informală parțială până la descrierea formală, și la orice nivel de granularitare, de la un singur modul la întregul sistem. De asemenea, proiectarea testelor în acest mod este mai ieftină și mai usor de executat de cat în cazut testării white-box.

În testarea și analiza aplicate în scopul verificării (adică a descoperirii oricaror discrepante intre ceea ce face un program și ceea ce ar trebui să faca), trebuie făcuta referire la cerințe, exprimate de de utilizatori, și specificate de inginerii software. O specificare funcțională (adică o descriere a comportamentului esteptat al programului)este sursa primară de intormatii pentru căzurile de test. Testarea funcțională, cunoscuta și ca testare black-box (sau testare bazată pe specificații) implică tehnici care creaza cazuri pentru testare derivate din specificațiile funcționale. În general, aceste tehnici produc specificații pentru căzurile de test care identifica anumite clase de teste, și care sunt instantiate pentru a produce teste individuale.

Principiul care stă la baza proiectării cazurilor de test este partitionarea posibilelor comportamente ale programului intr-un numar finit de clase omogene, unde fiecare clasa poate fi considerata corecta sau incorecta. Practic, proiectantul de cazuri de test trebuie să formălizeze specificațiile suficient de mult, astfel incat acestea să poata servi ca baza de identificare a claselor de comportamente. Un avantaj al proiectării de teste este acela ca scoate în evidenta slabiciunile și inconsistenta specificațiilor.

Crearea de cazuri de teste funcționale este un proces analitic care descompune specificațiile în cazuri. Multitidinea de aspecte care trebuie luate în considerare în timpul testării funcționae face ca procesul să fie predispus la erori. Chiar și proiectantii cu experienta pot omite cazuri importante de testare. O metodologie pentru proiectarea testelor funcționale ajuta la descompunerea design-ului de testare în pasi elementari. Astfel se poate controla complexitaea procesului și se pot separa activitatile umane de cele care pot fi automatizate.

Uneori, testarea funcțională poate fi complet automata. Este posibil atunci cand specificațiile sunt date sub formă unui model formăl (de exemplu, la un procesor de text, regulile gramaticale, sau starile în cazul unui automat). În aceste cazuri exceptionale, etapa de lucru efectiv are loc în simpul specificarii și proiectării software-ului. Treaba proiectantului de teste se limiteaza la alegerea criteriilor de test, care definesc strategia de generare a specificațiilor cazurilor de test. În majoritatea cazurilor insa, testarea funcțional este o activitate intensicv umana. Designer-ii trebuie să pornească de la

specificațiile informăle scrise în limbaj natural și să le structureze astfel incat să identifice cazurile de test.[4], [1]

5.5. Testarea nefuncțională

Testarea nefuncțională consta în testarea cerințelor nefuncționale alea produsului software. Se mai numește și testare structurală. O cerinta nefuncțională este un tip de cerinta care specifica criteriul ce poate fi folosit pentru a evalua operarea unui sistem, în locul unor comportamente specifice (cerințele funcționale definesc funcții și comportamente bine definite). Planul pentru implementarea cerințelor nefuncționale este detaliat în arhitectura sistemului. Cerințele nefuncționale sunt adesea numite calitati ale sistemului. Alti termeni care definesc acelasi lucru: constrangeri, atribute ale calitatii, scopuri ale calitati, cerințe ne-comportamentale.

Cerințele nefuncționale se impart în două categorii:

Calitate în executie, ca securitatea și usurinta în utilizare, care se poate observa în timpul rularii

Calitati de evolutie, ca testabilitate, mentenabilitate, extensibilitate și scalabilitate, care sunt incluse în structura statica a produsului software.

Revenind la testarea nefuncțională, aceasta include: testarea de compatibilitate, testarea de conformitate, de anduranță, de incarcare, de localizare, de performanță, de recuperare, de securitate, de scalabilitate, de stres, de utilizabilitate și testare de volum. [8]

În continuare le vom detalia pe fiecare, prezentand caracteristicile fiecarui tip de testare.

5.5.1. Testarea de compatibilitate

Acest tip de testare se aplică pentru a evalua compatibilitatea aplicației cu sistemul de calcul. Testarea unei aplicații pentru compatibilitate poate deveni o sarcina foarte complicata. Exista multe sisteme de operare diferite, configuratii hardware și software, deci se ajunge la un numar imens de combinatii. Nu exista o cale ocolitoare pentru problema testării compatibilitatii: aplicația fie ruleaza cum a fost proiectata pe un anumit sistem de calcul, fie nu ruleaza deloc. Un mediu potrivit pentru acest tip de testare poate totusi usura acest proces și il poate face mai eficient.

Clientul care cere dezvoltarea unei aplicații ar trebui să menționeze toate sistemele de operare-tinta posibile pentru utilizatorul final (inclusiv sistemele de operare pentru servere, dacă este cazul) pa care aplicația trebuie să ruleze. Exista și alte cerințe de compatibilitate care trebuie specificate: de exemplu, produsul de testat trebuie să ruleze cu mai multe versiuni de browsere web, cu dispozitive ca imprimante de toate felurile, sau cu alte componente software, ca programe de scanare antivirus sau procesoare de text.

Compatibilitatea se poate extinde și asupra actualizarilor de la versiuni mai vechi de software. Nu numai sistemul trebuie să poata fi actualizat corect de la o versiune mai veche, trebuie luate în considerare și datele și alte informații folosite în versiunile precedente (ca preferinte de setari. etc). Uneori, aplicația mai trebuie să fie compatibila și în sens invers, adică datele prelucrate și salvate de versiunile anterioare trebuie să poata fi accesate de versiunea noua și invers (datele noi trebuie să poata fi deschise pe statii de calcul care au versiunea veche a aplicației). Acestea sunt situatii care trebuie luate în calcul cand se pune problema compatibilitatii.

Data fiind diversitatea configuratiilor și potentialele probleme de compatibilitate, este probabil imposibil de testat fiecare permutare. Tester-ii trebuie să ierarhizeze configuratiile posibile în ordine, de la cele mai comune configuratii pânăla cele mai rar intalnite (de exemplu, dacă majoritate utilizatorilor folosesc Windows 7 și browser-ul Mozilla Firefox, aceasta configuratie ar trebui să fie prima în lista, punandu-se accent pe aceasta). Pentru alte configuratii mai rare, se pot impune limite de timp și cost și se pot face teste mai putÎn dure.

Pe langa selectarea celor mai importante configuratii, tester-ii trebuie să identifice și cele mai potrivite cazuri de test și de date pentru aceasta testare. De obicei nu este fezabila rularea întregului set de cazuri de test pe fiecare configuratie de test posibila (decat dacă aplicația este una mica), altfel ar dura prea mult timp. Deci, trebui selectat cel mai reprezentativ set de cazuri , care să confirme ca aplicația funcționează corect pe o anumita platformă.Pe langa acestea, tester-ii au nevoie și de date adecvate pentru testare, care să suporte acel set de cazuri de test.

Este importantă actualizarea continua a cazurilor de test de compatibilitate și datele pe masura ce sunt descoperite probleme în aplicație, nu numai în timpul dezvoltarii ei, ci și dupa livrare. Datele colectate de la utilizatori în cazul defectarii sunt o sursa buna de informații pentru crearea și adaugarea de noi teste în suita de testare. Aceste date sunt utile și pentru determinarea celor mai potrivite configuratii pentru căre să se faca teste de compatibilitate (se poate ajunge la refacerea ierarhiei de configuratii folosita la testare). O alta sursa de informații este programul lansat pentru beta-testing.

Pentru a testa aplicația ca la carte, trebuie re-creat mediul în care ea va opera la utilizatorul final. Pentru a indeplini aceste conditii, se instalează sistemul de operare, hardware-ul și configuratia software, și abia atunci se executa testele pe diferite set-up-uri. Cum pot exista nenumarate combinatii de setari, este cam scumpa achizitionarea masinilor pe care le-ar putea avea utilizatorii. De asemenea, reinstalarea aplicației pe fiecare ar dura prea mult pentru a fi o optiune viabila.

O abordare mai fezabila implică utilizarea de hard-disk-uri care se pot demonta usor, în combinatie cu o unealta de gestiune a partitiilor. Astfel, pe un numar mic de masini, se poate testa un numar mai mare de configuratii. Tester-ul doar plaseaza discul care trebuie. Restarteaza masina și selecteaza configuratia dorita.

Alta solutie este aceea de a folosi programe care creaza imagini diferite pentru hard-disk (ca Symantec GHOST). Astfel se creaza fisiere imagine pentru configuratiile necesare, care pot fi folosite oentru a recrea configuratia pe alta masina (sau chiar pe aceeasi) mai tarziu. Singurul dezavantaj este acela ca dureaza destul de mult timp refacerea configuratiilor pe masina tinta folosind fisierele imagine (depinde de dimensiune). În plus, și fisierele imagine sunt mari ca dimensiune, deci trebuie gasita o cale usoara (si compatibila pentru fiecare computer) de transport de pe o masina pe alta.

Indiferent de tehnică folosita pentru a gestiona partitiile, odata ce s-a setat mediul necesar, testele pot fi rulate și se poate evalua compatibilitatea pe platformă respectiva.

Este important și modul de instalare al aplicației pe configuratia setat anterior. Este critica respectarea modului de instalare pe care il va folosi utilizatorul, folosind aceleasi versiuni de componente și aceleasi proceduri de instalare. Aceasta constrangere se respecta cel mai bine atunci cand un program de instalare este pus la dispozitie devreme în fluxul de proiectare, și software-ul este predat echipei de testare sub formă instalabila, în loc de un pachet specializat. Dacă o metodă specializata sau manuala este folosita la instalare, este posibil ca mediul de test să nu reflecte mediul utilizatorului, și rezultatele testului de compatibilitate să fie imprecise.

Este important să nu existe unelte de dezvoltare pe platformă de test, pentru că ele pot interfera și mediu nu va mai fi atat de similar cu conditiile reale de operare.[7],[4],[5]

5.5.2. Testarea de anduranță (soak testing)

Acest tip de test se folosește pentru a determina dacă o aplicație face fata incarcarii de procesare siîșipastreaza comportamentul corect o perioada lunga de timp. În timpul testării de anduranță, se monitorizeaza atent consumul de resurse, mai ales memoria, pentru a observa potentialele defecte. De exemplu, un program poate funcționa corect cand se testează 1 ora. Dar peste o zi, apar probleme, ca pierderi de memorie, care schimba comportamentul în mod nedorit.

În timpul acestor teste se monitorizeaza și performanță produsului. Observatiile inregistrate în timpul soak-testing-ului se folosesc pentru a imbunatatii parametrii elementului testat. De multe ori se urmărește ca parametrii să nu coboare sub un anumit nivel dupa funcționarea prelungita.

Un sistem complet integrat trebuie să fie funcțional continuu, saptamani (sau chiar luni) întregi fără a se bloca/intrerupe.[9]

5.5.3. Testarea de incarcare (load testing)

Testarea la sarcina consta în incarcarea aplicației cu cerințe și masurarea răspunsului și parametrilor ei. Se aplică pentru a determina comportamentul unui sistem

sub conditii normale de funcționare și în cazuri de supra-sarcina. Astfel se poate afla care este capacitatea maxima de operare a aplicației, dar și unde se afla punctele slabe (bottle-necks) care cauzeaza degradarea performăntei. Uneori sarcina este pusa intentionat mult peste capacitatiile aplicației pentru a se observa tipul de erori care pot aparea și pentru a se crea solutii pentru ele (metode de scadere a incarcarii sau macar de semnalare a limitarii).

În multe cazuri, în software, incarcarea inseamna acces concurent al mai multor utilizatori. Situatia în care relevanta acestui test este maxima se intalneste la aplicațiile multi-user, de tip client-server.

Exista și alte tipuri de software care pot fi testate prin incarcare. De exemplu, un procesor de text sau un editor grafic pot fi fortate să citeasca un document extrem de mare, sau un program din domeniul bancar, caruia i se poate cere să genereze un raport folosind date distribuite pe mai multi ani. Cel mai precis test de sarcina este cel care simuleaza lucrul în conditii normale (la capatul opus se afla testele folosind modelarea teoretica sau analitica).

Pentru un website, testarea de incarcare poate ajuta la determinarea calitatii serviciilor (QoS performănce) folosind comportamentul asteptat al clientului. Aproape toate uneltele de testare prin incarcare urmaresc principiul clasic de testare. Cand clientul viziteaza site-ul, un programel inregistreaza comunicarea și apoi creaza script-urile de interactiune asociate. Apoi, un generator de sarcina incearca să redea scripturile inregistrare, care pot fi modificate inainte de rulare, pentru a avea diferiti parametrii și a simula mai multi clienti. La re-rularea procedurii, se monitorizeaza și se colecteaza statistici hardware și software. Acestea includ starile procesorului, ale memoriei, operatiile de intrare-ieșire ale discului pentru servere, timpul de răspuns, transferul sistemului de testat etc. La final, aceste statistici se vor analiza și se intocmeste un raport de testare la incarcare.

Testele de sarcina analizeaza foarte precis aplicațiile care au fost proiectate pentru mai multi utilizatori, prin incarcarea lor cu un numar divers de utilizatori virtuali și reali și masurarea performăntelor în acest conditii. Aceste teste se fac de obicei în medii identice (sau cat mai apropiate) cu cele reale.

Exemplu: un site de tip magazin virtual. Se evalueaza funcționarea componentei care se ocupa de cosul de cumparaturi. De pildă, trebuie să suporte 100 de accese concurente, de tipuri diferite: 25 de utilizatori virtuali (UV) se logheaza, se uita prin catalog și apoi se de-logheaza. Alti 25 se locheaza, adauga produse în cos, le cumpara și apoi pleaca. Inca 25 se locheaza, returneaza produsele cumparate anterior și pleaca, și inca 25 se logheaza pentru prima data (nu au alte activitati anterioare).

Specificul unui plan de test sau script variaza în funcție de organizare. Pentur exemplul de mai sus primii 25 de utilizatori virtuali ar putea cauta produse unice, sau aleatoare, sau pot selecta dintr-un anumit set; depinde de planul de test dezvoltat

anterior. Oricum ar fi, toate testele de incarcare vor să simuleze performăntele pe o gama cat mai mare de varfuri de lucru.

O credinta gresita este aceea ca uneltele de testare ofera doar inregistrare și redare (ca uneltele de testare prin regresie). De fapt, ele analizeaza intreaga stiva OSI (uneltele de testare prin regresie se axeaza pe performanță interferei grafice). De exemplu, o unealta de regresie inregistreaza și reda un click pe un buton intr-un browser, dar o unealta de testare prin incarcare trimite „hipertext-ul" (html) pe care browser-ul il emite dupa ce utilizatorul apasa acel buton. Intr-un mediu multi-user, uneltele de incarcare pot trimite comenzi pentru mai multi utiizatori, ficare cu ID, parola, setari diferite.

Experienta utilizatorului la testul de incarcare

În exemplul de mai sus, 100 de utilizatori virtuali lucrau cu aplicația de magazÎn virtual. Performanță se masoara aici din punctul de vedere al utilizatorului. Aceasta descrevalueazaie cat de repede raspunde produsul software și cat de satisfăcut este user-ul (cum percepe el performanță).

Unelte de testare prin incarcare

- AppLoader: solutie de testare de performănte și de incarcare. Se testează la nivel de interfata grafica. Poate fi folosit și pentru testarea de unitate, de integrare și de regresie.
- IBM Rational Performănce Tester: unealta de testare de scara larga, pe baza ed Eclipse. Se folosește în principal pentru executia unui numar foarte mare de teste pentru a masura timpul de răspuns al aplicațiilor pentru servere.
- Apache Jmeter: aplicație desktop Java pentru testarea de incarcare și masurarea performăntelor.
- LoadTest: verifică funcționalitatea și performanță sub sarcina.
- LoadRunner: utilizata pentru executarea unui volum mare de teste concurente
- OpenSTA (Open System Testing Arhitecture): aplicație de test prin incarcare, open source.
- SilkPerformer: testare de performănte intr-un model deschis și partajabil, care simuleaza un teste realistice de incarcare pentru mii de utilizatori, care ruleaza scenarii de afaceri.
- SLAMD: open source, aplicație web Java
- Visual Studio Load Test: unealta inclusa în Visual Studio, care permite dezvoltatorilor să rulese un numar mare de teste cu o combinatie de configuratii care simuleaza incarcarea reala de la utilizator.

Testare de incarcare se poate face în 2 feluri:

- testare de longevitate (sau de anduranță, descrisa mai sus), care evalueaza abilitatea sistemului de a suporta o sarcina moderata, costanta, timp indelungat.
- Testarea de volum, care supune aplicația la sarcina maxima o perioada limitata de timp.[8]

Testarea de sarcina difera de testarea prin stres (care evalueaza modul în care produsul lucreaza sub sarcina extrema, peste cea maxima, specificata).[11]

5.5.4. Testarea de localizare

Testarea de localizare este axata pe aspectele de internătionalizarea și de localizarea ale produsului software. Localizarea este procesul de adaptare a unei aplicații globalizate pentru o anumita cultura. Pentru a localiza o aplicație este nevoie de o intelegere de baza a seturilor de caractere și a problemelor asociate lor. Localizarea include și traducerea interfetei cu utilizatorul și adaptarea graficii pentru zona în care va fi distribuit produsul. De asemenea, și documentatia trebuie tradusa, inclusiv cea de ajutor (help menu).

O atentie speciala trebuie acordata solutiilor pentru afaceri. Trebuie implementate corect procesele tipice practicii locale. Diferentele legate de desfasurarea afacerilor în anumite culturi sunt date de cerințele guvernamentale și legislative ale tarii respective, deci localizarea produselor destinate afacerilor poate deveni o sarcina dificila.

Testarea de localizare evalueaza cat de bine este localizat produsul intr-o anumita limba-tinta. Acest test se bazeaza pe resultatele testării globalizate, unde suportul funcțional pentru o anumita tara a fost deja verificăt (se compara cu acesta). Dacă produsul nu este suficient de bine globalizat pentru a suporta o anumita limba, mai bine se evita localizarea în acea tara decat as apara probleme aditionale, care pot deveni greu de rezolvat (si cu costuri mai mari, pentru că, cu cat o problema este descoperita mai tarziu, cu atat solutionarea este mai scumpa). La livrare, producatorul trebuie să se asigura ca aplicația funcționează corect pe piata pentru căre a fost proiectat. Mai jos sunt cateva puncte pe care trebuie să se puna accent la testul de localizare:

- Componente care se altereaza în timpul localizarii, ca interfata cu utilizatorul și anumite fisiere:
- Sistemul de operare;
- Versiuni de tastaturi (pentru fiecare limba, accente, diacritice, caractere speciale etc.);
- Filtre de text;
- Hot-key-uri (combinatii de taste);

- Reguli de verificăre a scrierii;
- Reguli de sortare;
- Trecerea de la litere mai la litere mici (si invers) pentru anumite cuvinte;
- Imprimante;
- Diverse dimensiuni ale hartiei pentru imprimat;
- Mouse-ul;
- Formătul de data;
- Rigle și masuratori;
- Disponibilitate de memorie;
- Interfata vocala (accente) (unde este cazul);
- Continut video;[8]

5.5.5. Testarea de performanță

Aceasta testare se folosește pentru estimarea comportamentului unui produs software, în privinta răspunsului și stabilitatii, sub o anumita sarcina. Este utila și pentru a investiga, masura, valida sau verifică alte calitati ale sistemului, ca scalabilitatea, fiabilitatea și utilizarea de resurse.

Testarea de performanță este o categorie mai larga, car include alte subtipuri de testare, dintre care unele vor fi abordate în paragrafele urmatoare, iar altele au fost deja descrise:

- *Testare la sarcina:* cea mai simpla formă de testare de performanță. Evalueaza comportamentul produsului la o anumita sarcina asteptata (exemplu: multiple tranzactii, accesuri multiple de la utilizatori etc.). Evidentiaza foarte bine bottle-neck-urile (gatuirile care limiteaza performăntele).

- *Testarea de anduranță (soak testing*): pentru determinarea comportamentului aplicației la incarcare continua. Se urmărește gestionarea memoriei și degradarea performăntelor în raport cu timpul (exemplu: timpul de răspuns pentru o aplicație server-client, care nu trebuie să creasca peste o anumita valoare);

- *Testarea de stres (stress testing):* se folosește pentru determinarea limitelor superioare ale capacitatii aplicației (pentru toti parametrii de masurat) și a robustetii ei la incarcare extrema;

- *Testarea în impulsuri (spike testing):* dupa cum sugereaza și numele, se realizeaza print simularea cresterii brutse a numarului de utilizatori (acolo unde se aplică) și evaluarea comportamentului produsului.

- *Testarea în configuratie:* este o alta varianta clasica a testării de performanță. În locul testării performăntei prin perspectiva incarcarii, se testează efectele schimbarii de configuratie din mediul programului asupta comportamentului și performăntei.

- *Testarea de izolare:* nu este un test unic de performanță. Este mai mult un termen folosit pentru a denumi repetarea unui anumit test în scopul evidentierii defectului.

Scopurile testării de performanță:

- Demonstreaza ca aplicațiaîșiatinge performantele cerute.
- Poate fi folosita pentru compararea a două solutii diferite.
- Poate determina care componenta din produs sau din setul de date duce la comportamente neasteptate.

Conditii de test

În testarea de performanță este important (si de cele mai multe ori și dificil) ca mediul de test să fie similar cu cel asteptat la funcționarea normala, insa nu este intotdeauna posibil în practica. Motivul este acela ca, în productie (adică în timpul funcționarii), sarcina de lucru are o natura aleatoare, si, deși sarcinile de test fac tot posibilul de a le imita pe cele reale, este imposibila replicarea exacta a variabilitatii din realitate.

Implementarile arhitecturale slab legare au creat dificultati aditionale tester-ilor. Serviciile pentru intreprinderi au nevoie de testare coordonata (cu toti consumatorii care creaza volume de tranzactii similare cu cele din realitate) pentru a replica cat mai precis starile care apar în funcționarea reala. Din cauza costurilor, complexitatii și a constrangerilor legate de timp, majoritatea organizatiilor folosesc acum unelte care monitorizeaza și creaza conditii similare cu cele reale (denumite și „zgomot") în mediile lor de test.

Este critic pentru costurile de dezvoltare ca testele de performanță să aiba loc cat mai repede. Cu cat un defect este descoperit mai devreme, cu atat este mai ieftÎn de remediat.

Mituri ale testării de performanță:

1. Testarea de performanță este folosita pentru a defecta produsul

Aceasta testare se face pentru a intelege punctul slab al produsului. Altfel, testarea sub conditii normale de funcționare se aplică (pentru a determina comportamentul sub sarcina asteptata de la utilizator). În funcție de cerințe, se poate face și testare pentru pike-uri, anduranță sau de stres.

2. Testarea de performanță ar trebui făcută numai dupa testarea de integrare

Deși asa se procedeaza de multe ori în industrie, testarea ed performanță se poate face și în timpul dezvoltarii. Se numește testare timpurie de performanță. Astfel, proiectarea se face cu cerințele de performanță mereu în vedere; gasirea unui bug în performănte ar fi mai usor de remediat în etapele de dezvoltare decat dupa implementarea finala.

3. Testarea de performanță implică numai crearea de script-uri și orice schimbare a aplicației se poate reflecta prin modificarea script-urilor.

Procesul de testare este unul evolutiv. Deși scripting-ul este important, este doar una din componentele testării de performanță. Cea mai mare provocare pentru tester-ul de performanță este aceea de a determina ce tip de test este necesar și de a analiza indicatorii de performanță pentru a afla unde se afla bottle-neck-ul.[8]

5.5.6. Testarea de securitate

Aceasta testare se aplică pentru a determina dacă un produs software protejeaza datele siîși mentine funcționalitatea. Exista 6 concepte de baza legate de cuvantul „securitare" în software: confidentialitate, integritare, autentificare, disponibilitate, autorizare și non-repudiere.

Taxonomie

Termeni utilizati în securitare și explicatiile lor:

- *Descoperire*= are scopul de a identifica serviciile folosite de acel produs software. Nu are sensul de descoperire a vulnerabilitatilor, ci doar o detectare a versiunii, care evidentiaza versiuni depasite de software vulnerabile la atacuri noi.
- *Scanare de vulnerabilitate* = se cauta probleme cunoscute de securitate, folosind unelte automate pentru a crea conditiile pentru vulnerabilitatiile stiute. Nivelul de risc raportat este setat automat de unealta, fără verificăre manuala sau interpretare.
- *Estimarea vulnerabilitatiilor* = se folosește descoperirea și scanarea pentru a identifica vulnerabilitatile și de a plasa descoperirile în contextul mediului sub test. Exemplu: eliminarea pozitivelor false din raport și deciderea nivelului de risc pentru diecare descoperire.
- *Estimarea securitatii* = construita peste estimarea vulnerabilitatii prin adaugarea de verificăre manuala pentru confirmare. Nu include și exploatarea vulnerabilitatii. Verificărea are formă accesului autorizat pentru confirmarea setarilor și implică examinarea fisierelor log, răspunsului, mesajelor de eroare, coduri etc. O estimare a securitatii vrea să obtina o acoperire cat mai mare a sistemelor testate, dar nu intra în specificul unei anumite vulnerabilitati.
- *Testarea de penetrare*: simuleaza un atac al unui program rau intentionat. Implică exploatarea vulnerabilitatilor gasite pentru a obtine acces în continuare la aplicație. Astfel tester-ii inteleg cum poate un atacator să castige acces la informații confidentiale, să afecteze integritatea datelor sau disponibilitatea unui serviciu, și impactul actiunilor lui.

Confidentialitate inseamna ca datele și procesele trebuie protejate de publicare neautorizata.

Integritatea este cerinta ca datele și procesele să fie protejate de modificare neautorizata.este o masura pentru asigurarea corectitudinii informației. Schemele de integritate folosesc de obicei aceleasi tehnologii ca cele pentru confidentialitate, dar implică adaugarea de informații la o comunicare pentru a formă baza unei verificări algoritmice, în locul codarii întregii comuncari.

Autentificarea se refera la confirmarea identitatii unei persoane, urmarirea sursei de provenienta a unei informații sau asigurarea ca o masina sau un program este de incredere.

Non-repudierea consta în asigurarea ca un mesaj transferat a fost trimis și receptionat de participantii la comunicare (si nu de altcineva). Este o cale de a garanta ca emitatorul unui mesaj nu poate nega emiterea acelui mesaj și receptorul nu poate nega receptia mesajului.

Disponibilitatea este cerinta ca datele și procesele să fie protejate de atacuri de tipul refuz de acces pentru utilizatorii autorizati.[8]

5.5.7. Testarea de utilizabilitate

Aceasta testare este o tehnică de evaluare a produsului prin testarea lui pe utilizatori. Ofera informații directe despre cum folosesc utilizatorii reali aplicația.

Testarea de utilizabilitate se concentreaza pe masurarea capacitatii produsului de a indeplinii scopurile pentru căre a fost creat, si, mai precis, usurinta în utilizare

Se poate spune ca primul test de utilizabilitate a avut loc în anii ’40 și a fost efectuat de Henry Dreyfuss, care a proiectat cabinele pentru două vase de croaziera (Independence și Constitution). Mai intai a construit cateva prototipuri și a adus mai multe persoane să locuiasca în ele. Astfel proiectantii si-au dat seama ca trebuie să puna intrerupatoarele pentru iluminat langa pat, ca oamenii să nu se lovească în intuneric de alte obiecte, ca trebuie să lase spatiu pentru bagajele mari și alte astfel de lucruri care par marunte.

Testarea de utilizabilitate este o metodă de tip black-box. Scopul este acela de a observa cum folosesc oamenii produsul, unde apar erori, și cum se pot remedia acestea. Aceasta testare implică de obicei cum raspund subiectii în 4 zone de interes: eficientă, precizie, amintiri și răspuns emotional. Rezultatele primului test sunt folosite ca baza, sau masuratoare de control. Celelalte teste pot fi comparate cu baza pentru a indica o imbunatatire.

- Eficientă: cat dureaza și cati pasi sunt necesari ca utilizatorul să indeplineasca o anumita sarcina;
- Precizie: cate greseli fac oamenii, și cu ce consecinte
- Amintiri: cat de usorîșiaminteste utilizatorul aplicația, dupa o perioada în care nu a avut contact cu ea
- Raspuns emotional: ce senzatii îi să utilizatorului indeplinirea unei sarcini (stres, usurare, incredere)

Pregătirea unui test de utilizabilitate implică proiectarea atenta a unui scenariu sau a unei situatii reaslistice, în care persoana executa o lista de sarcini folosind produsul de testat timp ce observatorii iau notite. Alte instrumente de testare, ca instrucțiuni din script-uri, prototipuri de pe hartie, și chestionare pre/post testare, sunt folosite pentru a obtine feedback în legatura cu produsul testat. Scopul este acela de a observa interactiunea cu utilizatorul, ce îi place și ce il deranjeaza.

Testarea prin „metodă holului" este una din tehnici. În locul unei echipe angajate, specializate de testeri, sunt implicăti doar 5 sau 6 oameni, alesi aleator, care să simuleze utilizatorul final. De aici vine și numele metodei, adică se iau niste oameni la intamplare de pe hol. Se folosesc oameni care nu au legatura cuproiectul, care nu stiu detaliile interioare ale acestuia. Este eficientă mai ales în stadiile timpurii ale proiectării design-ului, cand designerii cauta impasuri posibile în care se pot afla utilizatorii.

În situatii în care evaluatorii, dezvoltatorii și utilizatorii se afla în locatii diferite (alte tari și alte fusuri orare), realizarea unui test în conditii de laborator poate fi o provocare serioasa. Astfel se ajunge și la evaluarea utilizabilitatii a distanta, utilizatorii și evaluatorii aflandu-se la mare departare unii de altii (atat spatial cat și temporal, adică fusuri orare diferite). Testarea la distanta poate fi sincrona sau asincrona. Metodă sincrona implică video-conferinte și alte unelte de comunicare în timp real. În cazul metodei asincrone, evaluatorul so tester-ul lucreaza separat. Metodele asincrone includ urmarirea automata a click-urilor, fisiere log pentru incidente critice și feedback subiectiv. Ca un studiu de laborator, un test de utilizabilitate la distanta se bazeaza pe sarcini, și platformă permite captura de click-uri și de alte activitati. Acest tip de test are loc în mediul utilizatorului, fapt care ajuta și mai mult la simularea folosirii reale.

O alta metodă de testare a utilizabilitatii este raportul expert. Se aduc experti cu cunoștiinte extinse în domeniul utilizarii unui produs.

Rapoartele expert pot fi făcute și automat, dar folosind programe carora li s-au impus anumite reguli pentru un design corect. Deși nu ofera atatea detalii ca un raport făcut de om, este mai rapid și mai ieftin.

Este important și numarul de oameni care testează. În 1990, Jakob Nielsen propunea folosirea unui numar mare de teste, dar cu grupuri mici (de maxim 5 oameni). El considera ca dacă 2-3 oameni nu se descurca usor intr-o anumita interfata grafica, nu are sens să observe și pe altii facand acelasi lucru. Argumentul lui a prmit și un suport matematic:

$U = 1 - (1 – p)^{n}$

U= numarul de probleme descoperite

P= probabilitatea ca un utilizator să decopere o problema

N= numarul de utilizatori

Însă exista și dezavantaje ale metodei lui Nielsen:

- Din moment ce utilizabilitatea este legata de un set mic de testeri, un esantion asa de mic nu este reprezentativ pentru intreaga populatie, deci rezultatele vor fi mai

mult legate de esantionul testat decat de populatia pe care o reprezinta (legea numerelor mari)

- Problemele de utilizabilitate nu sunt toate la fel de usor de descoperit, deci cele ascunse pot incetini procesul.

Nielsen nu se oprea la un singur test cu cei 5 utilizatori. Dupa testare și decoperirea problemelor, acestea erau remediate pe loc și se faca un nou test cu alt grup. Astfel a observat ca obtinea rezultate mai bune decat dacă facea un singur test cu 10 oameni. În practica, testele se fac odata sau de 2 ori pe saptamana în timpul dezvoltarii, folosind grupuri de 3-5 oameni, iar rezultatele se livreaza în cursul a 24 de ore designer-ilor. Se poate ajunge la un numar total de 50 pânăla 100 de subiecti care au testat programul în timplu ciclului de dezvoltare.

În primele etape, în care probabilitatea de a descoperi probleme serioase este mai mare, se pot folosi subiecti cu orice grad de inteligenta. În urmatoarele etape, se aleg subiectii cu abilitati intr-o gama mai larga. S-a bservat ca cei cu experienta nu au avut deloc problema la nici unul din teste, în timp ce incepatorii intalneau mai multe obstacole. În urmatoarele etape, se recruteaza testeri din populatia tinta (carora le este destinat produsul software).[8]

5.6.Concluzii

Dupa cum se poate vedea, testarea software reprezintă o investigație dirijată în scopul de a obține informații legate de calitatea produsului software, prin rularea programului în scopul de a descoperi bug-uri software. Este o etapă puternic dependentă de contextul operațional în care programul va fi folosit.

Testarea nu ofera garantia că produsul funcționează corect în orice condiții ci numai în acele condiții pentru care a fost testat. Astfel alegerea dintre modalitatile de testare folosite este esențială pentru acoperirea unei game cât mai largi de posibile erori.

Informația obținută în urma testării este esențială pentru remedierea defectului respectiv. În momentul în care se detectează o eroare, este posibil să nu se cunoască și cauza care a generat acea eroare. Programatorul este cel care trebuie ca după detecția erorii să identifice pe baza acesteia instrucțiunea sau secvența de instrucțiuni care a dus la apariția erori.

Numai dupa remedierea defectelor se poate trece la etapa urmatoare de dezvoltare a produsului software.

Bibliografie

- [1] Elfriede Dustin: Effective Software Testing - *50 Specific Ways to Improve Your Testing*

- [2] Paul Ammann, Jeff Offutt: *Introduction To software testing*
- [3] Dan Pilone, Russ Miles: *Head First Software Development*
- [4] Mauro Pezzand, Michael Young: *Software testing and Analisys: Process, Principles and Techniques*
- [5] Kshirasagar Naik, Priyadashi Tripathy: *Software Testing and Quality Assurance*
- [6] Gerald D. Everett, Raymond McLeod Jr: *Software Testing: Testing Across the Entire Software Development Life Cycle*
- [7] Glenford J. Myers: The *Art Of Software Testing, 2nd Edition* (2004)
- [8] www.wikipedia.com
- [9] www.informit.com
- [10] www.calsoftlabs.com
- [11] www.searchsoftwarequality.techtarget.com
- [12]www.integrant.com
- [13] www.nresult.com

IV. Verificarea și validarea produselor software

Tică Andra Maria

1. Introducere

Prin activităţile de verificare și validare se urmărește ca software-ul să satisfacă specificaţiile sale in timpul fiecărei faze a ciclului său de dezvoltare. Se asigură faptul ca fiecare articol software să fie verificat de o persoană diferită de aceea care l-a produs și că efortul de verificare și validare este adecvat pentru ca fiecare articol software să fie operaţional.

Obiectivul activităţilor de verificare și validare este de a reduce erorile software la un nivel acceptabil. Efortul necesar poate reprezenta 30-90% din totalul resurselor proiectului, in functie de complexitatea și gradul de risc al funcţionării necorespunzătoare a software-ului. Organizarea activităţilor de verificare și validare este inclusă in activităţile de management ale proiectului software.

Verificarea asigură că produsul este construit in concordanţă cu cerinţele, specificaţiile și condiţiile stabilite in etapele de proiectare și standardele in vigoare la momentul punerii pe piaţă a acestuia. Este un proces de control al calităţii folosit la

evaluarea conformităţii și se desfasoară concomitent cu dezvoltarea software-ului respectiv, fiind de cele mai multe ori un proces intern.[1],[9]

Verificarea inseamnă stabilirea și documentarea faptului că articolele referitoare la software, procesele și serviciile sunt in conformitate cu cerinţele specificate.

Validarea asigură utilizabilitatea produsului pe piaţa. Este procesul de a evalua un sistem sau o componentă in timpul sau la sfârșitul procesului de dezvoltare pentru a determina dacă satisface cerinţele specificate. Validarea este un proces extern de asigurare a calităţii. [1],[9]

Diferenţa dintre verificare și validare este importantă numai pentru teoreticieni. In practică, verificarea și validarea se referă la toate activităţile care asigură că software-ul va funcţiona conform cerinţelor. [1]

Validarea poate fi clasificată astfel:

- Validare in perspectivă – activitaţile desfășurate inainte ca noi articole să fie lansate pentru a se asigura caracteristicile de interes care intrunesc standarde de funcţionare corecte si sigure.
- Validare retrospectivă – proces pentru articolele deja in uz, distribuţie sau producţie. Se reia evaluarea specificaţiilor și așteptărilor de la produs, iar daca lipsesc date se sistează etapa curentă. Acest proces se realizează dacă se constată lipsa realizării unei validări de perspectivă, la schimbarea unor legislaţii sau standarde, la repunerea pe piaţă a unor produse anterior excluse.
- Validare curentă – are loc concomitent cu procesul de proiectare/dezvoltare.[1]

Activităţile de verificare includ recenzii si verificări formale.

Scopurile verificării sunt acelea de a demonstra ca programul este corect (dacă metoda și programul permite) și de a găsi erori. Verificarea este consistentă dacă un sistem raportat corect se dovedește a fi corect și completă, dacă prin metoda respectivă se poate determina corectitudinea fiecărui sistem. Detecţia de erori este consistentă când erorile raportante sunt reale și completă când s-au găsit toate erorile. [2]

Metodele de verificare pot fi:

- Statice, când se verifică fără execuţia codului și aici se inscriu analiza de fluxuri de date si verificarea formală.
- Dinamice, când se execută codul: rulare pe mașina virtuală, execuţia simbolică.

2. **Recenziile (Reviews)** de software sunt reprezentate de intâlniri pe parcursul cărora un produs este examinat de personalul care s-a ocupat de proiect, manageri, utilizatori, clienţi sau alte părţi interesate să comenteze sau să aprobe. [3]

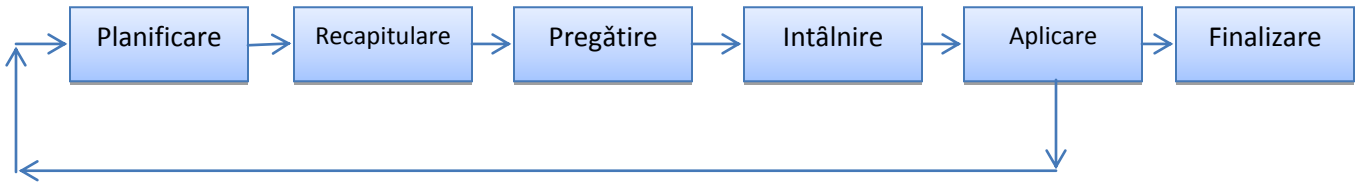
Clasificare:

- Recenzii de la egal la egal, realizate de autorul proiectului sau de mai mulţi colegi din echipa respectivă, cu scopul evaluării conţinutului tehnic și al calităţii muncii depuse.
  - Recenzia codului – examinarea sistematică a codului sursă.
  - Programare in echipă – doi programatori dezvoltă impreună un produs software la aceeași staţie de lucru.
  - Inspecţii – se urmărește o procedură strictă de depistare a defectelor.
  - Walkthrough – autorii organizează pentru toate parţile interesate o prezentare a proiectului, participanţii având oportunitatea să afle răspunsuri, să sesizeze defecte sau neconcordanţe.
  - Recenzii tehnice – o echipa formată din personal calificat examinează dacă produsul este adecvat pentru uzul destinat și identifică discrepanţe cu standarde și specificaţii.
- Recenzii la nivel de management, realizate de reprezentanţi ai nivelului managerial al intreprinderii pentru a evalua activitatea curentă și pentru a lua decizii referitor la desfășurarea următoarelor etape ale proiectului.
- Audituri, realizate de personal din exteriorul proiectului pentru a evalua conformitatea produsului cu specificaţiile, standardele și acordurile contractuale iniţiale.[3]

Procedura IEEE 1028, [3], pentru formalizarea recenziilor definește un set de activităţi bazate pe procesul de inspecţie al lui Michael Fagan, dezvoltat pentru prima dată la IBM. Pașii următori sunt obligatorii:

0. Coordonatorul recenziei folosește o listă standard de criterii de intrare care asigură condiţiile optime pentru succesul intregului proces.
1. Organizarea: invitarea personalului, informarea acestuia referitor la topic-ul review-ului, a locaţiei și a orei, asigurarea materialelor și dispozitivelor pentru buna desfașurare a intâlnirii.
2. Planificarea: coordonatorul identifică și confirmă obiectivele intâlnirii, organizează o echipă care să realizeze discuţia.
3. Recapitularea procedurilor: asigurarea faptului că toţi participanţii inţeleg scopul acestei dezbateri.

4. Pregătirea individuală: fiecare parte a echipei se pregătește pentru dezbaterea de grup, examinând atent potenţialele defecte luate in discuţie.
5. Examinarea de grup: are loc intâlnirea propriu-zisă cu toţi participanţii unde sunt expuse rezultatele individuale cu scopul ajungerii la un consens.
6. Aplicarea deciziilor: autorul produsului impreună cu echipa sa repară defectele pentru a satisface cerinţele exprimate in timpul intâlnirii. Se verifică satisfacerea tuturor cerinţelor.
7. Sfârșitul evaluării: coordonatorul verifică finalizarea cu succes a intregului proces.



Avantajul organizării acestei recenzii este că se pot identifica probleme mult mai ieftin decât dacă identificarea acestora s-ar face in etapa de testare (proces de detectare a erorilor). Mai mult, ele oferă oportunitatea autorilor tehnici de a invăţa să dezvolte documentaţie cu o marjă de eroare foarte redusă, putând astfel să identifice din timp și să renunţe la aspecte ale proiectului care pot duce la detectarea ulterioară de defecte. Vorbim aici de un proces de prevenire a apariţiei defectelor.[3]

Recenziile de cod constau in verificarea codurilor și inlăturarea defectelor de către membrii echipei proiectului. Un defect este reprezentat de un bloc de cod care nu este implementat conform cerinţelor, care nu funţionează după intenţia programatorului sau care nu este incorect dar poate fi perfecţionat. Pe lângă faptul că inlătură bug-urile, această tehnică este de asemenea benefică pentru dezvoltatorii incepători care pot invăţa in aceasta manieră noi modalităţi de programare.[3]

Inspecţiile sunt cele mai folosite in proiectele software. Scopul inspectorilor este ca aceștia să ajungă impreună la un consens asupra unui produs și să aprobe utilizarea acestuia in cadrul proiectului. Scopul principal al inspecţiei este identificarea defectelor. [4]

Spre deosebire de restul recenziilor, walkthrough-urile au ca obiective familiarizarea audienţei cu conţinutul proiectului și obţinerea de răspunsuri despre calitatea tehnică și despre conţinuntul documentaţiei acestuia. [5]

3. **Verificările formale** sunt modalităţile prin care se aprobă sau dezaprobă corectitudinea algoritmilor care stau la baza unui sistem pe baza unor specificaţii sau proprietăţi, folosind metode formale, matematice. Deoarece sistemul este modelat matematic, sunt posibile rezultate garantate in limitele posibilităţilor/presupunerilor de modelare. [8]

Verificările formale sunt foarte folositoare la demonstrarea corectitudinii sistemelor software exprimate prin codul lor sursă. Acest tip de verificare constă in furnizarea unei dovezi formale asupra corectitudinii algoritmului pe baza modelării matematice abstracte a sistemului, corespondenţa dintre modelul matematic și natura sistemului fiind cunoscute din etapa conceptuală.

Verificarea formală pentru software cunoaște mai multe abordări:

- Abordarea tradiţionala a anilor `70 când codul este mai intâi scris si apoi dovedit a fi corect intr-o etapa separată.
- Programare scrisă in mod dependent, când scrierea codurilor se face concomitent cu verificarea corectitudinii acestora. [8]

3.1. Model checking

Metoda **model checking** constă in explorarea sistematică si exhaustivă a modelului matematic. Acest lucru este posibil atât pentru modelele finite cât și pentru cele infinite unde seturile infinite de stări pot fi reprezenate in mod finit prin abstractizare. De obicei această metodă constă in explorararea tuturor stărilor și tranziţiilor din model, utilizând tehnici de abstractizare pe domenii, care consideră grupuri intregi de stări intr-o singura operaţie, reducând astfel timpul de calcul. Implementarea tehnicilor include enumerarea spaţiului de stări, a spaţiului simbolic de stări, interpretarea abstractă, simularea simbolică și rafinarea abstractă. Proprietăţile care vor fi verificate sunt descrise in logica temporală, precum LTL- linear temporal logic si CTL- computational tree logic. [9]

Această metodă rezolvă urmatoarea problemă: dându-se modelul unui sistem, se verifică dacă acesta indeplinește o anume specificaţie dată. Pentru a rezolva această problemă dupa un algoritm, modelul sistemului și specificaţiile acestuia vor fi formulate intr-un limbaj matematic precis: se verifică dacă o structură dată satisface o formulă logică dată.

Model checking-ul este un automat cu stări finite. Spre deosebire de verificarea prin demonstrare de teoreme, această metoda nu necesită o anotare a programului de către utilizator, verificarea făcându-se automat pentru toate execuţiile posibile, in caz de eroare generându-se un contraexemplu.

Se urmează următorii pași:

- Se face specificarea proprietăţilor și traducerea ei in limbajul C. Programul original este instrumentat.
- Deoarece programul poate fi complex, se folosește abstracţia in verificare, adică se dorește concentrarea asupra porţiunii relevante din program. Acest lucru se realizează prin tehnica program slicing - se determină fragmentul de program care afectează o anumita proprietate a programului.
- Se generează programul boolean pornind de la predicatele din specificaţie.
- Se analizează programul boolean: se calculează mulţimea stărilor atinse.
- Dacă se depisteaza erori se vor genera contraexemple și noi predicate:
    - Dacă contraexemplul este realizabil, atunci eroarea este reală.
    - Altfel, se pleacă din nou din starea de eroare in programul abstract și se parcurge calea de eroare inapoi până când se găsește o inconsistenţă. După aceea se generează predicatele corespunzătoare. Se regenerează programul boolean și se reia verificarea.

3.2. Inferenţa logică

O altă abordare este reprezentată de **inferenţa logică** care constă in folosirea unei versiuni formale a unui raţionament matematic. Se folosesc de obicei teoreme precum HOL, ACL2, Isabelle, Coq. Aceste teoreme sunt aplicate parţial automat și sunt conduse de inţelgerea de către utilizator a sistemului de validat. Instrumentele mai noi precum Perfect Developer și Escher C Verifier incearcă să automatizeze complet procesul. Logica liniară și cea temporală poate de asemenea fi folosită la inferenţa logică și nu numai in cazul model checking-ului.

3.3. Derivarea de program

O alta metoda este reprezentata de **derivarea de program,** când producerea de cod eficient se face plecând de la specificaţiile funcţionale și se urmează o serie de pași cu rolul de a conserva corectitudinea. Formalismul Bird-Meertens respectă aceste principii, fiind considerat ca o altă formă de verificare concomitent cu construirea codului.

3.4. Verificarea prin demonstrare de teoreme se realizează prin folosirea unui demonstrator de teoreme, plecând de la anumite condiţii de verificare. [10]

In anul 1967, in lucrarea sa intitulată „Atribuirea de sensuri programelor”, Robert W. Floyd spunea că „o bază adecvată pentru definirea formală a sensurilor programelor trebuie sa fie in așa fel incât să stabilească un standard riguros de dovezi”. Mai mult, el pleacă de la ideea că „dacă valorile iniţiale ale variabilelor programului satisfac relaţia R1, atunci valorile finale vor satisface relaţia R2”. Astfel, scopul lucrării sale era să formalizeze semantica limbajelor de programare. [8],[10]

Lucrarea lui Floyd dezvoltă reguli generale pentru combinarea condiţiilor de verificare și reguli specifice pentru diferitele tipuri de instrucţiuni. Se introduc invarianţii pentru raţionamentele despre cicluri și se tratează terminarea folosind o masură pozitivă de descărcare.

Metoda introdusa de Floyd poartă numele de **metoda anotării unui program cu aserţiuni**:

- Condiţia de verificare: se consideră o formula Vc(P,Q) astfel incât dacă P este adevărat inainte de a se executa c, atunci Q este adevărat la ieșire.
- Cea mai verificabilă consecinţă: avem un program și consideram o condiţie iniţială. După execuţia programului aceasta va fi cea mai puternică proprietate valabilă.

Aceste aserţiuni au fost exprimate in logica predicatelor de ordinul I.

Mai târziu, in anul 1969, Floyd a conceput impreuna cu Hoare ceea ce se numeste **logica Floyd-Hoare sau regulile lui Hoare**. Acesta este un sistem formal cu un set de reguli logice pentru rationarea riguroasă asupra corectitudinii programelor. Ei au avut ca punct de plecare lucrarea lui Floyd. [6],[10]

In această lucrare au tratat precondiţii și postcondiţii pentru execuţia unei instrucţiuni, folosind notaţia de triplet Hoare care pune mai clar in evidenţă relaţia dintre instrucţiune și cele doua aserţiuni. De această dată au lucrat cu programe textuale și nu cu scheme logice.

Cele doua notaţii de triplet Hoare:

- ✓ Corectitudinea parţială: {P} S {Q} – dacă S este executat intr-o stare care satisface P și execuţia lui S se termină, atunci starea rezultantă satisface Q.

- ✓ Corectitudinea totală: [P] S [Q] – daca S este executat intr-o stare care satisface P , atunci execuţia lui S se termină și starea rezutantă satisface Q.

Regulile lui Hoare sunt definite pentru fiecare tip de instrucţiune in parte, prin combinaţia lor putându-se raţiona programe intregi:

- ➢ Axioma declaraţiei goale: $\overline{\{P\}sare\ peste\{P\}}$- declaraţia „sare peste” nu schimbă starea programului așa că tot ceea ce este adevărat inainte de „sare peste” va fi adevarat și după.
- ➢ Axioma pentru atribuire: $\overline{\{P[E/x]\}x := E\{P\}}$. Aici $P[E/x]$ denotă expresia P in care toate apariţiile variabilei x au fost inlocuite cu expresia E. Această axiomă afirmă faptul că adevărul lui $\{P[E/x]\}$ este echivalent cu adevărul dupa asignare al lui {P}. Astfel, când $\{P[E/x]\}$ este adevărat inainte de asignare, atunci {P} este adevărat după asignare și dacă $\{P[E/x]\}$ este fals inainte de asignare atunci {P} va fi de asemenea fals.

Ex: {x=y-2}x:=x+2{x=y} – in rezultat, x=y, substituim x cu expresia atribuită, x+2 și se obţine x+2=y de unde x=y-2.

Această axiomă nu se aplică atunci când mai multe nume referă aceeași valoare stocată.

Ex: {y=3}x:=2{y=3}.

- ➢ Regula de compunere sau de secvenţiere: $\frac{\{P\}S\{Q\},\{Q\}T\{R\}}{\{P\}S;T\{R\}}$ - se aplică programelor executate secvenţial S și T, unde S se execută inaintea lui T.

Ex: Pentru S: {x+1=43}y:=x+1{y=43} și T: {y=43}z:=y{z=43}, atunci {x+1=43}y:=x+1;z:=y{z=43}.

- ➢ Regula de decizie: $\frac{\{B \wedge P\}S\{Q\},\{\neg B \wedge P\}T\{Q\}}{\{P\}if\ B then\ S\ else T\ endif\ \{Q\}}$.
- ➢ Regula consecinţei: $\frac{P` \rightarrow P,\{P\}S\{Q\},Q \rightarrow Q`}{\{P`\}S\{Q`\}}$
- ➢ Regula ciclului cu test initial while: $\frac{\{P \wedge B\}S\{P\}}{\{P\}while\ B\ do\ S\ done\{\neg B \wedge P\}}$, unde P este testul iniţial.

- Regula ciclului cu test iniţial while pentru corectitudine totală: $\frac{< is\ well-founded, \{P \wedge B \wedge t = z\} S \{P \wedge t < z\}}{\{P\} while\ B\ do\ S\ done \{\neg B \wedge P\}}$.

In continuare, in 1975, E.W.Dijkstra a reformulat regulile lui Hoare in lucrarea sa intitulată „Guarded commands, non-determinancy and formal derivation of programs". Spre deosebire de Hoare și Floyd a căror logică este prezentată ca un sistem deductiv, semantica transformării bazata pe predicate a lui Dijkstra este o strategie completă pentru a reduce problema verificării tripletului Hoare la demonstrarea unei legi logice de ordinul intai (concluzionare pe baza unui predicat considerat adevărat). [7],[10]

Abordarea lui Dijkstra folosește **operatorul „weakest precondition"**. Astfel, pentru o instrucţiune S și o postcondiţie dată Q pot exista mai multe precondiţii P astfel incât {P} S {Q} sau [P] S [Q]. Dijkstra calculează o precondiţie necesară și sufiecientă wp(S,Q) pentru terminarea cu succes a lui S cu postcondiţia Q. Wp este un transformator de predicate care permite definirea unui calcul cu astfel de transformări. [7],[10]

Preconditiile lui Dijkstra:

- Salt peste (skip): wp(skip,R)=R.
- Renunţă la (abort): wp(abort,R)=false.
- Atribuire
  - Var1: wp(x:=E,R)= $\forall$ y, y=E =>R[x<-y], unde y este o nouă variabilă care stochează valoarea finală a variabilei x.
  - Var2: wp(x:=E,R)=R[x<-E].

  Prima variantă evită o potenţială replicare a lui E in R, pe când a doua variantă este mult mai simplă, deoarece avem o singură apariţie a lui x in R.

- Secvenţiere: wp(S1;S2,R)=wp(S1,wp(S2,R)).
- Condiţionare:
  $wp(if\ E\ then\ S1\ else\ S2\ else\ end, R) = (E \Rightarrow wp(S1, R)) \wedge (\neg E \Rightarrow wp(S2, R))$.
- Bucla while:
  $wp(while\ E\ do\ S\ done, R) = I$
  $\wedge \forall y, ((E \wedge I) \Rightarrow wp(S, I \wedge x < y))[x \leftarrow y]$
  $\wedge \forall y, ((\neg E \wedge I) \Rightarrow R)[x \leftarrow y]$

Y este un tuplu nou de variabile. Prima formulă inseamnă că invariantul I trebuie să ramână neschimbat. A doua inseamnă că S (corpul buclei while) trebuie să păstreze invariantul și să micșoreze variantul: aici variabila y

reprezintă starea iniţiala a blocului de execuţie. Ultima formulă inseamnă că R trebuie să fie stabilit la finalul buclei while.

In concluzie, la verificarea prin demonstrare de teoreme se urmează pașii:

- Se scriu tripletele Hoare sau precondiţiile Dijkstra.
- Se verifică inlănţuirea acestora cu un demonstrator de teoreme sau cu o procedură de decizie.

4. Concluzii [8],[11]

Metodele formale fac posibile inţelegerea și punerea in discuţie a problemelor, descoperirea contradicţiilor, prin notaţii clare și precise.

Ele sunt o modalitate prin care se verifică părţi de o importanţă majoră din programe cu ajutorul unui studiu abstract, deoarece notaţiile matematice au o semantică precisă care poate fi inţeleasă intr-un context internaţional, inlăturând ambiguităţile.

5. Bibliografie

[1] http://en.wikipedia.org/wiki/Verification_and_validation

[2] http://en.wikipedia.org/wiki/Verification_and_Validation_(software)

[3] http://en.wikipedia.org/wiki/Software_review

[4] http://en.wikipedia.org/wiki/Software_inspection

[5] http://en.wikipedia.org/wiki/Software_walkthrough

[6] http://en.wikipedia.org/wiki/Hoare_triple

[7] http://en.wikipedia.org/wiki/Weakest_precondition

[8] http://en.wikipedia.org/wiki/Formal_verification

[9] http://en.wikipedia.org/wiki/Model_checking

[10] http://bigfoot.cs.upt.ro/~marius/curs/vvs/index.html

[11] http://se.inf.ethz.ch/old/teaching/ws2006/0273/slides/outsourcing_20_requirements.pdf

V. Concluzii

Fiecare etapă care are loc este puternic dependentă de etapele realizate anterior. Astfel anumite etape nu pot fi realizate decât în urma finalizăirii alteia (ex: Proiectarea nu poate să înceapă decat după finalizarea analizei). În plus nu există o divizare temporală exactă a acestora, ele putând să se suprapună (ex: Documentarea se face pentru fiecare etapă în parte). Astfel etapele nu pot fi tratate independent.

De asemenea, fiecare etapă trebuie să respecte anumite reguli pentru a putea fi considerată de calitate. Asigurarea calității duce la minimizarea complexității produsului și deci la o mai bună organizare a sa, oferind posibilitatea de a obține un produs performant.

Nerespecterea regulilor de calitate poate duce la realizarea unui produs neperformant, la realizarea lui la costuri mult prea ridicare sau chiar la nefinalizarea acestuia.