INGINERIE SOFTWARE

Suport de Curs

Octombrie 2023

Cuprins

1. Introducere

In acest curs sunt descrise produsele software la nivel basic, proiectarea și procesul de dezvoltare a software-ului, managementul proiectelor software și complexitatea proiectării. La sfârșitul acestui curs conceput pentru studentii Facultatii de Informatica din Universitatea Titu Maiorescu ar trebui să detineti o bună înțelegere a conceptelor de inginerie software.

2. Condiții prealabile

Acest material este conceput și dezvoltat pentru începători. Cu toate acestea, conștientizarea cu privire la sistemele software, procesul de dezvoltare software și fundamentele arhitecturii computerului este benefică.

3. Prezentare generală a software-ului

Pentru a intelege ce înseamnă ingineria software observam ca termenul este format din două cuvinte, software și inginerie.

Software-ul este mai mult decât un cod de program. Un program este un cod executabil, care servește unor scopuri de calcul. Software-ul este considerat a fi o colecție de cod de programare executabil, biblioteci asociate și documentații.

Software-ul, atunci când este creat pentru o cerință specifică, se numește *produs software*.

Pe de altă parte**, ingineria** se referă la dezvoltarea de produse, folosind principii și metode științifice bine definite.

Ingineria software este o ramură de inginerie asociată cu dezvoltarea produselor software folosind principii, metode și proceduri științifice bine definite. Rezultatul ingineriei software este un produs software eficient și fiabil.

3.1 Definiții

IEEE definește **ingineria software** ca:

(1) Aplicarea unei abordări sistematice, disciplinate și cuantificabile a dezvoltării, funcționării și întreținerii software-ului; adică aplicarea ingineriei la software.

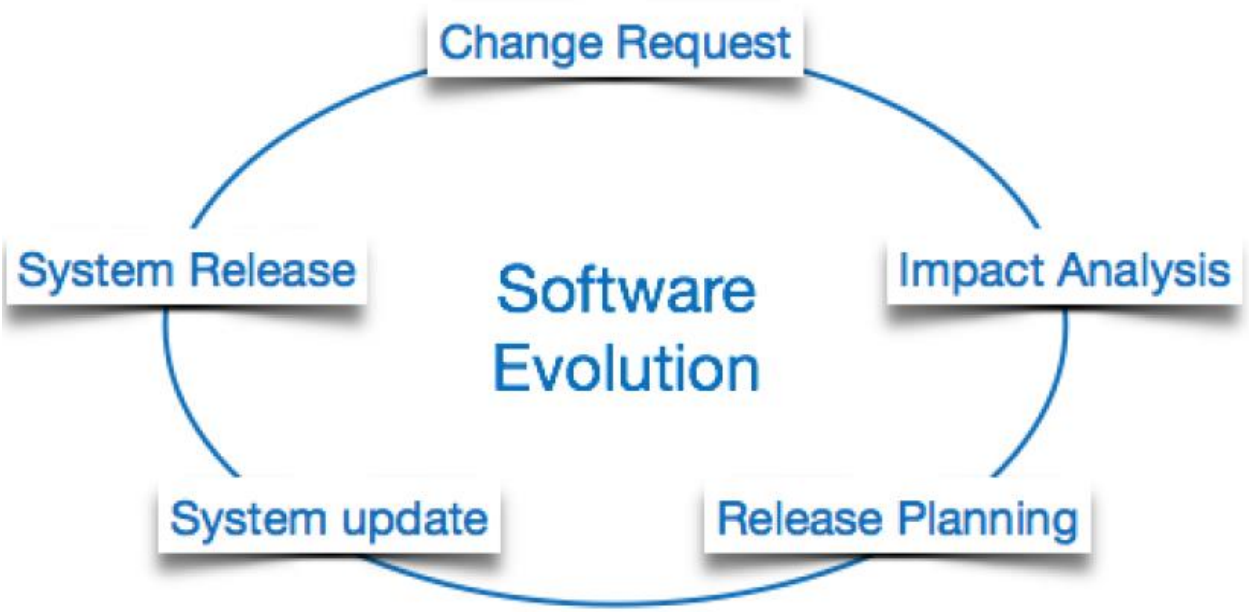
(2) Studiul abordărilor ca în afirmația de mai sus.

Fritz Bauer, un informatician german, definește ingineria software ca:

„Ingineria software este stabilirea și utilizarea unor principii inginerești solide pentru a obține un software economic care să fie fiabil și să funcționeze eficient pe mașini reale.”

3.2 Evoluția Software-ului

Procesul de dezvoltare a unui produs software utilizând principiile și metodele de inginerie software este denumit Software Evolution. Aceasta include dezvoltarea inițială a software-ului și întreținerea și actualizările acestuia, până când se dezvoltă produsul software dorit, care satisface cerințele așteptate.



Evoluția începe de la procesul de colectare a cerințelor. După care dezvoltatorii creează un prototip al software-ului dorit și îl arată utilizatorilor pentru a primi feedback-ul lor în faza incipientă a dezvoltării produsului software. Utilizatorii sugerează modificări, pe care se produc mai multe actualizări consecutive și de întreținere. Acest proces modifică software-ul original, până când software-ul dorit este realizat.

Chiar și după ce utilizatorul are software-ul dorit, tehnologia avansată și cerințele în schimbare obligă produsul software să se schimbe în consecință. Sa re-creați software-ul de la zero pentru a merge individual cu cerința este nefezabil. Singura soluție fezabilă și economică este actualizarea software-ului existent astfel încât să corespundă celor mai recente cerințe.

3.3 Legile privind evoluția software-ului

Lehman* a dat cateva legi pentru evoluția software-ului. El a împărțit software-ul în trei categorii diferite:

1. *Static-type* (***S***-type) - Acesta este un software care funcționează strict conform specificațiilor și soluțiilor definite. Soluția și metoda de realizare a acestuia sunt înțelese imediat înainte de codare. Software-ul de tip ***S*** este cel mai puțin supus modificărilor, prin urmare acesta este cel mai simplu dintre toate. De exemplu, un program pentru calcul numeric.

2. *Tip practic* (tip ***P***) - Acesta este un software cu o colecție de proceduri si este definit exact de ceea ce pot face procedurile. În acest software specificațiile pot fi descrise, dar soluția nu este evident imediata. De exemplu, software pentru jocuri.

3. *Tip încorporat* (tip ***E***) - Acest software funcționează aproape ca o cerință a mediului real, avand un grad ridicat de evoluție, deoarece există diverse schimbări în legi, impozite etc. în situațiile din lumea reală. De exemplu, software de tranzacționare online.

* Meir "Manny" Lehman, (24 ianuarie 1925 - 29 decembrie 2010) a fost profesor la School of Computing Science de la Universitatea Middlesex. Din 1972 până în 2002 a fost profesor și șef al departamentului de calcul la Imperial College London. Contribuțiile sale de cercetare includ descrierea timpurie a fenomenului de evoluție a software-ului și legile omonime ale lui Lehman privind evoluția software-ului.

3.4 Evoluția software-ului E-Type

Lehman a dat opt legi pentru evoluția software-ului ***E***-Type -

1. **Schimbare continuă** - Un sistem software de tip ***E*** trebuie să se adapteze în continuare la schimbările din lumea reală, altfel devine progresiv mai puțin util.

2. **Creșterea complexității** - Pe măsură ce un sistem software de tip E evoluează, complexitatea acestuia tinde să crească, cu excepția cazului în care se lucrează la întreținerea sau reducerea acestuia.

3. **Conservarea familiarității** - Familiarizarea cu software-ul sau cunoștințele despre modul în care a fost dezvoltat, de ce a fost dezvoltat în acel mod special etc., trebuie păstrată cu orice preț, pentru a implementa schimbările în sistem.

4. **Creștere continuă** - Pentru ca un sistem de tip ***E*** destinat să rezolve unele probleme de afaceri, dimensiunea sa de implementare a modificărilor crește în funcție de schimbările de stil de viață ale companiei.

5. **Reducerea calității** - Un sistem software de tip ***E*** scade în calitate, cu excepția cazului în care este întreținut riguros și adaptat la un mediu operațional în schimbare.

6. **Sisteme de feedback** - Sistemele software de tip ***E*** constituie sisteme de feedback cu mai multe bucle și mai multe niveluri și trebuie tratate ca atare pentru a fi modificate sau îmbunătățite cu succes.

7. **Autoreglare** - procesele de evoluție a sistemului de tip ***E*** se autoreglează cu distribuția produsului și măsurile de proces aproape de normal.

8. **Stabilitatea organizațională** - Rata medie efectivă de activitate globală într-un sistem de tip ***E*** în evoluție este invariantă pe durata de viață a produsului.

3.5 Paradigme software

Paradigmele software se referă la metodele și pașii care sunt luați în timpul proiectării software-ului. Există multe metode propuse care și sunt implementate dar trebuie să vedem unde se află aceste paradigme în conceptul de inginerie software. Acestea pot fi combinate în diferite categorii, deși sunt conținute una în alta:

Paradigma de programare este un subset al paradigmei de proiectare software, care este la randul ei un subset al paradigmei de dezvoltare software.

3.6 Paradigma de Dezvoltare Software

Această paradigmă este cunoscută sub numele de **paradigma de inginerie software**, unde sunt aplicate toate conceptele de inginerie referitoare la dezvoltarea de software. Include diverse cercetări și colectarea cerințelor, care ajută la dezvoltarea produsului software. Se compune din:

• Colectarea cerințelor

• Proiectare software

• Programare

3.7 Paradigma de Proiectare Software

Această paradigmă face parte din dezvoltarea de software și include -

• Proiectare

• Întreținere

• Programare

3.8 Paradigma de Programare

Această paradigmă este strâns legată de aspectul programării dezvoltării de software. Aceasta include -

• Codificare

• Testarea

• Integrare

3.9 Nevoia de Inginerie Software

Necesitatea ingineriei software apare din cauza ratei mai mari de modificare a cerințelor utilizatorilor și a mediului in care lucrează software-ul. Iata câteva dintre nevoile menționate:

• **Software mare** - Este mai ușor să construiți un perete decât o casă sau o clădire, la fel, deoarece dimensiunea software-ului devine mare, ingineria trebuie să facă un pas pentru a-i oferi un proces științific.

• **Scalabilitate** - În cazul în care procesul software nu ar fi bazat pe concepte științifice și inginerești, ar fi mai ușor să recreezi un nou software decât să-l scalezi pe unul existent.

• **Cost** - Industria hardware și-a arătat abilitățile și producția uriașă a scăzut prețul hardware-ului pentru computer și electronice. Dar, costul software-ului rămâne ridicat dacă procesul adecvat nu este adaptat.

• **Natura dinamică** - Întotdeauna creșterea și adaptarea naturii software-ului depind enorm de mediul în care lucrează utilizatorul. Dacă natura software-ului se schimbă, întotdeauna trebuie făcute noi îmbunătățiri în cel existent. Acesta este locul în care ingineria software joacă un rol esential.

• **Managementul calității** - Un proces mai bun de dezvoltare software oferă produse software mai bune și de calitate.

3.10 Caracteristicile unui Software

Un produs software poate fi judecat după ce oferă și cât de bine poate fi utilizat.

Acest software trebuie să satisfacă următoarele cerinte de baza:

• Operațional

• Tranzițional

• Usor de intreținut

Este de așteaptat ca software-ul bine conceput și creat să aibă următoarele caracteristici:

Operațional

Acest lucru ne spune cât de bine funcționează software-ul în operațiuni. Poate fi măsurat de:

• Buget

• Utilitate

• Eficiență

• Corectitudine

• Funcționalitate

• Fiabilitate

• Securitate

• Siguranță

Tranzitional

Acest aspect este important atunci când software-ul este mutat de la o platformă la alta:

• Portabilitate

• Interoperabilitate

• Reutilizare

• Adaptabilitate

întreținere

Acest aspect reprezintă cât de bine software-ul are capacitățile de a se menține într-un mediu în continuă schimbare:

• Modularitate

• Mentenabilitate

• Flexibilitate

• Scalabilitate

Pe scurt, ingineria software este o ramură a informaticii, care folosește concepte de inginerie bine definite necesare pentru a produce produse software eficiente, durabile, scalabile, inbudget și la timp.

Ciclul de Viață al Dezvoltării Software-ului

Ciclul de viață al dezvoltării software-ului, CVDS pe scurt, este o secvență bine definită și structurată de etape în ingineria software pentru a dezvolta produsul software dorit.

3.11 Activități CVDS

CVDS oferă o serie de pași care trebuie urmați pentru proiectarea și dezvoltarea eficientă a unui produs software. Cadrul CVDS include următorii pași:

3.12 Comunicare

Acesta este primul pas în care utilizatorul inițiază cererea pentru un produs software dorit. Utilizatorul contactează furnizorul de servicii și încearcă să negocieze termenii, înaintează cererea către organizația care furnizează servicii, în scris.

3.13 Centarlizarea Cerințelor

Acesta este un pas înainte al echipei de dezvoltare software ce lucrează pentru a defini proiectul. Echipa poartă discuții cu diferiți actori din domeniul problemei și încearcă să aducă cât mai multe informații cu privire la cerințele lor. Cerințele sunt avute în vedere și separate în cerințele utilizatorului, cerințele de sistem și cerințele funcționale. Cerințele sunt colectate folosind o serie de practici, cum ar fi:

• studierea sistemului și a software-ului existent sau învechit,

• realizarea de interviuri cu utilizatorii și dezvoltatorii,

• referinte la baza de date sau

• colectarea răspunsurilor din chestionare.

3.14 Studiu de fezabilitate

După colectarea cerințelor, echipa vine cu un plan dur al procesului software. La acest pas, echipa analizează dacă un software poate fi proiectat pentru a îndeplini toate cerințele utilizatorului și dacă există posibilitatea ca software-ul să nu mai fie util. De asemenea, se analizează dacă proiectul este fezabil din punct de vedere financiar, practic și tehnologic pentru ca organizația să-l poată prelua. Există mulți algoritmi disponibili, care îi ajută pe dezvoltatori să concluzioneze fezabilitatea unui proiect software.

3.15 Analiza de sistem

La acest pas, dezvoltatorii decid o foaie de parcurs a planului lor și încearcă să aducă cel mai bun model de software potrivit pentru proiect. Analiza sistemului include înțelegerea limitărilor produselor software, problemele legate de sistemul de învățare sau modificările care trebuie făcute în prealabil în sistemele

existente, identificarea și abordarea impactului proiectului asupra organizației și personalului etc. Echipa proiectului analizează domeniul de aplicare al proiectului și planifică programul și în mod corespunzător resursele.

3.16 Proiectare software

Următorul pas este de a aduce cunoștințele complete despre cerințe și analize pe masa și de a proiecta produsul software. Informatiile de la utilizatori și cele din faza de colectare a cerințelor sunt intrările acestui pas. Rezultatul acestui pas vine sub forma a două modele; proiectare logică și proiectare fizică. Inginerii produc meta-date și dicționare de date, diagrame logice, diagrame de flux de date și, în unele cazuri, pseudo-coduri.

3.17 Codificare

Acest pas este, de asemenea, cunoscut sub numele de fază de programare. Implementarea proiectării software-ului începe în ceea ce privește scrierea codului de program în limbajul de programare adecvat și dezvoltarea eficientă a programelor executabile fără erori.

3.18 Testarea

O estimare spune că 50% din întregul proces de dezvoltare software ar trebui sa fie testarea. Erorile pot distruge software-ul de la un nivel critic până la eliminarea sa. Testarea software-ului se realizează în timp ce codarea este realizată de dezvoltatori, iar testarea amănunțită este efectuată de experți de testare la diferite niveluri de cod, cum ar fi testarea modulelor, testarea programului, testarea produsului, testarea internă și testarea produsului la finalul utilizatorului. Descoperirea timpurie a erorilor și remedierea lor este cheia unui software de încredere.

3.19 Integrare

Este posibil ca software-ul să fie necesar să fie integrat cu bibliotecile, bazele de date și alte programe. Această etapă a CVDS este implicată în integrarea software-ului cu entități din lumea exterioară.

3.20 Implementare

Aceasta înseamnă instalarea software-ului pe computerele utilizatorului. Uneori, software-ul are nevoie de configurații post-instalare la finalul utilizatorului. Software-ul este testat pentru portabilitate și adaptabilitate, iar problemele legate de integrare sunt rezolvate în timpul implementării.

3.21 Funcționare și întreținere

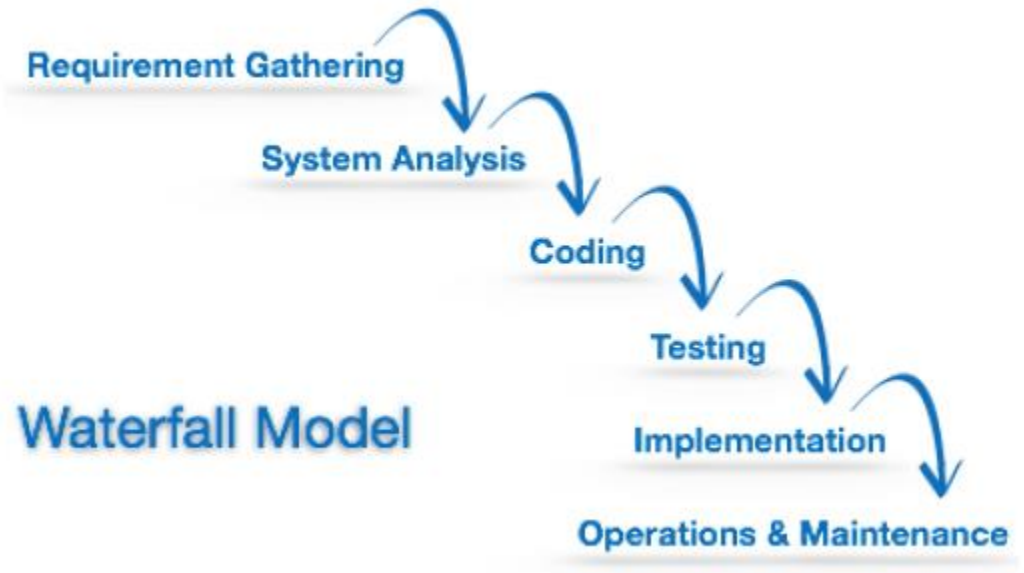
Această fază confirmă funcționarea software-ului în termeni de eficiență mai mare și cu mai puține erori. Dacă este necesar, utilizatorii sunt instruiți sau sunt ajutați cu privire la modul de operare a software-ului și modul de menținere a software-ului operațional. Software-ul este menținut în timp util prin actualizarea codului în funcție de schimbările care au loc în mediul final sau tehnologia utilizatorului. Această fază se poate confrunta cu provocări provocate de bug-uri ascunse și probleme neidentificate din lumea reală.

4. Paradigme de dezvoltare software

Paradigma de dezvoltare software ajută un dezvoltator să selecteze o strategie de dezvoltare a software-ului. O paradigmă de dezvoltare software are propriul set de instrumente, metode și proceduri, care sunt exprimate clar și definesc ciclul de viață al dezvoltării software-ului. Câteva dintre paradigmele de dezvoltare software sau modelele de proces sunt definite după cum urmează:

4.1 Modelul Waterfall

Modelul Waterfall sau cascada este cel mai simplu model de paradigmă de dezvoltare software. Toate fazele CVDS vor funcționa una după alta în mod liniar. Adică, atunci când prima fază este terminată, va începe doar a doua fază și așa mai departe.



Acest model presupune că totul se desfășoară perfect dupa cum a fost planificat în etapa anterioară și nu este nevoie să ne gândim la problemele trecute care pot apărea în faza următoare. Acest model nu funcționează corect dacă mai sunt câteva probleme lasate nerezolvate la pasul anterior. Natura secvențială a modelului nu ne permite să ne întoarcem și să anulăm sau să refacem acțiunile noastre.

Acest model este cel mai potrivit atunci când dezvoltatorii au proiectat și dezvoltat deja software similar în trecut și sunt conștienți de toate domeniile sale.

4.2 Modelul AGILE

Dezvoltare Agile a Software-ului

În dezvoltarea de software , Agile (uneori scrise Agile) este un set de practici menite să îmbunătățească eficiența profesioniștilor în dezvoltarea de software, a echipelor și a organizațiilor. Aceasta implică descoperirea cerințelor și dezvoltarea de soluții prin efortul de colaborare de auto-organizare și eco-funcționale ale echipelor și clientul/utilizator final al lor. Acesta pledează pentru planificarea adaptivă, dezvoltarea evolutiva, livrarea timpurie și îmbunătățire continuă și încurajează răspunsuri flexibile la schimbările cerințelor, disponibilitatea resurselor și înțelegerea problemelor care trebuie rezolvate.

Valorile și principiile expuse în Manifestul Agile din 2001 au fost derivate și fundamentează o gamă largă de cadre de dezvoltare software , inclusiv Scrum și Kanban .

Deși există multe dovezi declarative conform cărora adoptarea unor practici și valori Agile îmbunătățește eficiența profesioniștilor în software, a echipelor și a organizațiilor, dovezile empirice sunt relativ mixte și greu de găsit.

Manifestul pentru dezvoltarea Agile a software-ului

Valori de dezvoltare software Agile

Pe baza experienței lor combinate de dezvoltare a software-ului și de a-i ajuta pe alții să facă acest lucru, cei șaptesprezece semnatari ai manifestului au proclamat valorile ca fiind:

• Indivizi și interacțiuni deasupra/mai mult decat procesele și instrumentele

• Software de lucru mai mult decat documentația cuprinzătoare

• Colaborarea clienților mai mult decat negocierea contractului

• Răspunsul la schimbarea mai mult decat urmarirea unui plan.

Adică articolele din stânga sunt evaluate mai bine decât articolele din dreapta. Nu înseamnă că articolele din dreapta ar trebui excluse în favoarea articolelor din stânga. Ambele părți au valoare, dar din punct de vedere al dezvoltării Agile, autorii manifestului înclină în favoarea articolelor din stânga.

După cum a explicat Scott Ambler:

• Instrumentele și procesele sunt importante, dar este mai important să existe oameni competenți care să lucreze împreună eficient.

• O documentație bună este utilă pentru a ajuta oamenii să înțeleagă cum este construit software-ul și cum să-l folosească, dar principalul punct de dezvoltare este crearea de software, nu de documentare.

• Un contract este important, dar nu este un substitut pentru colaborarea strânsă cu clienții pentru a descoperi de ce au nevoie.

• Un plan de proiect este important, dar nu trebuie să fie prea rigid pentru a se adapta schimbărilor tehnologice sau de mediu, priorităților părților interesate și înțelegerii oamenilor a problemei și a soluției sale.

Unii dintre autori au format Agile Alliance, o organizație non-profit care promovează dezvoltarea de software în conformitate cu valorile și principiile manifestului. Prezentând manifestul în numele Alianței Agile, Jim Highsmith a declarat:

Mișcarea Agile nu este anti-metodologie, de fapt mulți dintre noi dorim să restabilim credibilitatea cuvântului metodologie. Vrem să restabilim un echilibru. Îmbrățișăm modelarea, dar nu pentru a înregistra o diagramă într-un depozit corporativ prăfuit. Îmbrățișăm documentația, dar nu sute de pagini de tomuri care nu sunt întreținute niciodată și rareori folosite. Planificăm, dar recunoaștem limitele planificării într-un mediu turbulent. Cei care i-ar clasifica pe susținătorii XP sau SCRUM sau ai oricărei alte metodologii Agile drept „hackeri” nu știu atât metodologiile, cât și definiția originală a termenului de hacker. [Jim Highsmith, History: The Agile Manifesto]

Principii de dezvoltare software Agile

Manifestul pentru Agile Software Development se bazează pe douăsprezece principii:

• Satisfacția clienților prin livrarea timpurie și continuă a unui software apreciat.

• Sunt bine venite schimbarea cerințelor, chiar și în cursul dezvoltării intârziate.

• Livrați software-ul frecvent (mai degrabă săptămânal decât lunar)

• Cooperare strânsă, zilnică, între oamenii din business și dezvoltatori

• Proiectele sunt construite în jurul unor persoane motivate, în care ar trebuie să fie de încredere

• Conversația față în față este cea mai bună formă de comunicare (co-locație)

• Software-ul produs este principala măsură a progresului

• Dezvoltare durabilă, capabilă să mențină un ritm constant

• O atenție continuă pentru excelența tehnică și un design bun

• Simplitatea - arta de a maximiza cantitatea de muncă neefectuată - este esențială

• Cele mai bune arhitecturi , cerințe și designuri apar din echipele auto-organizate

• În mod regulat, echipa reflectă modul de a deveni mai eficient și se ajustează în consecință

Prezentare generală

Iterativ, incremental și evolutiv

Majoritatea metodelor de dezvoltare Agile împart munca de dezvoltare a produsului în mici creșteri care reduc la minimum cantitatea de planificare și proiectare inițială. Ierațiile sau sprinturile sunt intervale scurte de timp (cutii de timp) care durează de obicei de la una la patru săptămâni. Fiecare iterație implică o echipă multifuncțională care lucrează în toate funcțiile: planificare , analiză , proiectare , codificare , testare unitară și testare de acceptare . La sfârșitul iterației, un produs funcțional este livrat părților interesate. Acest lucru minimizează riscul general și permite produsului să se adapteze rapid la schimbări.

Este posibil ca o iterație să nu adauge suficientă funcționalitate pentru a garanta o lansare pe piață, dar scopul este să existe o versiune disponibilă (cu bug-uri minime) la sfârșitul fiecărei iterații. Prin intermediul dezvoltării incrementale, produsele au posibilitatea să „eșueze des și devreme” de-a lungul fiecărei faze iterative, în loc de a ajunge drastic la dată de lansare finală. Este posibil să fie necesare mai multe iterații pentru lansarea unui produs sau a unor caracteristici noi. Oricum software-ul obtinut este principala măsură a progresului.

Un avantaj cheie al metodologiei Agile este viteza de intrare pe piață și reducerea riscurilor. Incrementari mai mici sunt de obicei lansate pe piață, reducând riscurile de timp și costul ingineresc al unui produs care nu îndeplinește cerințele utilizatorului.

Comunicare eficientă și față în față

Principiul co-locației se refera la colegii din aceeași echipă care ar trebui să fie locați împreună pentru a stabili mai bine identitatea ca echipă și pentru a îmbunătăți comunicarea. Aceasta permite interacțiunea față în față , în mod ideal în fața unei tabele albe, care reduce timpul total fata de ciclul pentru care întrebările și răspunsurile sunt mediate prin telefon, chat persistent, wiki sau e-mail.

Indiferent de metoda de dezvoltare urmată, fiecare echipă ar trebui să includă un reprezentant al clientului („Proprietar de produs” în Scrum). Această persoană este agreată de părțile interesate să acționeze în numele lor și își asume angajamentul personal de a fi disponibil dezvoltatorilor pentru a răspunde la întrebări pe tot parcursul iterației. La sfârșitul fiecărei iterații, părțile interesate și reprezentantul clienților analizează progresul și reevaluează prioritățile în vederea optimizării rentabilității investiției (ROI) și asigurării alinierii la nevoile clienților și la obiectivele companiei. Importanța satisfacției părților interesate, detaliată de interacțiuni și revizuiri frecvente la sfârșitul fiecărei faze, este motivul pentru care metodologia este adesea denumită „Metodologie centrată pe client”.

În dezvoltarea Agile a software-ului , un radiator de informații este un display fizic (în mod normal mare) situat vizibil lângă echipa de dezvoltare, unde trecătorii îl pot vedea. Prezintă un rezumat actualizat al stării de dezvoltare a produsului. Un indicator luminos al producției poate fi, de asemenea, utilizat pentru a informa o echipă despre starea actuală a dezvoltării produsului lor.

Buclă de feedback foarte scurtă și ciclu de adaptare

O caracteristică comună în dezvoltarea software-ului Agile este stand-up-ul zilnic (un scrum zilnic în cadrul Scrum). Într-o scurtă sesiune, membrii echipei își raportează reciproc ceea ce au făcut în ziua precedentă cu privire la obiectivul de iterație al echipei lor, ce intenționează să facă astăzi spre obiectiv și orice obstacole sau obstacole pe care le pot vedea în atingerea obiectivului. Sprinturile sunt de obicei menținute la o agendă minimă, deoarece în mod normal este prezentă echipa completă. Acest lucru înseamnă că elementele importante pot fi ridicate și rezolvate în timp ce discuțiile de nivel scăzut sunt preluate în afara stand-up-ului.

Focus de calitate

Instrumente și tehnici specifice, cum ar fi integrarea continuă , automate de unit testing , programare in tandem, dezvoltare condusă de teste , modele de design , dezvoltare condusă de comportament , de design-driven domeniu , cod refactoring si alte tehnici sunt adesea folosite pentru a îmbunătăți calitatea și de a spori dezvoltarea produsului agilitate. Acest lucru se bazează pe proiectarea și ridicarea calității încă de la început și posibilitatea de a livra software pentru clienți în orice moment sau cel puțin la sfârșitul fiecărei iterații.

Filosofia

Comparativ cu ingineria software tradițională, dezvoltarea software-ului Agile vizează în principal sistemele complexe și dezvoltarea produselor cu caracteristici dinamice, nedeterministe și neliniare. Estimările exacte, planurile stabile și predicțiile sunt adesea greu de obținut în stadii incipiente, iar încrederea în ele este probabil să fie scăzută. Practicienii Agile vor încerca să reducă saltul de credință care este necesar înainte ca orice dovadă de valoare să poată fi obținută.

Cerințele și proiectarea sunt considerate a fi emergente. Specificațiile mari ar provoca probabil multe deșeuri în astfel de cazuri, adică nu sunt sănătoase din punct de vedere economic. Aceste argumente de bază și experiențele anterioare din industrie, învățate din ani de succese și eșecuri, au contribuit la modelarea favorizării dezvoltării Agile a dezvoltării adaptive, iterative și evolutive.

Adaptiv vs. predictiv

Metodele de dezvoltare există pe un continuum de la adaptiv la predictiv . Metodele de dezvoltare software Agile se află pe latura adaptativă a acestui continuum. O cheie a metodelor de dezvoltare adaptivă este o abordare ca un val de rulare al planificării programului, care identifică repere, dar lasă flexibilitate în calea de a le atinge și permite, de asemenea, modificarea etapelor în sine.

Metodele adaptive se concentrează pe adaptarea rapidă la realitățile în schimbare. Când se schimbă nevoile unui proiect, se schimbă și o echipă adaptivă. O echipă adaptivă are dificultăți în a descrie exact ce se va întâmpla în viitor. Cu cât o dată este mai îndepărtată, cu atât este mai vagă o metodă adaptativă despre ceea ce se va întâmpla la acea dată. O echipă adaptivă nu poate raporta exact ce sarcini vor face săptămâna viitoare, ci doar ce caracteristici intenționează pentru luna viitoare.

În schimb, metodele predictive se concentrează pe analiza și planificarea detaliată a viitorului și răspund riscurilor cunoscute. În extrem, o echipă predictivă poate raporta exact ce caracteristici și sarcini sunt planificate pe întreaga durată a procesului de dezvoltare. Metodele predictive se bazează pe o analiză eficientă a fazelor timpurii și, dacă acest lucru merge foarte greșit, proiectul poate avea dificultăți în schimbarea direcției. Echipele predictive instituie adesea un panou de control al schimbărilor pentru a se asigura că iau în considerare doar cele mai importante schimbări.

Agile vs. Waterfall

Una dintre diferențele dintre metodele de dezvoltare software Agile și Waterfall este abordarea calității și testarea. În modelul cascadă , lucrările se desfășoară prin faze ale Software Development Lifecycle (SDLC) - cu o fază finalizată înainte ca alta să poată începe - deci faza de testare este separată și urmează o fază de construire. Cu toate acestea, în dezvoltarea software-ului Agile , testarea este finalizată în aceeași iterație ca programarea. Un alt mod de a-l privi este „Agile: inventează-l pe măsură ce mergi. Waterfall: inventează-o înainte de a începe, continua cu consecințele ”.

Deoarece testarea se face în fiecare iterație - care dezvoltă o mică parte din software - utilizatorii pot folosi frecvent acele noi piese de software și pot valida. După ce utilizatorii cunosc valoarea reală a software-ului actualizat, pot lua decizii mai bune cu privire la viitorul software-ului. Având o retrospectivă a valorii și o sesiune de re-planificare a software-ului în fiecare iterație - Scrum are de obicei iterații de doar două săptămâni - ajută echipa să își adapteze continuu planurile pentru a maximiza valoarea pe care o oferă. Acesta urmează un model similar ciclului Plan-Do-Check-Act (PDCA), deoarece lucrarea este planificată , realizată , verificată (în revizuire și retrospectivă), iar orice schimbări convenite sunt acționate peste.

Această abordare iterativă susține un produs mai degrabă decât o mentalitate de proiect . Aceasta oferă o mai mare flexibilitate pe tot parcursul procesului de dezvoltare; întrucât pentru proiecte cerințele sunt definite și blocate de la bun început, ceea ce face dificilă modificarea acestora ulterior. Dezvoltarea iterativă a produsului permite software-ului să evolueze ca răspuns la schimbările din mediul de afaceri sau cerințele pieței.

Metode de dezvoltare software Agile

Metodele de dezvoltare software Agile acceptă o gamă largă a Software Development Life Cycle . Unele metode se concentrează pe practici (de exemplu, XP, programare pragmatică, modelare Agile), în timp ce unele se concentrează pe gestionarea fluxului de muncă (de exemplu, Scrum, Kanban). Unele activități de

sprijin pentru specificarea și dezvoltarea cerințelor (de exemplu, FDD), în timp ce altele caută să acopere întregul ciclu de viață al dezvoltării (de exemplu, DSDM, RUP).

Practici de dezvoltare software Agile

Dezvoltarea software-ului Agile este susținută de o serie de practici concrete, acoperind domenii precum cerințe, proiectare, modelare, codificare, testare, planificare, gestionarea riscurilor, proces, calitate etc.

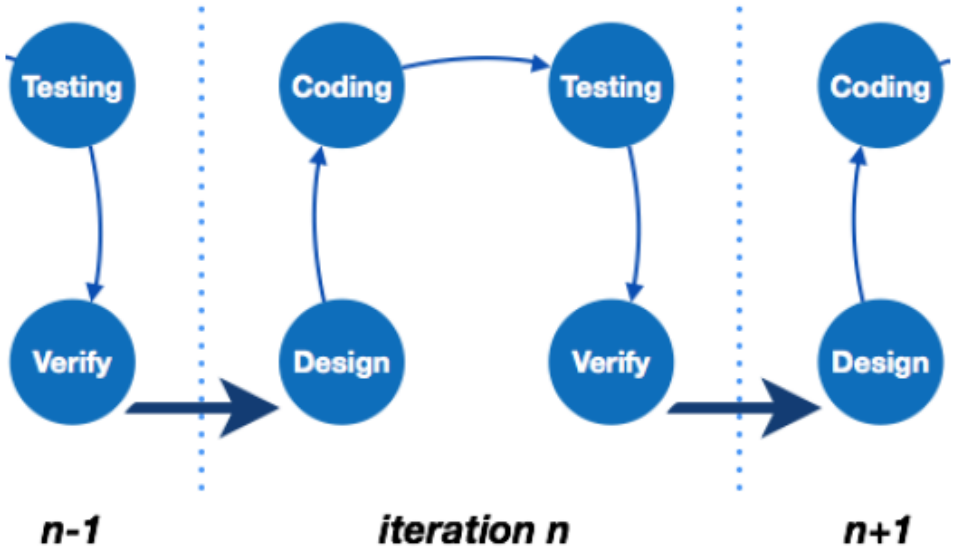
Sondaje publice

Unul dintre studiile timpurii care a raportat câștiguri în calitate, productivitate și crsterea satisfacției afacerii prin utilizarea metodelor de dezvoltare software Agile a fost un sondaj realizat de Shine Technologies din noiembrie 2002 până în ianuarie 2003.

Un sondaj similar, State of Agile , se desfășoară în fiecare an începând din 2006 cu mii de participanți din comunitatea de dezvoltare software. Aceasta urmărește tendințele privind beneficiile percepute ale agilității, lecțiile învățate și bunele practici. Fiecare sondaj a raportat un număr din ce în ce mai mare care spun că dezvoltarea software-ului Agile îi ajută să livreze software mai rapid, îmbunătățește capacitatea lor de a gestiona schimbarea priorităților clienților, și le crește productivitatea. Sondajele au arătat în mod constant rezultate mai bune cu metode Agile de dezvoltare a produselor, comparativ cu managementul clasic al proiectelor. În schimb, există rapoarte conform cărora unii consideră că metodele de dezvoltare Agile sunt încă prea tinere pentru a permite o cercetare academică extinsă a succesului lor.

4.3 Model Iterativ

Acest model conduce procesul de dezvoltare software în iterații. Proiectează procesul de dezvoltare în mod ciclic, repetând fiecare pas după fiecare ciclu al procesului CVDS.

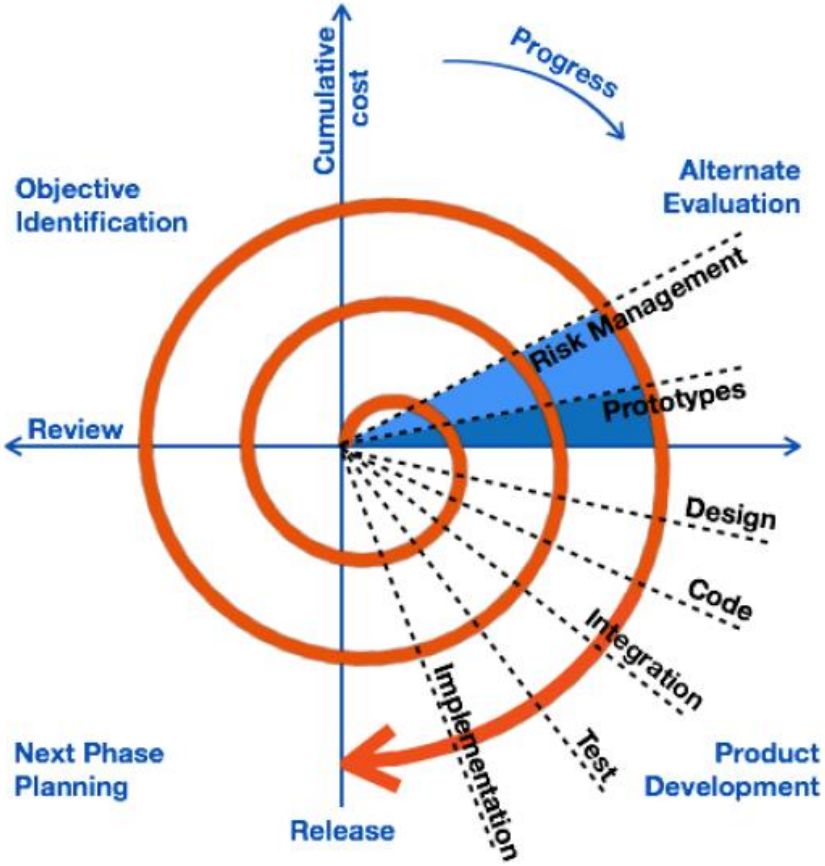


Software-ul este dezvoltat mai întâi la scară foarte mică și sunt urmați toți pașii care sunt luați în considerare. Apoi, la fiecare următoare iterație, mai multe caracteristici și module sunt proiectate, codificate, testate și adăugate la software. Fiecare ciclu produce un software, care este complet în sine și are mai multe caracteristici și capabilități decât cel al precedentului.

După fiecare iterație, echipa de management poate lucra la gestionarea riscurilor și se poate pregăti pentru următoarea iterație. Deoarece un ciclu include o mică parte din întregul proces software, este mai ușor să gestionezi procesul de dezvoltare, dar consumă mai multe resurse.

4.4 Model în Spirală

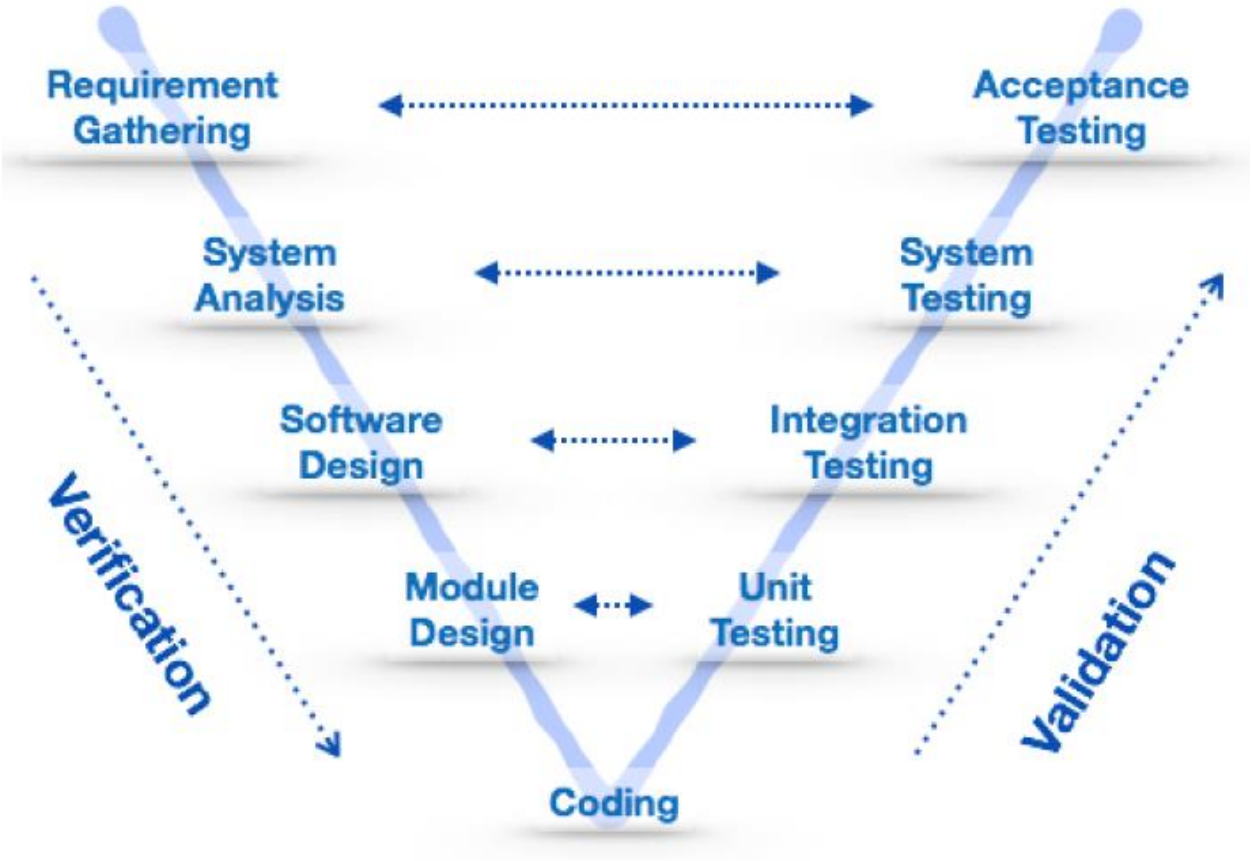
Modelul în spirală este o combinație a modelului iterativ și a modelului CVDS. Poate fi văzut ca și cum ai alege un model CVDS și l-ai combina cu proces ciclic (model iterativ).



Acest model consideră riscul, care deseori nu este observat de majoritatea celorlalte modele. Modelul începe cu determinarea obiectivelor și constrângerilor software-ului la începutul unei iterații. Următoarea fază este de a prototipa software-ul. Aceasta include analiza riscurilor. Apoi, un model CVDS standard este utilizat pentru a construi software-ul. În faza a patra a planului următoarei iterații este pregătit.

4.5 V - model

Dezavantajul major al modelului de cascadă este că trecem la etapa următoare numai când cea anterioară este terminată și nu a existat nicio șansă de a ne întoarce dacă ceva se găsește în neregulă în etapele ulterioare. *V-Model* oferă mijloace de testare a software-ului în fiecare etapă în sens invers.

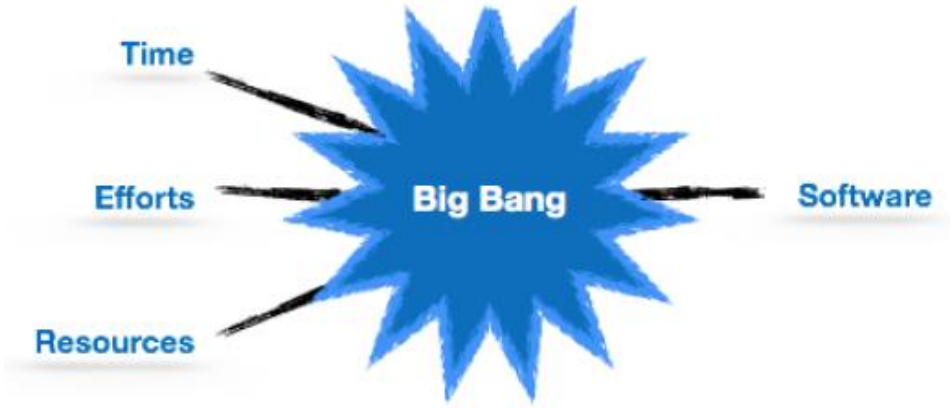


În fiecare etapă, sunt create planuri de testare și cazuri de testare pentru a verifica și valida produsul în conformitate cu cerința etapei respective. De exemplu, în etapa de colectare a cerințelor, echipa de testare pregătește toate cazurile de testare în conformitate cu cerințele. Mai târziu, când produsul este dezvoltat și este gata pentru testare, cazurile de testare din această etapă verifică software-ul în funcție de valabilitatea sa față de cerințele din această etapă.

Acest lucru face ca verificarea și validarea să meargă în paralel. Acest model este, de asemenea, cunoscut sub numele de *model de verificare și validare*.

4.6 Modelul Big Bang

Acest model este cel mai simplu model prin forma sa. Este nevoie de puțină planificare, multă programare și multe fonduri. Acest model este conceptualizat în jurul big bang-ului universului. După cum spun oamenii de știință, după Big Bang, multe galaxii, planete și stele au evoluat ca un eveniment. La fel, dacă adunăm o mulțime de programe și fonduri, este posibil să obțineți cel mai bun produs software.



Pentru acest model este necesară o planificare foarte mică. Nu urmărește niciun proces sau, uneori, clientul nu este sigur de cerințele și nevoile viitoare. Deci, cerințele de intrare sunt arbitrare.

Acest model nu este potrivit pentru proiecte software mari, dar este bun pentru învățare și experimentare.

5. Managementul Proiectelor Software

Modelul de muncă al unei companii IT implicate în dezvoltarea de software poate fi împărțit în două părți:

• Crearea de software

• Management de proiect software

Un proiect este o sarcină bine definită, care este o colecție de mai multe operațiuni efectuate pentru a atinge un obiectiv (de exemplu, dezvoltarea și livrarea software-ului). Un proiect poate fi caracterizat ca:

• Fiecare proiect poate avea un scop unic și distinct.

• Proiectul nu este o activitate de rutină sau o operațiune de zi cu zi.

• Proiectul vine cu ora de început și de sfârșit.

• Proiectul se încheie atunci când obiectivul său este atins. Prin urmare, este o fază temporară în viața unei organizații.

• Proiectul are nevoie de resurse adecvate în termeni de timp, forță de muncă, finanțe, materiale și bancă de cunoștințe.

5.1 Proiect Software

Un proiect software este procedura completă de dezvoltare software de la colectarea cerințelor până la testare și întreținere, efectuată în conformitate cu metodologiile de execuție, într-o perioadă de timp specificată pentru a realiza produsul software dorit.

5.2 Nevoia de gestionare a proiectelor software

Se spune că software-ul este un produs intangibil. Dezvoltarea de software este un fel de flux nou în afacerile mondiale și există foarte puțină experiență în construirea de produse software. Majoritatea produselor software sunt adaptate la cerințele clientului. Cel mai important este faptul că tehnologia de bază se schimbă și avansează atât de frecvent și de rapid încât experiența unui produs poate să nu fie aplicată celuilalt. Toate aceste constrângeri de afaceri și de mediu aduc riscuri în dezvoltarea software-ului, prin urmare este esențial să gestionați eficient proiectele software.

Imaginea de mai sus prezintă constrângeri triple pentru proiectele software. Este o parte esențială a organizării software-ului de a livra produse de calitate, menținând costurile în limitele bugetului clientului și livrând proiectul conform programării. Există mai mulți factori, atât interni, cât și externi, care pot afecta acest triunghi de constrângere triplă. Oricare dintre cei trei factori îi poate afecta grav pe ceilalți doi.

Prin urmare, managementul proiectelor software este esențial pentru a încorpora cerințele utilizatorilor, împreună cu constrângeri de buget și timp.

5.3 Manager de proiect software

Un manager de proiect software este o persoană care își asumă responsabilitatea executării proiectului software. Managerul de proiect software este complet conștient de toate fazele CVDS prin care ar trece software-ul. Managerul de proiect nu se poate implica niciodată direct în producerea produsului final, dar controlează și gestionează activitățile implicate în producție.

Un manager de proiect monitorizează îndeaproape procesul de dezvoltare, pregătește și execută diverse planuri, aranjează resursele necesare și adecvate, menține comunicarea între toți membrii echipei pentru a aborda probleme de cost, buget, resurse, timp, calitate și satisfacția clienților.

Să vedem câteva responsabilități pe care un manager de proiect le asumă:

Gestionarea oamenilor

• Acționați ca lider de proiect

• Legătură cu părțile interesate

• Gestionarea resurselor umane

• Configurarea ierarhiei de raportare etc.

Gestionarea proiectului

• Definirea și configurarea scopului proiectului

• Gestionarea activităților de gestionare a proiectelor

• Monitorizarea progresului și performanței

• Analiza riscurilor în fiecare fază

• Faceți pașii necesari pentru a evita sau ieși din probleme

• Acționați ca purtător de cuvânt al proiectului

5.4 Activități de gestionare a software-ului

Managementul proiectelor software cuprinde o serie de activități, care conțin planificarea proiectului, stabilirea domeniului de aplicare al produsului software, estimarea costurilor în termeni diferiți, programarea sarcinilor și evenimentelor și gestionarea resurselor. Activitățile de gestionare a proiectelor pot include:

• Planificarea proiectului

• Managementul domeniului

• Estimarea proiectului

5.5 Planificarea proiectului

Planificarea proiectelor software este o sarcină, care se realizează înainte de începerea efectivă a producției de software. Este acolo pentru producția de software, dar nu implică nicio activitate concretă care să aibă vreo legătură directă cu producția de software; mai degrabă este un set de procese multiple, care facilitează producția de software.

Planificarea proiectului poate include următoarele:

Managementul domeniului

Acesta definește scopul proiectului; aceasta include toate activitățile, procesul trebuie făcut pentru a crea un produs software livrabil. Managementul domeniului este esențial deoarece creează limite ale proiectului prin definirea clară a ceea ce s-ar face în proiect și a ceea ce nu s-ar face. Acest lucru face ca proiectul să conțină sarcini limitate și cuantificabile, care pot fi ușor documentate și, la rândul lor, evită depășirea costurilor și a timpului.

În timpul managementului domeniului proiectului, este necesar să -

• Definiți domeniul de aplicare

• Decideți verificarea și controlul acestuia

• Împărțiți proiectul în diferite părți mai mici pentru o gestionare ușoară.

• Verificați domeniul de aplicare

• Controlați domeniul de aplicare prin încorporarea modificărilor domeniului de aplicare

Estimarea proiectului

Pentru un management eficient, este necesară o estimare exactă a diferitelor măsuri. Cu o estimare corectă, managerii pot gestiona și controla proiectul mai eficient și mai eficient.

Estimarea proiectului poate implica următoarele:

• **Estimarea dimensiunii software-ului**

Dimensiunea software-ului poate fi estimată fie în termeni de KLOC (Kilo Line of Code), fie calculând numărul de puncte funcționale din software. Liniile de cod depind de practicile de codificare. Punctele funcționale variază în funcție de cerința utilizatorului sau a software-ului.

• **Estimarea efortului**

Managerul estimează eforturile în ceea ce privește cerința de personal și ora de muncă necesară pentru a produce software-ul. Pentru estimarea efortului, trebuie cunoscută dimensiunea software-ului. Acest lucru poate fi obținut fie prin experiența managerului, datele istorice ale organizației, fie dimensiunea software-ului poate fi convertită în eforturi prin utilizarea unor formule standard.

• **Estimarea timpului**

Odată ce dimensiunea și eforturile sunt estimate, timpul necesar pentru a produce software-ul poate fi estimat. Eforturile necesare sunt separate în subcategorii, conform specificațiilor cerințelor și interdependenței diferitelor componente ale software-ului. Sarcinile software sunt împărțite în sarcini, activități sau evenimente mai mici, după structura Work Breakthrough Structure (WBS). Sarcinile sunt programate zilnic sau în luni calendaristice.

Suma de timp necesară pentru a finaliza toate sarcinile în ore sau zile este timpul total investit pentru finalizarea proiectului.

• **Estimarea costurilor**

Acest lucru ar putea fi considerat cel mai dificil dintre toate, deoarece depinde de mai multe elemente decât oricare dintre cele anterioare. Pentru estimarea costului proiectului, este necesar să se ia în considerare:

• Dimensiunea software-ului

• Calitatea software-ului

• Hardware

• Software sau instrumente suplimentare, licențe etc.

• Personal calificat cu abilități specifice sarcinii

• Călătorii implicate

• Comunicare

• Instruire și sprijin

5.6 Tehnici de estimare a proiectului

Am discutat diverși parametri care implică estimarea proiectului, cum ar fi dimensiunea, efortul, timpul și costul.

Managerul de proiect poate estima factorii enumerați utilizând două tehnici larg recunoscute -

5.7 Tehnica descompunerii

Această tehnică presupune software-ul ca produs al diferitelor compoziții.

Există două modele principale -

• **Linie de cod**: aici estimarea se face în numele numărului de linii de coduri din produsul software.

• **Puncte funcționale**: aici estimarea se face în numele numărului de puncte funcționale din produsul software.

Tehnica de estimare empirică

Această tehnică folosește formule derivate empiric pentru a face estimări. Aceste formule se bazează pe LOC sau FP.

• Modelul Putnam

Acest model este realizat de Lawrence H. Putnam, care se bazează pe distribuția frecvenței Norden (curba Rayleigh). Modelul Putnam mapează timpul și eforturile necesare cu dimensiunea software-ului.

• COCOMO

COCOMO înseamnă Constructive Cost Model, dezvoltat de Barry W. Boehm. Împarte produsul software în trei categorii de software: organic, semidecomandat și încorporat.

5.8 Planificarea proiectului

Planificarea într-un proiect se referă la foaia de parcurs a tuturor activităților care trebuie realizate cu ordinea specificată și în intervalul de timp alocat fiecărei activități. Managerii de proiect au tendința de a defini diverse sarcini și de a reține proiectele, apoi le aranjează ținând cont de diferiți factori. Ei caută sarcini cum ar fi în calea critică în program, care sunt necesare pentru a fi finalizate în mod specific (din cauza interdependenței sarcinilor) și strict în timpul alocat. Aranjarea sarcinilor care iese din calea critică are mai puține șanse să aibă impact asupra tuturor programelor proiectului.

Pentru programarea unui proiect, este necesar să:

• Descompuneți sarcinile proiectului într-o formă mai mică și mai ușor de gestionat

• Aflați diverse sarcini și corelați-le

• Estimarea intervalului de timp necesar pentru fiecare sarcină

• Împărțiți timpul în unități de lucru

• Alocați un număr adecvat de unități de lucru pentru fiecare sarcină

• Calculați timpul total necesar proiectului de la început până la sfârșit

5.9 Managementul resurselor

Toate elementele utilizate pentru dezvoltarea unui produs software pot fi asumate ca resurse pentru acel proiect. Aceasta poate include resurse umane, instrumente productive și biblioteci de software.

Resursele sunt disponibile în cantitate limitată și rămân în organizație ca un grup de active. Lipsa resurselor împiedică dezvoltarea proiectului și poate rămâne în urmă cu programul. Alocarea resurselor suplimentare crește costul de dezvoltare în cele din urmă. Prin urmare, este necesar să se estimeze și să se aloce resurse adecvate pentru proiect.

Managementul resurselor include:

• Definirea unui proiect de organizare adecvat prin crearea unei echipe de proiect și alocarea responsabilităților fiecărui membru al echipei

• Determinarea resurselor necesare într-o anumită etapă și disponibilitatea acestora

• Gestionați resursele generând cereri de resurse atunci când sunt necesare și dezalocându-le atunci când nu mai sunt necesare.

5.10 Managementul riscului proiectului

Managementul riscurilor implică toate activitățile legate de identificare, analiză și asigurarea riscurilor previzibile și neprevăzute în cadrul proiectului. Riscul poate include următoarele:

• Personal cu experiență care părăsește proiectul și vine personal fara experienta.

• Schimbarea managementului organizațional.

• Modificarea cerinței sau cerința de interpretare greșită.

• Subestimarea timpului și resurselor necesare.

• Schimbări tehnologice, schimbări de mediu, concurență în afaceri.

Procesul de gestionare a riscurilor

Există următoarele activități implicate în procesul de gestionare a riscurilor:

• Identificare - Notați toate riscurile posibile, care pot apărea în cadrul proiectului.

• Categorizare - Clasificați riscurile cunoscute în intensitate de risc ridicată, medie și scăzută, în funcție de impactul lor posibil asupra proiectului.

• Gestionați - Analizați probabilitatea apariției riscurilor în diferite faze.

Planificați-vă pentru a evita sau a face față riscurilor. Încercați să minimizați efectele lor secundare.

• Monitor - Monitorizați îndeaproape riscurile potențiale și simptomele lor timpurii.

De asemenea, monitorizați măsurile eficiente luate pentru a le atenua sau a le evita.

5.11 Executarea și monitorizarea proiectului

În această fază, sarcinile descrise în planurile de proiect sunt executate în conformitate cu programele lor.

Execuția are nevoie de monitorizare pentru a verifica dacă totul merge conform planului. Monitorizarea este observarea pentru a verifica probabilitatea riscului și luarea de măsuri pentru abordarea riscului sau raportarea stării diferitelor sarcini.

Aceste măsuri includ:

• Monitorizarea activității - Toate activitățile programate în cadrul anumitor sarcini pot fi monitorizate de la o zi la alta. Când toate activitățile dintr-o sarcină sunt finalizate, aceasta este considerată completă.

• Rapoarte de stare - Rapoartele conțin starea activităților și sarcinilor finalizate într-un anumit interval de timp, în general o săptămână. Starea poate fi marcată ca terminată, în așteptare sau în curs de desfășurare etc.

• Lista de verificare a etapelor - Fiecare proiect este împărțit în mai multe faze în care sunt îndeplinite sarcini majore (repere) pe baza fazelor CVDS. Această listă de verificare a reperelor este pregătită o dată la câteva săptămâni și raportează starea reperelor.

5.12 Managementul comunicării de proiect

Comunicarea eficientă joacă un rol vital în succesul unui proiect. Acoperă decalajele dintre client și organizație, între membrii echipei, precum și alți stakeholderi din proiect, cum ar fi furnizorii de hardware.

Comunicarea poate fi orală sau scrisă. Procesul de gestionare a comunicării poate avea următorii pași:

• **Planificare** - Acest pas include identificarea tuturor părților interesate din proiect și modul de comunicare între aceștia. De asemenea, ia în considerare dacă sunt necesare facilități de comunicare suplimentare.

• **Partajare** - După determinarea diferitelor aspecte ale planificării, managerul se concentrează pe schimbul de informații corecte cu persoana corectă la momentul corect. Acest lucru îi menține pe toți cei implicați în proiect la curent cu progresul proiectului și starea acestuia.

• **Feedback** - Managerii de proiect utilizează diferite măsuri și mecanisme de feedback și creează rapoarte de stare și performanță. Acest mecanism asigură faptul că contribuția diferitelor părți interesate primește managerului de proiect ca feedback al acestora.

• **Închidere** - La sfârșitul fiecărui eveniment major, la sfârșitul unei faze a CVDS sau la sfârșitul proiectului în sine, se anunță formal închiderea administrativă pentru a actualiza fiecare părți interesate prin trimiterea de e-mailuri, prin distribuirea unei hârtie a documentului sau prin alte mijloace de comunicare eficientă .

După închidere, echipa trece la următoarea fază sau proiect.

5.13 Managementul configurației

Managementul configurației este un proces de urmărire și control al modificărilor software-ului în ceea ce privește cerințele, proiectarea, funcțiile și dezvoltarea produsului.

IEEE îl definește ca „procesul de identificare și definire a articolelor din sistem, controlul schimbării acestor articole de-a lungul ciclului lor de viață, înregistrarea și raportarea stării articolelor și a cererilor de modificare și verificarea completitudinii și corectitudinii articolelor”.

În general, odată ce SRS este finalizat, există mai puține șanse de a solicita modificări de la utilizator. Dacă apar, modificările sunt abordate numai cu aprobarea prealabilă a conducerii superioare, deoarece există posibilitatea depășirii costurilor și a timpului.

De bază

Se presupune o fază a CVDS dacă este bazată pe linie, adică linia de bază este o măsurătoare care definește completitudinea unei faze. O fază este inițiată atunci când toate activitățile aferente acesteia sunt finalizate și bine documentate. Dacă nu a fost faza finală, rezultatul său va fi utilizat în următoarea fază imediată.

Managementul configurației este o disciplină a administrării organizației, care are grijă de apariția oricăror modificări (proces, cerință, tehnologice, strategice etc.) după ce o fază este bazată pe bază. CM continuă să verifice orice schimbări efectuate în software.

Controlul modificărilor

Controlul modificărilor este funcția de gestionare a configurației, care asigură faptul că toate modificările aduse sistemului software sunt coerente și efectuate conform regulilor și reglementărilor organizaționale.

O modificare a configurației produsului parcurge următorii pași -

• **Identificare** - Sosește o cerere de modificare fie din sursa internă, fie din cea externă. Când cererea de modificare este identificată formal, aceasta este documentată corespunzător.

• **Validare** - Validitatea cererii de modificare este verificată și procedura de gestionare a acesteia este confirmată.

• **Analiză** - Impactul cererii de modificare este analizat în termeni de program, cost și eforturi necesare. Este analizat impactul global al schimbării prospective asupra sistemului.

• **Control** - Dacă schimbarea potențială are un impact asupra prea multor entități din sistem sau este inevitabilă, este obligatoriu să primiți aprobarea autorităților înalte înainte ca modificarea să fie încorporată în sistem. Se decide dacă schimbarea merită sau nu încorporată. În caz contrar, cererea de modificare este respinsă în mod formal.

• **Executare** - Dacă faza anterioară determină executarea cererii de modificare, această fază ia măsurile adecvate pentru a executa modificarea, printr-o revizuire amănunțită, dacă este necesar.

• **Cerere** i**nchidere** - Modificarea este verificată pentru implementarea corectă și fuzionarea cu restul sistemului. Această modificare recent încorporată în software este documentată corect și solicitarea este închisă în mod formal.

5.14 Instrumente de gestionare a proiectelor

Riscul și incertitudinea cresc mult în raport cu dimensiunea proiectului, chiar și atunci când proiectul este dezvoltat conform metodologiilor stabilite.

Există instrumente disponibile, care ajută la gestionarea eficientă a proiectelor. Câteva descrise sunt: -

Diagrama Gantt

Diagrama Gantt a fost concepută de Henry Gantt (1917). Reprezintă programul proiectului în ceea ce privește perioadele de timp. Este o diagramă orizontală cu bare care reprezintă activități și timpul programat pentru activitățile proiectului.

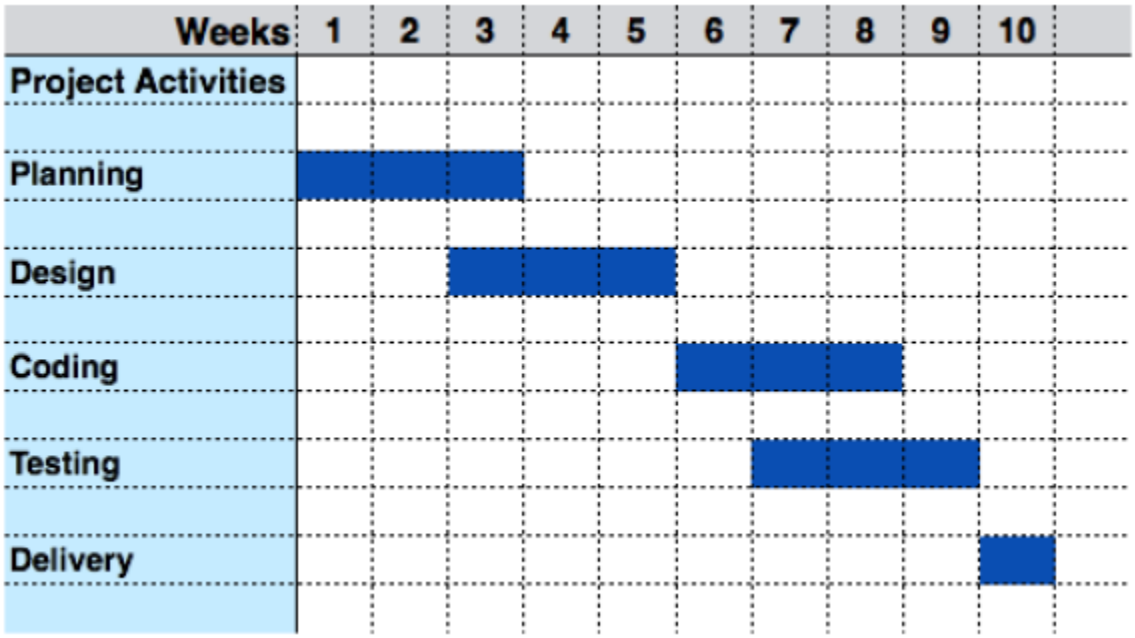
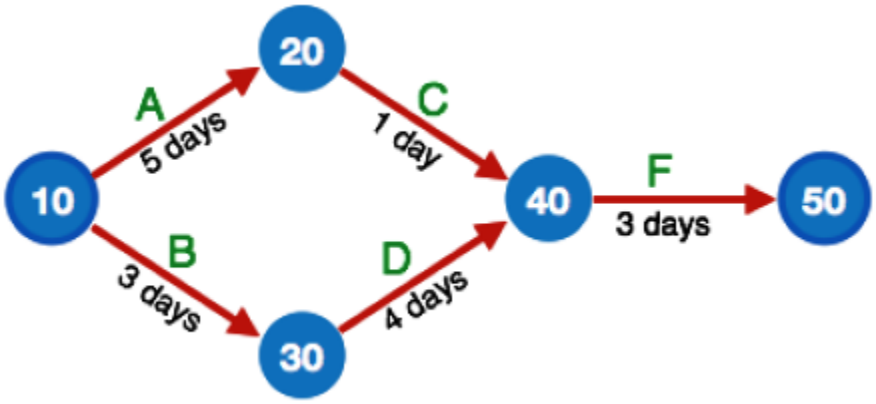


Diagrama PERT

Diagrama de evaluare și revizuire a programului (PERT) este un instrument care descrie proiectul ca diagramă de rețea. Este capabil să reprezinte grafic evenimentele principale ale proiectului atât în mod paralel, cât și în mod consecutiv. Evenimentele, care apar una după alta, arată dependența de evenimentul ulterior față de cel anterior.

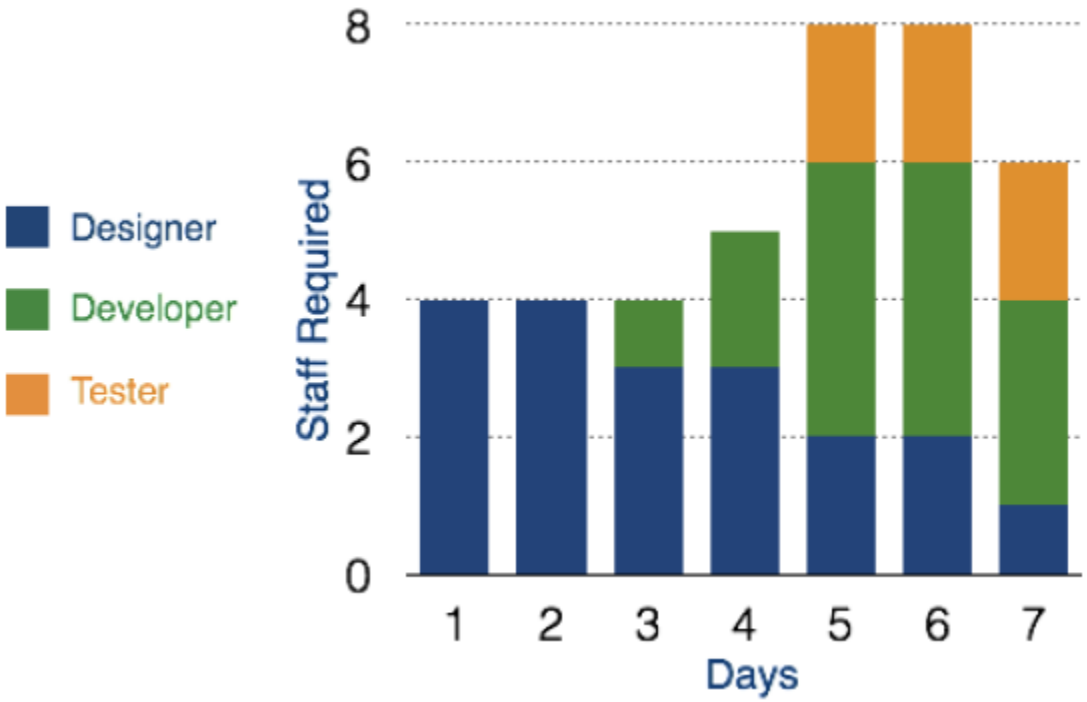


Evenimentele sunt afișate ca noduri numerotate. Acestea sunt conectate prin săgeți etichetate care descriu secvența sarcinilor din proiect.

Histograma resurselor

Acesta este un instrument grafic care conține bare sau diagrame care reprezintă numărul de resurse (de obicei personal calificat) necesare în timp pentru un eveniment (sau fază) de proiect. Histograma resurselor este un instrument eficient pentru planificarea și coordonarea personalului.

Staff	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
Designer	4	4	3	3	2	2	1
Developer	0	0	1	2	4	4	3
Tester	0	0	0	0	2	2	2
Total	4	4	4	5	8	8	6



Analiza căilor critice

Aceste instrumente sunt utile în recunoașterea sarcinilor interdependente din proiect. De asemenea, vă ajută să aflați calea cea mai scurtă sau calea critică pentru a finaliza cu succes proiectul. La fel ca diagrama PERT, fiecărui eveniment i se alocă un anumit interval de timp. Acest instrument arată dependența de eveniment, presupunând că un eveniment poate trece la următorul numai dacă cel anterior este finalizat.

Evenimentele sunt aranjate în funcție de cel mai devreme timp de început posibil. Calea dintre nodul de început și sfârșit este o cale critică, care nu poate fi redusă în continuare și toate evenimentele necesită a fi executate în aceeași ordine.

6. Cerințele Software

Cerințele software sunt descrierea caracteristicilor și funcționalităților sistemului țintă. Cerințele transmit așteptările utilizatorilor de la produsul software. Cerințele pot fi evidente sau ascunse, cunoscute sau necunoscute, așteptate sau neașteptate din punctul de vedere al clientului.

6.1 Ingineria cerințelor

Procesul de colectare a cerințelor software de la client, analizarea și documentarea acestora este cunoscut ca inginerie de cerințe.

Scopul ingineriei cerințelor este de a dezvolta și menține documentul „Specificații cerințe de sistem" sofisticat și descriptiv.

6.2 Procesul de inginerie a cerințelor

Este un proces în patru etape, care include:

• Studiu de fezabilitate

• Adunarea cerințelor

• Specificația cerințelor software

• Validarea cerințelor software

Să vedem procesul pe scurt:

Studiu de fezabilitate

Când clientul se apropie de organizație pentru a dezvolta produsul dorit, acesta are o idee aproximativă despre ce funcții trebuie să îndeplinească software-ul și care sunt toate caracteristicile așteptate de la software.

Referindu-se la aceste informații, analiștii fac un studiu detaliat cu privire la faptul dacă sistemul dorit și funcționalitatea acestuia sunt fezabile să se dezvolte.

Acest studiu de fezabilitate este axat pe obiectivul organizației. Acest studiu analizează dacă produsul software poate fi practic concretizat în termeni de implementare, contribuția proiectului la organizație, constrângeri de cost și conform valorilor și obiectivelor organizației. Acesta explorează aspectele tehnice ale proiectului și produsului, cum ar fi utilizabilitatea, mentenabilitatea, productivitatea și capacitatea de integrare.

Rezultatul acestei faze ar trebui să fie un raport de studiu de fezabilitate care ar trebui să conțină comentarii și recomandări adecvate pentru conducere cu privire la dacă proiectul ar trebui sau nu întreprins.

Adunarea cerințelor

Dacă raportul de fezabilitate este pozitiv pentru realizarea proiectului, faza următoare începe cu colectarea cerințelor de la utilizator. Analiștii și inginerii comunică cu clientul și utilizatorii finali pentru a-și cunoaște ideile cu privire la ceea ce ar trebui să furnizeze software-ul și ce caracteristici doresc să includă software-ul.

Specificația cerințelor software (SRS)

SRS este un document creat de analistul de sistem după ce cerințele sunt colectate de la diferiți actori.

SRS definește modul în care software-ul intenționat va interacționa cu hardware-ul, interfețele externe, viteza de funcționare, timpul de răspuns al sistemului, portabilitatea software-ului pe diferite platforme, mentenabilitatea, viteza de recuperare după blocare, Securitate, Calitate, Limitări etc.

Cerințele primite de la client sunt scrise în limbaj natural. Este responsabilitatea analistului de sistem să documenteze cerințele în limbaj tehnic, astfel încât acestea să poată fi înțelese și utilizate de către echipa de dezvoltare software.

SRS ar trebui să vină cu următoarele caracteristici:

• Cerințele utilizatorilor sunt exprimate în limbaj natural.

• Cerințele tehnice sunt exprimate într-un limbaj structurat, care este utilizat în cadrul organizației.

• Descrierea proiectului trebuie să fie scrisă în codul Pseudo.

• Format de formulare și serigrafii GUI.

• Notații condiționale și matematice pentru DFD etc.

Validarea cerințelor software

După elaborarea specificațiilor cerințelor, cerințele menționate în acest document sunt validate. Utilizatorul poate solicita o soluție ilegală, nepractică sau experții pot interpreta cerințele în mod incorect. Acest lucru duce la o creștere uriașă a costului dacă nu este prins în mugur. Cerințele pot fi verificate în funcție de următoarele condiții -

• Dacă pot fi implementate practic

• Dacă sunt valabile și conform funcționalității și domeniului software

• Dacă există ambiguități

• Dacă sunt complete

• Dacă pot fi demonstrate

6.3 Procesul de Solicitare a Cerințelor

Procesul de solicitare a cerințelor poate fi descris folosind diagrama următoare:

• **Colectarea cerințelor** - Dezvoltatorii discută cu clientul și utilizatorii finali și își cunosc așteptările de la software.

• **Cerințe de organizare** - Dezvoltatorii acordă prioritate și aranjează cerințele în ordinea importanței, urgenței și comodității.

• **Negociere & discutii** - Dacă cerințele sunt ambigue sau există anumite conflicte în cerințele diferitelor părți interesate, acestea sunt apoi negociate și discutate cu părțile interesate. Cerințele pot fi apoi prioritizate și compromise în mod rezonabil.

Cerințele provin de la diferiți actori. Pentru a elimina ambiguitatea și conflictele, acestea sunt discutate pentru claritate și corectitudine. Cerințele nerealiste sunt compromise în mod rezonabil.

• **Documentare** - Toate cerințele formale și informale, funcționale și nefuncționale sunt documentate și puse la dispoziție pentru procesarea următoarei etape.

6.4 Tehnici de solicitare a cerințelor

Eliberarea cerințelor este procesul pentru a afla cerințele pentru un sistem software intenționat prin comunicarea cu clientul, utilizatorii finali, utilizatorii de sistem și alții care au o miză în dezvoltarea sistemului de software.

Există diferite moduri de a descoperi cerințele. Unele dintre ele sunt explicate mai jos:

Interviuri

Interviurile sunt un mediu puternic pentru a colecta cerințe. Organizația poate efectua mai multe tipuri de interviuri, cum ar fi:

• Interviurile structurate (închise), în care fiecare informație care trebuie colectată este decisă în prealabil, urmează cu fermitate tiparul și chestiunea de discuție.

• Interviuri nestructurate (deschise), în care informațiile de colectat nu sunt luate în prealabil, mai flexibile și mai puțin părtinitoare.

• Interviuri orale

• Interviuri scrise

• Interviuri individuale care au loc între două persoane de-a lungul mesei.

• Interviuri de grup care au loc între grupuri de participanți. Acestea ajută la descoperirea oricărei cerințe care lipsesc, deoarece sunt implicate numeroase persoane.

Sondaje

Organizația poate efectua sondaje în rândul diferitelor părți interesate întrebând despre așteptările și cerințele acestora din sistemul viitor.

Chestionare

Un document cu un set predefinit de întrebări obiective și opțiuni respective este predat tuturor părților interesate pentru a răspunde, care sunt colectate și compilate.

O deficiență a acestei tehnici este, dacă o opțiune pentru o problemă nu este menționată în chestionar, problema ar putea fi lăsată nesupravegheată.

Analiza sarcinilor

Echipa de ingineri și dezvoltatori poate analiza operațiunea pentru care este necesar noul sistem. Dacă clientul are deja un software pentru a efectua anumite operațiuni, acesta este studiat și sunt colectate cerințele sistemului propus.

Analiza domeniului

Fiecare software se încadrează într-o anumită categorie de domeniu. Experții din domeniu pot fi de mare ajutor pentru analiza cerințelor generale și specifice.

Brainstorming

O dezbatere informală are loc între diferiți actori și toate contribuțiile lor sunt înregistrate pentru o analiză ulterioară a cerințelor.

Prototipare

Prototiparea construiește interfața utilizatorului fără a adăuga funcționalități detaliate pentru ca utilizatorul să interpreteze caracteristicile produsului software dorit. Ajută la o mai bună idee a cerințelor. Dacă nu există niciun software instalat la sfârșitul clientului pentru referința dezvoltatorului și clientul nu este conștient de propriile cerințe, dezvoltatorul creează un prototip pe baza cerințelor menționate inițial. Prototipul este prezentat clientului și feedback-ul este notat. Feedback-ul clientului servește drept element de intrare pentru colectarea cerințelor.

Observare

Echipa de experți vizitează organizația sau locul de muncă al clientului. Ei observă funcționarea efectivă a sistemelor instalate existente. Ei observă fluxul de lucru la sfârșitul clientului și modul în care sunt tratate problemele de execuție. Echipa în sine trage câteva concluzii care ajută la formarea cerințelor așteptate de la software.

6.5 Caracteristici ale cerințelor software

Adunarea cerințelor software este fundamentul întregului proiect de dezvoltare software. Prin urmare, acestea trebuie să fie clare, corecte și bine definite.

Specificațiile complete ale cerințelor software trebuie să fie:

- Clar
- Corect
- Consistent
- Coerent
- Înțeles

• Modificabil

• Verificabil

• Prioritizat

• Neambigu

• Urmărire

• Sursă credibilă

Cerințe software

Ar trebui să încercăm să înțelegem ce fel de cerințe pot apărea în faza de solicitare a cerințelor și ce tipuri de cerințe sunt așteptate de la sistemul software.

În general, cerințele software ar trebui clasificate în două categorii:

Cerințe funcționale

Cerințele legate de aspectul funcțional al software-ului se încadrează în această categorie.

Acestea definesc funcțiile și funcționalitatea din și din sistemul software.

EXEMPLE -

• Opțiunea de căutare dată utilizatorului pentru a căuta din diverse facturi.

• Utilizatorul ar trebui să poată trimite orice raport către conducere.

• Utilizatorii pot fi împărțiți în grupuri, iar grupurilor li se pot acorda drepturi separate.

• Ar trebui să respecte regulile de afaceri și funcțiile administrative.

• Software-ul este dezvoltat păstrând intactă compatibilitatea descendentă.

Cerințe nefuncționale

Cerințele, care nu sunt legate de aspectul funcțional al software-ului, intră în această categorie. Acestea sunt caracteristici implicite sau așteptate ale software-ului, pe care utilizatorii le presupun.

Cerințele nefuncționale includ -

• Securitate

• Înregistrare

• Depozitare

• Configurare

• Performanță

• Cost

• Interoperabilitate

• Flexibilitate

• Recuperare în caz de dezastru

• Accesibilitate

Cerințele sunt clasificate logic ca:

• **Trebuie să aibă**: Software-ul nu poate fi declarat funcțional fără ele.

• **Ar trebui să aibă**: îmbunătățirea funcționalității software-ului.

• **S-ar putea să aibă**: Software-ul poate funcționa în continuare corect cu aceste cerințe.

• **Lista de dorințe**: aceste cerințe nu corespund niciunui obiectiv al software-ului.

În timpul dezvoltării software-ului, „Must have" trebuie implementat, „Should have" este o chestiune de dezbatere cu părțile interesate și negare, în timp ce „Could have" și „Wish list" pot fi păstrate pentru actualizări de software.

6.6 Cerințe de interfață utilizator

Interfața cu utilizatorul (UI) este o parte importantă a oricărui software sau hardware sau sistem hibrid. Un software este acceptat pe scară largă dacă este -

• ușor de operat

• rapid ca răspuns

• gestionarea eficientă a erorilor operaționale

• furnizarea unei interfețe de utilizator simple, dar consistente

Acceptarea utilizatorului depinde în principal de modul în care utilizatorul poate utiliza software-ul. IU este singura modalitate prin care utilizatorii pot percepe sistemul. Un sistem software performant trebuie, de asemenea, să fie echipat cu o interfață de utilizator atractivă, clară, consecventă și receptivă. În caz contrar, funcționalitățile sistemului software nu pot fi utilizate în mod convenabil. Se spune că un sistem este bun dacă oferă mijloace de utilizare eficientă. Cerințele interfeței cu utilizatorul sunt menționate pe scurt mai jos -

• Prezentarea conținutului

• Navigare ușoară

• Interfață simplă

• receptiv

• Elemente UI consistente

• Mecanism de feedback

• Setări implicite

• Aspect intenționat

• Utilizarea strategică a culorii și a texturii.

• Furnizați informații de ajutor

• Abordarea centrată pe utilizator

• Setări de vizualizare bazate pe grup.

7. Analist de sistem software

Analistul de sistem dintr-o organizație IT este o persoană care analizează cerința sistemului propus și se asigură că cerințele sunt concepute și documentate în mod corespunzător și acuurat. Rolul unui analist începe în timpul fazei de analiză software a CVDS. Este responsabilitatea analistului să se asigure că software-ul dezvoltat îndeplinește cerințele clientului.

Analiștii de sistem au următoarele responsabilități:

• Analizarea și înțelegerea cerințelor software-ului intenționat

• Înțelegerea modului în care proiectul va contribui la obiectivele organizaționale

• Identificați sursele de cerință

• Validarea cerinței

• Elaborați și implementați un plan de gestionare a cerințelor

• Documentarea cerințelor de afaceri, tehnice, de proces și de produs

• Coordonarea cu clienții pentru prioritizarea cerințelor și eliminarea ambiguității

• Finalizarea criteriilor de acceptare cu clientul și alte părți interesate

8. Metrici și măsuratori software

Măsurile software pot fi înțelese ca un proces de cuantificare și simbolizare a diferitelor atribute și aspecte ale software-ului.

Software Metrics oferă măsuri pentru diferite aspecte ale procesului software și ale produsului software.

Măsurile software sunt cerințe fundamentale ale ingineriei software. Acestea ajută nu numai la controlul procesului de dezvoltare a software-ului, ci și la menținerea excelentă a calității produsului final.

Potrivit lui Tom DeMarco, un (inginer software), „Nu puteți controla ceea ce nu puteți măsura”. Prin spusele sale, este foarte clar cât de importante sunt măsurile software.

Să vedem câteva valori de software:

• **Măsuri de dimensiune** - Linii de cod (LOC) (), calculate în principal în mii de linii de cod sursă livrate, denumite KLOC.

• Numărul punctelor funcției este măsurarea funcționalității oferite de software. Numărul de puncte funcționale definește dimensiunea aspectului funcțional al software-ului.

• **Metricitatea complexității** - Complexitatea ciclomatică a lui McCabe cuantifică limita superioară a numărului de căi independente dintr-un program, care este percepută ca fiind complexitatea programului sau a modulelor sale. Este reprezentat în termeni de concepte ale teoriei graficelor utilizând graficul fluxului de control.

• **Măsuri de calitate** - Defectele, tipurile și cauzele acestora, consecința, intensitatea severității și implicațiile acestora definesc calitatea produsului.

• Numărul de defecte constatate în procesul de dezvoltare și numărul de defecte raportate de client după instalarea sau livrarea produsului la client, definesc calitatea produsului.

• **Metricele proceselor** - În diferite faze ale CVDS, metodele și instrumentele utilizate, standardele companiei și performanța dezvoltării sunt metricele proceselor software.

• **Metricele resurselor** - Efortul, timpul și diversele resurse utilizate reprezintă valori pentru măsurarea resurselor.

9. Bazele Proiectării Software-ului

Proiectarea software-ului este un proces de transformare a cerințelor utilizatorilor într-o formă adecvată, care ajută programatorul în codificarea și implementarea software-ului.

Pentru evaluarea cerințelor utilizatorilor, se creează un document SRS (Specificația cerințelor software), în timp ce pentru codificare și implementare, este nevoie de cerințe mai specifice și detaliate în termeni software. Rezultatul acestui proces poate fi utilizat direct în implementarea în limbaje de programare.

Proiectarea software-ului este primul pas în CVDS (Software Design Life Cycle), care mută concentrația din domeniul problemei în domeniul soluției. Încearcă să specifice cum să îndeplinească cerințele menționate în SRS.

10. Nivele de proiectare software

Proiectarea software oferă trei niveluri de rezultate:

• **Proiectare arhitecturală** - Proiectarea arhitecturală este cea mai înaltă versiune abstractă a sistemului. Identifică software-ul ca un sistem cu multe componente care interacționează între ele. La acest nivel, proiectanții au ideea domeniului soluției propuse.

• **Proiectare la nivel înalt** - Proiectarea la nivel înalt rupe conceptul de „componentă unică entitate multiplă” a designului arhitectural într-o vedere mai puțin abstractizată a subsistemelor și modulelor și descrie interacțiunea lor între ele. Proiectarea la nivel înalt se concentrează pe modul în care sistemul împreună cu toate componentele sale pot fi implementate în forme de module. Recunoaște structura modulară a fiecărui subsistem și relația și interacțiunea lor între ele.

• **Proiectare detaliată** - Proiectarea detaliată se ocupă de partea de implementare a ceea ce este văzut ca un sistem și subsistemele sale în cele două modele anterioare. Este mai detaliat în ceea ce privește modulele și implementările acestora. Acesta definește structura logică a fiecărui modul și interfețele acestora pentru a comunica cu alte module.

10.1 Modularizare

Modularizarea este o tehnică de împărțire a unui sistem software în mai multe module discrete și independente, care sunt de așteptat să poată îndeplini sarcini (sarcini) în mod independent. Aceste module pot funcționa ca structuri de bază pentru întregul software. Proiectanții tind să proiecteze module astfel încât să poată fi executate și / sau compilate separat și independent.

Proiectarea modulară urmează în mod neintenționat regula strategiei de „rezolvare și împărțire" a problemei, acest lucru se datorează faptului că există multe alte avantaje asociate cu designul modular al unui software.

Avantajul modularizării:

• Componentele mai mici sunt mai ușor de întreținut

• Programul poate fi împărțit pe baza aspectelor funcționale

• Nivelul dorit de abstractizare poate fi adus în program

• Componentele cu coeziune ridicată pot fi refolosite din nou

• Execuția simultană poate fi posibilă

• Dorit din punct de vedere al securității

10.2 Concurență

Înapoi în timp, toate software-urile sunt menite să fie executate secvențial. Prin execuție secvențială, înțelegem că instrucțiunea codificată va fi executată una după alta, ceea ce implică activarea unei singure porțiuni de program la un moment dat. Spuneți că un software are mai multe module, apoi doar unul dintre toate modulele poate fi găsit activ în orice moment de execuție.

În proiectarea software-ului, concurența este implementată prin împărțirea software-ului în mai multe unități independente de execuție, cum ar fi module și executarea lor în paralel. Cu alte cuvinte, concurența oferă software-ului capacitatea de a executa mai multe părți de cod în paralel.

Este necesar ca programatorii și proiectanții să recunoască acele module, care pot fi executate în paralel.

Exemplu

Funcția de verificare ortografică din procesorul de text este un modul de software, care rulează de-a lungul procesorului de text în sine.

10.3 Cuplare și coeziune

Când un program software este modularizat, sarcinile sale sunt împărțite în mai multe module pe baza unor caracteristici. După cum știm, modulele sunt un set de instrucțiuni puse împreună pentru a îndeplini anumite sarcini. Acestea sunt, totuși, considerate ca o singură entitate, dar se pot referi reciproc pentru a lucra împreună. Există măsuri prin care se poate măsura calitatea unui design de module și interacțiunea lor între ele. Aceste măsuri se numesc cuplare și coeziune.

Coeziune

Coeziunea este o măsură care definește gradul de fiabilitate intra în elementele unui modul. Cu cât coeziunea este mai mare, cu atât designul programului este mai bun. Există șapte tipuri de coeziune și anume -

• **Coeziune coincidențială** - Este o coeziune neplanificată și aleatorie, care ar putea fi rezultatul ruperii programului în module mai mici de dragul modularizării. Deoarece este neplanificat, poate servi confuzie programatorilor și, în general, nu este acceptat.

• **Coeziune logică** - Când elementele clasificate logic sunt reunite într-un modul, se numește coeziune logică.

• **Coeziunea emporală** - Când elementele modulului sunt organizate astfel încât să fie procesate într-un moment similar al timpului, se numește coeziune temporală.

• **Coeziune procedurală** - Când elementele modulului sunt grupate împreună, care sunt executate secvențial pentru a îndeplini o sarcină, se numește coeziune procedurală.

• **Coeziune comunicațională** - Când elementele modulului sunt grupate împreună, care sunt executate secvențial și lucrează pe aceleași date (informații), se numește coeziune comunicațională.

• **Coeziune secvențială** - Când elementele modulului sunt grupate deoarece ieșirea unui element servește ca intrare la altul și așa mai departe, se numește coeziune secvențială.

• **Coeziune funcțională** - Este considerată a fi cel mai înalt grad de coeziune și este foarte așteptată. Elementele modulului de coeziune funcțională sunt grupate deoarece toate contribuie la o singură funcție bine definită. De asemenea, poate fi refolosit.

Cuplare

Cuplarea este o măsură care definește nivelul de interdependență între modulele unui program. Acesta spune la ce nivel modulele interferează și interacționează între ele. Cu cât cuplajul este mai mic, cu atât este mai bun programul.

Există cinci niveluri de cuplare, și anume -

• **Cuplarea conținutului** - Când un modul poate accesa sau modifica direct sau face referire la conținutul unui alt modul, acesta se numește cuplare la nivel de conținut.

• **Cuplare comună** - Când mai multe module au acces de citire și scriere la unele date globale, se numește cuplare comună sau globală.

• **Cuplare de control** - Două module se numesc cuplate de control dacă unul dintre ele decide funcția celuilalt modul sau își schimbă fluxul de execuție.

• **Cuplare cu ștampilă** - Atunci când mai multe module împart structura de date comună și lucrează pe diferite părți ale acesteia, se numește cuplare cu ștampilă.

• **Cuplarea datelor** - Cuplarea datelor este atunci când două module interacționează între ele prin transmiterea datelor (ca parametru). Dacă un modul trece structura de date ca parametru, atunci modulul de recepție ar trebui să folosească toate componentele sale.

În mod ideal, nici o cuplare nu este considerată a fi cea mai bună.

10.4 Verificarea proiectării

Rezultatul procesului de proiectare software este documentația de proiectare, pseudo-coduri, diagrame logice detaliate, diagrame de proces și descrierea detaliată a tuturor cerințelor funcționale sau nefuncționale.

Următoarea fază, care este implementarea software-ului, depinde de toate rezultatele menționate mai sus. Apoi devine necesar să verificați ieșirea înainte de a trece la faza următoare. Cu cât este detectată o eroare mai devreme, cu atât este mai bună sau este posibil să nu fie detectată până la testarea produsului. Dacă rezultatele fazei de proiectare sunt sub formă de notare formală, atunci instrumentele lor asociate pentru verificare ar trebui utilizate, în caz contrar, o verificare aprofundată a proiectării poate fi utilizată pentru verificare și validare.

Prin abordarea de verificare structurată, examinatorii pot detecta defecte care ar putea fi cauzate de trecerea cu vederea unor condiții. O analiză bună a proiectării este importantă pentru o bună proiectare a software-ului, precizie și calitate.

11. Analiza și Instrumente de proiectare Software

Analiza și proiectarea software-ului includ toate activitățile, care ajută la transformarea specificațiilor cerințelor în implementare. Specificațiile cerințelor specifică toate așteptările funcționale și nefuncționale de la software. Aceste specificații de cerință vin sub forma unor documente lizibile și ușor de înțeles de către oameni, la care un computer nu are nimic de făcut.

Analiza și proiectarea software-ului reprezintă etapa intermediară, care ajută la transformarea cerințelor care pot fi citite de om în cod real.

Să vedem câteva instrumente de analiză și proiectare utilizate de proiectanții de software:

11.1 Diagrama Fluxului de Date

Diagrama fluxului de date (**DFD**) este o reprezentare grafică a fluxului de date într-un sistem informațional. Este capabil să descrie fluxul de date primite, fluxul de date de ieșire și datele stocate. DFD nu menționează nimic despre modul în care datele circulă prin sistem.

Există o diferență importantă între DFD și diagramă de flux. Organigrama descrie fluxul de control în modulele de program. DFD-urile descriu fluxul de date în sistem la diferite niveluri. Nu conține niciun element de control sau ramificare.

11.2 Tipuri de DFD

Diagramele fluxului de date sunt fie logice, fie fizice.

• **DFD logic** - Acest tip de DFD se concentrează pe procesul sistemului și pe fluxul de date din sistem. De exemplu, într-un sistem software bancar, modul în care datele sunt mutate între diferite entități.

• **DFD fizic** - Acest tip de DFD arată modul în care fluxul de date este de fapt implementat în sistem. Este mai specific și apropiat de implementare.

11.3 Componente DFD

DFD poate reprezenta sursa, destinația, stocarea și fluxul de date utilizând următorul set de componente:

• **Entitate** - Entitățile sunt surse și destinații de date informaționale. Entitățile sunt reprezentate prin dreptunghiuri cu numele lor respective.

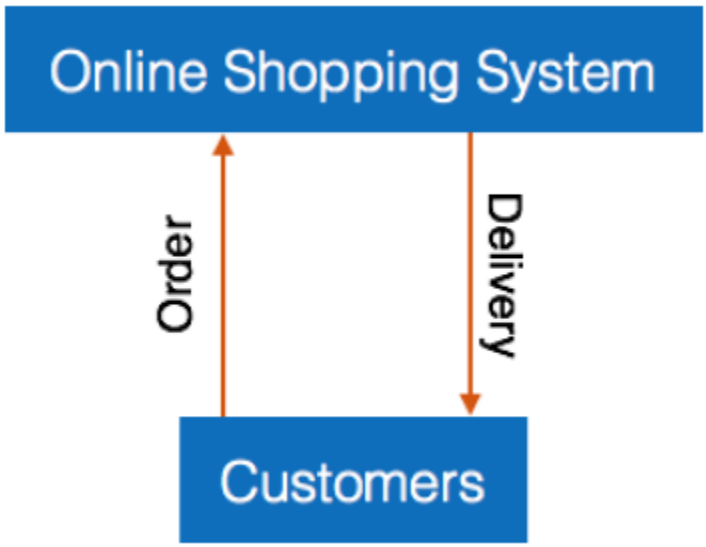
• **Proces** - Activitățile și acțiunile întreprinse asupra datelor sunt reprezentate de dreptunghiuri cu cerc sau cu muchii rotunde.

• **Stocarea datelor** - Există două variante de stocare a datelor - acesta poate fi reprezentat fie ca un dreptunghi cu absența ambelor laturi mai mici, fie ca un dreptunghi cu latură deschisă, lipsind doar o parte.

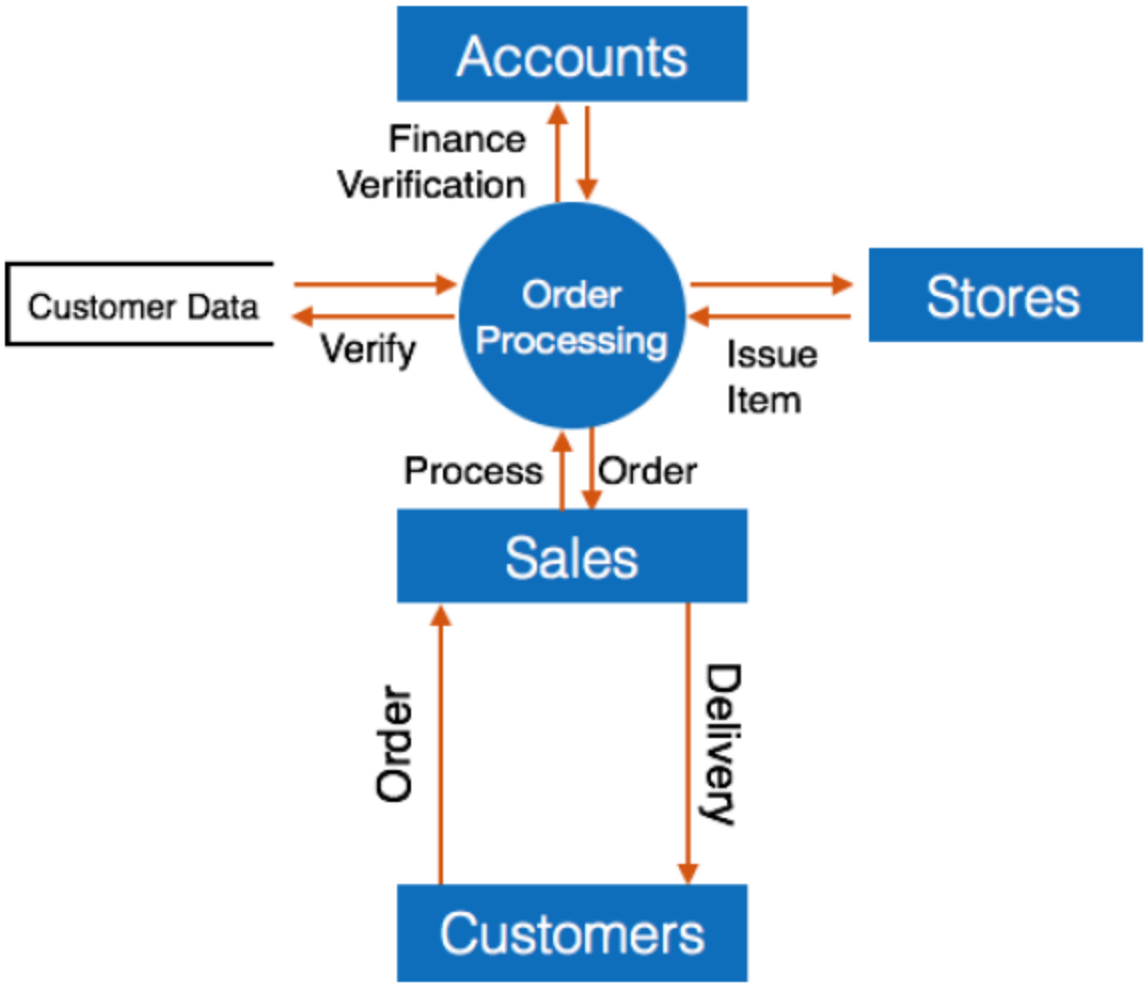
• **Flux de date** - Mișcarea datelor este afișată prin săgeți ascuțite. Mișcarea datelor este afișată de la baza săgeții ca sursă spre capul săgeții ca destinație.

11.4 Nivelurile DFD

• **Nivelul 0** - Cel mai înalt nivel de abstractizare DFD este cunoscut sub numele de Nivelul 0 DFD, care descrie întregul sistem informațional ca o singură diagramă care ascunde toate detaliile subiacente. DFD-urile de nivel 0 sunt, de asemenea, cunoscute sub numele de DFD-uri de context.



• **Nivelul 1** - Nivelul 0 DFD este împărțit în mai specific, nivelul 1 DFD. Nivelul 1 DFD descrie module de bază în sistem și fluxul de date între diferite module. Nivelul 1 DFD menționează, de asemenea, procesele de bază și sursele de informații.



• **Nivelul 2** - La acest nivel, DFD arată modul în care fluxurile de date în interiorul modulelor menționate la nivelul 1.

DFD-urile de nivel superior pot fi transformate în DFD-uri de nivel inferior mai specifice, cu un nivel mai profund de înțelegere, cu excepția cazului în care se atinge nivelul dorit de specificație.

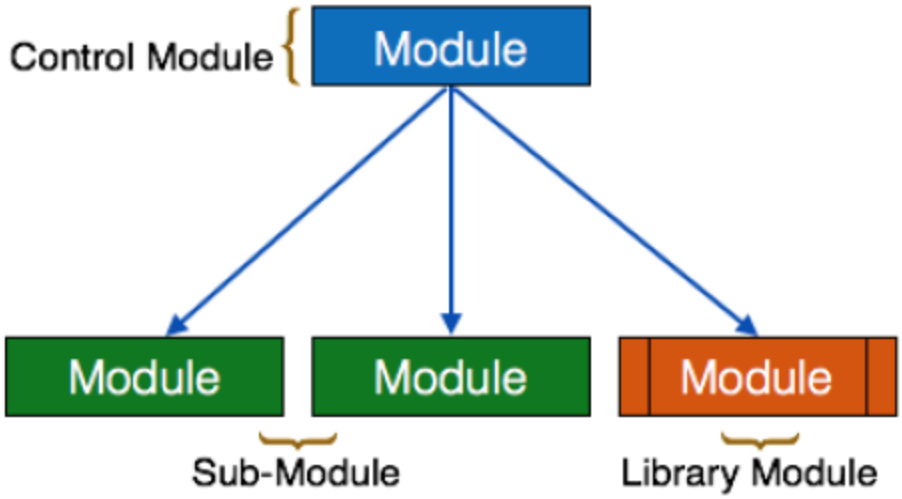
11.5 Diagramele structurale

Diagrama de structură este o diagramă derivată din diagrama fluxului de date. Reprezintă sistemul mai detaliat decât DFD. Descompune întregul sistem în cele mai mici module funcționale, descrie funcțiile și subfuncțiile fiecărui modul al sistemului într-un detaliu mai mare decât DFD.

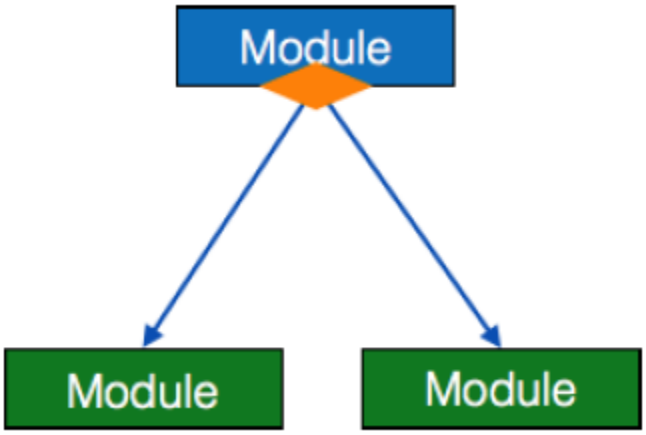
Structura diagramă reprezintă structura ierarhică a modulelor. La fiecare strat se efectuează o sarcină specifică.

Iată simbolurile utilizate în construcția diagramelor structurale -

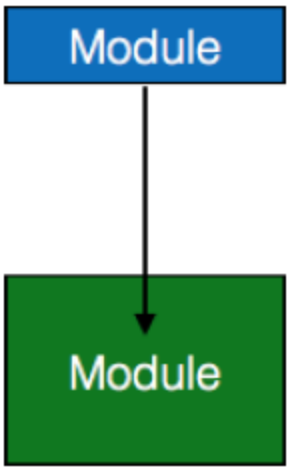
• **Modulul** - Reprezintă procesul sau subrutina sau sarcina. Un modul de control se ramifică în mai multe sub-module. Modulele de bibliotecă sunt reutilizabile și invocabile din orice modul.



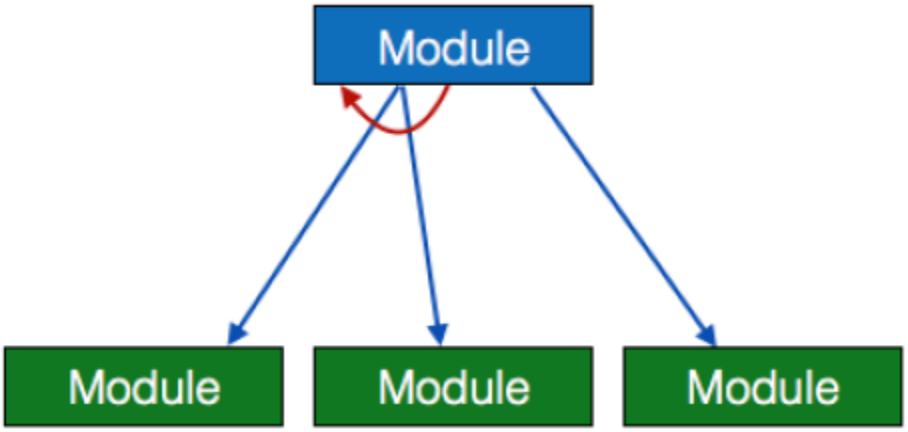
• **Conditia** - Este reprezentata de un mic romb la baza modulului prin care modulul de control poate selecta oricare dintre sub-rutine pe baza unor condiții.



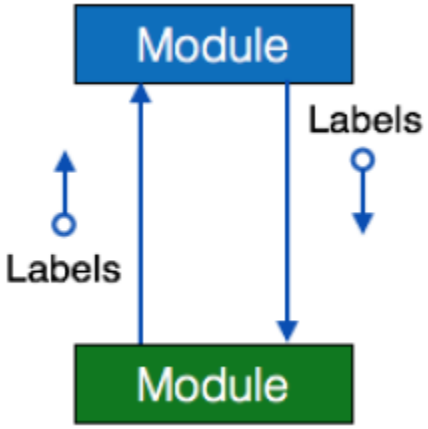
• **Saltul** - Este afișată o săgeată care tinteste în interiorul modulului pentru a arăta unde va sări controlul - în mijlocul sub-modulului.



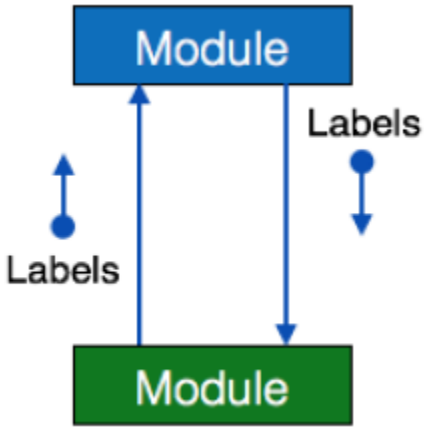
• **Buclă** - O săgeată curbată reprezintă o buclă în modul. Toate sub-modulele acoperite de buclă repetă execuția modulului.



• **Flux de date** - O săgeată direcționată cu un cerc gol la capăt reprezintă fluxul de date.



• **Flux de control** - O săgeată direcționată cu un cerc plin la capăt reprezintă fluxul de control.

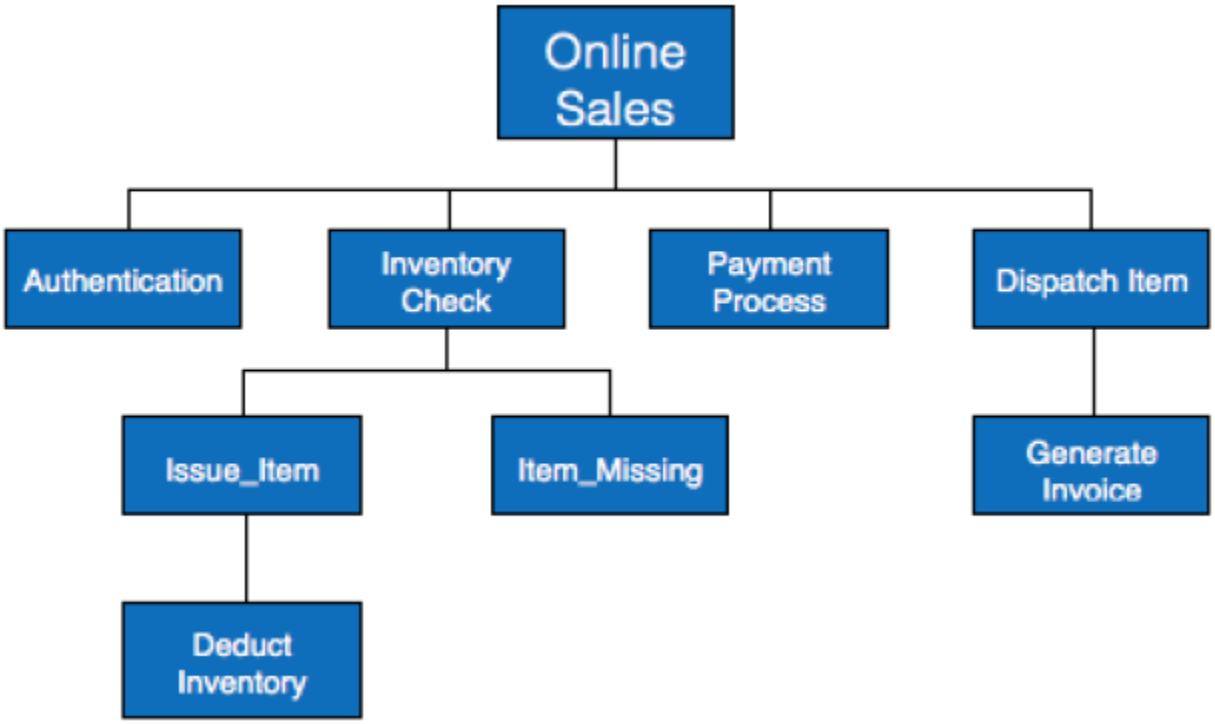


11.6 Diagrama HIPO

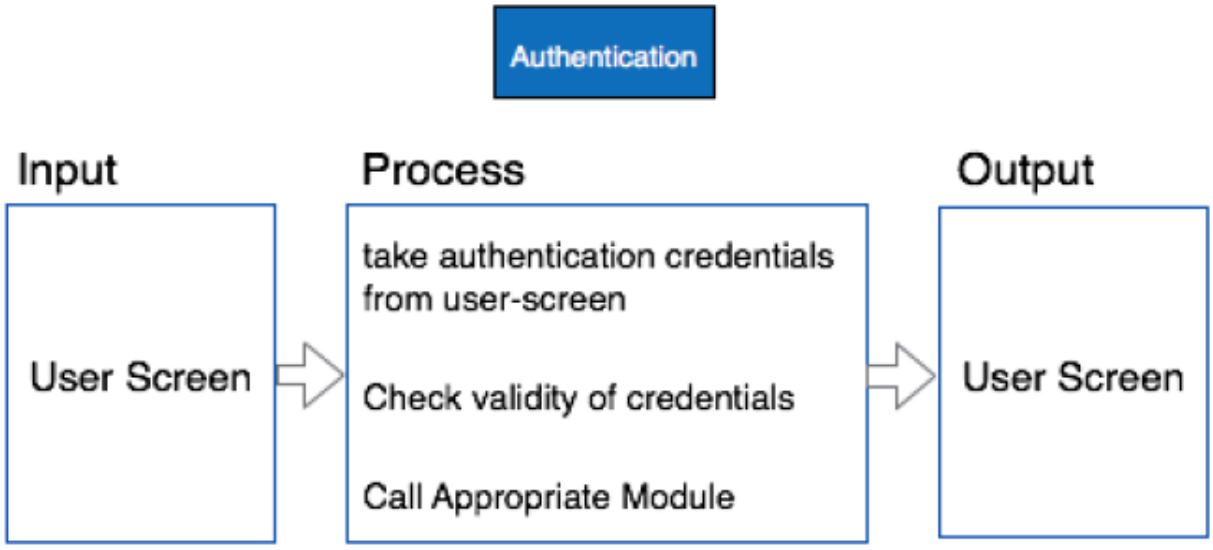
Diagrama Ierarhică Intrare Procesare Iesire **(HIPO**) este o combinație a două metode organizate pentru a analiza sistemul și a furniza mijloacele de documentare. Modelul HIPO a fost dezvoltat de IBM în anul 1970.

Diagrama HIPO reprezintă ierarhia modulelor din sistemul software. Analistul folosește diagrama HIPO pentru a obține o vizualizare la nivel înalt a funcțiilor sistemului. Descompune funcțiile în subfuncții într-un mod ierarhic. Acesta descrie funcțiile îndeplinite de sistem.

Diagramele HIPO sunt bune pentru documentare. Reprezentarea lor grafică face, pentru proiectanți și manageri, să obțină mai ușor ideea generala a structurii sistemului.



Spre deosebire de *diagrama Intrare Procesare Iesire* (IPO), care descrie fluxul de control și date într-un modul, HIPO nu oferă nicio informație despre fluxul de date sau fluxul de control.



Exemplu

Ambele părți ale diagramei HIPO, prezentarea ierarhică și diagrama IPO sunt utilizate pentru proiectarea structurii programului software, precum și pentru documentarea acestora.

11.7 Engleză structurată

Majoritatea programatorilor nu sunt conștienți de imaginea de ansamblu a software-ului, așa că se bazează doar pe ceea ce managerii lor le spun să facă. Este responsabilitatea unui management superior al software-ului să furnizeze programatorilor informații exacte pentru a dezvolta un cod precis, dar rapid.

Diferite metode, care utilizează grafice sau diagrame, pot fi uneori interpretate într-un mod diferit de către diferiți oameni.

Prin urmare, analiștii și proiectanții software-ului vin cu instrumente precum engleza structurată. Nu este altceva decât descrierea a ceea ce este necesar pentru codificare și modul de codificare. Engleza structurată ajută programatorul să scrie cod fără erori. Aici, atât limba engleză structurată, cât și pseudo-codul încearcă să atenueze acest decalaj de înțelegere.

Engleza structurată folosește cuvinte simple în engleză în paradigma de programare structurată. Nu este codul final, ci un fel de descriere, ceea ce este necesar pentru a codifica și cum să îl codificați. Următoarele sunt câteva jetoane de programare structurată:

```
IF-THEN-ELSE,

DO-WHILE-UNTIL
```

Analistul folosește aceeași variabilă și același nume de date, care sunt stocate în Dicționarul de date, ceea ce face mult mai simplu să scrieți și să înțelegeți codul.

Exemplu

Luăm același exemplu de autentificare a clienților în mediul de cumpărături online. Această procedură pentru autentificarea clientului poate fi scrisă în limba engleză structurată ca:

```
Enter Customer_Name
SEEK Customer_Name in Customer_Name_DB file
IF Customer_Name found THEN
  Call procedure USER_PASSWORD_AUTHENTICATE()
ELSE
  PRINT error message
  Call procedure NEW_CUSTOMER_REQUEST()
ENDIF
```

Codul scris în limba engleză structurată seamănă mai mult cu limba engleză vorbită de zi cu zi. Nu poate fi implementat direct ca un cod de software. Engleza structurată este independentă de limbajul de programare.

11.8 Pseudo cod

Pseudo codul este scris mai aproape de limbajul de programare. Poate fi considerat un limbaj de programare mărit, plin de comentarii și descrieri.

Pseudo-codul evită declarația variabilă, dar acestea sunt scrise folosind unele construcții reale ale limbajului de programare, cum ar fi C, Fortran, Pascal etc.

Pseudo codul conține mai multe detalii de programare decât engleza structurată. Acesta oferă o metodă pentru a efectua sarcina, ca și cum un computer execută codul.

Exemplu

Program pentru a imprima Fibonacci până la **n** numere:

```
void function Fibonacci
Get value of n;
Set value of a to 1;
Set value of b to 1;
Initialize I to 0
for (i=0; i< n; i++)
{
  if a greater than b
  {
    Increase b by a;
    Print b;
  }
  else if b greater than a
  {
    increase a by b;
    print a;
  }
}
```

11.9 Tabelele de decizie

Un tabel de decizie reprezintă condițiile și acțiunile respective care trebuie întreprinse pentru a le aborda, într-un format tabelar structurat.

Este un instrument puternic de depanare și prevenire a erorilor. Ajută la gruparea informațiilor similare într-un singur tabel și apoi prin combinarea tabelelor oferă o decizie ușoară și convenabilă.

Crearea tabelului decizional

Pentru a crea tabelul de decizie, dezvoltatorul trebuie să urmeze patru pași de bază:

• Identificați toate condițiile posibile care trebuie abordate

• Determinați acțiunile pentru toate condițiile identificate

• Creați reguli maxime posibile

• Definiți acțiunea pentru fiecare regulă

Tabelele de decizie ar trebui verificate de către utilizatorii finali și pot fi simplificate în ultima instanta prin eliminarea regulilor și acțiunilor duplicate.

Exemplu
Să luăm un exemplu simplu de problemă de zi cu zi, conectivitatea noastră la Internet. Începem prin identificarea tuturor problemelor care pot apărea în timpul pornirii internetului și a posibilelor soluții ale acestora.

	Conditions/Actions	Rules							
Conditions	Shows Connected	N	N	N	N	Y	Y	Y	Y
	Ping is Working	N	N	Y	Y	N	N	Y	Y
	Opens Website	Y	N	Y	N	Y	N	Y	N
Actions	Check network cable	X							
	Check internet router	X				X	X	X	
	Restart Web Browser							X	
	Contact Service provider		X	X	X	X	X	X	
	Do no action								

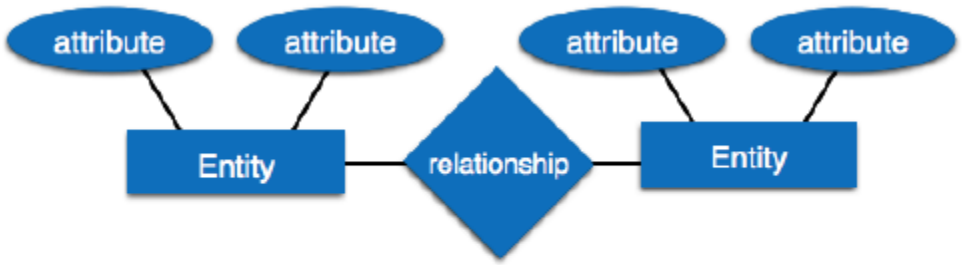
Table : Decision Table – In-house Internet Troubleshooting

Enumerăm toate problemele posibile în condițiile coloanei și acțiunile potențiale din coloana Acțiuni.

11.10 Modelul entitate-relație

Modelul entitate-relație este un tip de model de bază de date bazat pe noțiunea de entități din lumea reală și relația dintre acestea. Putem mapa scenariul din lumea reală pe modelul bazei de date ER. Modelul ER creează un set de entități cu atributele lor, un set de constrângeri și relații între ele.

Modelul ER este cel mai bine utilizat pentru proiectarea conceptuală a bazei de date. Modelul ER poate fi reprezentat după cum urmează:



• **Entitate** - O entitate din modelul ER este o ființă din lumea reală, care are unele proprietăți numite atribute. Fiecare atribut este definit de setul său corespunzător de valori, numit domeniu.

De exemplu, luați în considerare o bază de date școlară. Aici, un student este o entitate. Elevul are diverse atribute precum numele, id-ul, vârsta și clasa etc.

• **Relație** - Asocierea logică între entități se numește relație. Relațiile sunt mapate cu entități în diferite moduri. Cardinalitățile cartografice definesc numărul de asocieri între două entități.

Maparea cardinalităților:

• unu la unu

• unul la mulți

• mulți la unu

• mulți la mulți

Dicționar de date

Dicționarul de date este colecția centralizată de informații despre date. Stochează semnificația și originea datelor, relația sa cu alte date, formatul de date pentru utilizare etc. Dicționarul de date are definiții riguroase ale tuturor denumirilor pentru a facilita proiectanții utilizatorilor și software-ului.

Dicționarul de date este adesea menționat ca depozit de meta-date (date despre date). Este creat împreună cu modelul programului software DFD (Diagrama fluxului de date) și se așteaptă să fie actualizat ori de câte ori DFD este modificat sau actualizat.

Cerința de dicționar de date

Datele sunt trimise prin intermediul dicționarului de date în timp ce proiectăm și implementăm software. Dicționarul de date elimină orice șansă de ambiguitate. Ajută la menținerea sincronizată a activității programatorilor și a proiectanților în timp ce folosește același obiect de referință peste tot în program.

Dicționarul de date oferă o modalitate de documentare pentru sistemul complet de baze de date într-un singur loc. Validarea DFD se efectuează folosind dicționarul de date.

Continut
Dicționarul de date ar trebui să conțină informații despre următoarele:

• Flux de date

• Structură de date

• Elemente de date

• Magazine de date

• Procesarea datelor

Fluxul de date este descris prin intermediul DFD-urilor, așa cum a fost studiat mai devreme și reprezentat în formă algebrică așa cum a fost descris.

=	**Composed of**
{}	Repetition
()	Optional
+	And
[/]	Or

Exemplu
Adresa = Casa nr + (stradă / zonă) + oraș + stat

ID curs = Număr curs + Denumire curs + Nivel curs + Note curs

Elemente de date
Elementele de date constau din numele și descrierile articolelor de date și control, stocări de date interne sau externe etc., cu următoarele detalii:

• Numele principal

• Nume secundar (Alias)

• Caz de utilizare (Cum și unde să se utilizeze)

• Descrierea conținutului (notație etc.)

• Informații suplimentare (valori prestabilite, constrângeri etc.)

Stocarea datelor
Stochează informațiile de unde datele intră în sistem și există în afara sistemului. Magazinul de date poate include -

• **Fișiere**

- o interne software-ului.
- o Externe pentru software, dar pe aceeași mașină.
- o Externe pentru software și sistem, situate pe diferite mașini.

• **Tabelele**

- o Convenția de denumire
- o Proprietatea de indexare

Procesarea datelor
Există două tipuri de prelucrare a datelor:

• **Logică**: după cum vede utilizatorul

• **Fizica**: așa cum vede software-ul

12. Strategii Pentru Dezvoltare Software

Proiectarea software-ului este un proces de conceptualizare a cerințelor software în implementarea software-ului. Proiectarea software-ului ia cerințele utilizatorului drept provocări și încearcă să găsească soluția optimă. În timp ce software-ul este conceptualizat, se planifică un plan pentru a găsi cel mai bun design posibil pentru implementarea soluției intenționate.

Există mai multe variante de proiectare software. Să le studiem pe scurt:

12.1 Proiectare Structurată

Proiectarea structurată este o conceptualizare a problemei în mai multe elemente de soluție bine organizate. Practic este preocupat de proiectarea soluției. Avantajul proiectării structurate este că oferă o mai bună înțelegere a modului în care se rezolvă problema. Designul structurat face, de asemenea, mai simplu pentru designer să se concentreze asupra problemei mai precis.

Proiectarea structurată se bazează în cea mai mare parte pe strategia „împărțiți și cuceriți", unde o problemă este împărțită în mai multe probleme mici și fiecare problemă mică este rezolvată individual până când se rezolvă întreaga problemă.

Problemele mici sunt rezolvate prin intermediul modulelor de soluție. Accentuarea proiectării structurate ca aceste module să fie bine organizate pentru a obține o soluție precisă.

Aceste module sunt aranjate în ierarhie. Ei comunică între ei. Un design bun structurat respectă întotdeauna unele reguli pentru comunicarea între mai multe module, și anume -

• **Coeziune** - gruparea tuturor elementelor legate funcțional.

• **Cuplare** - comunicare între diferite module.

Un design bine structurat are coeziune ridicată și aranjamente de cuplare reduse.

12.2 Proiectare orientată pe funcții

În proiectarea orientată spre funcții, sistemul cuprinde mai multe subsisteme mai mici cunoscute sub numele de funcții. Aceste funcții sunt capabile să îndeplinească sarcini semnificative în sistem. Sistemul este considerat ca vedere de sus a tuturor funcțiilor.

Proiectarea orientată pe funcții moștenește unele proprietăți ale proiectării structurate unde se folosește metodologia de divizare și cucerire.

Acest mecanism de proiectare împarte întregul sistem în funcții mai mici, care oferă mijloace de abstractizare prin ascunderea informațiilor și funcționarea acestora. Aceste module funcționale pot partaja informații între ele prin transmiterea informațiilor și utilizarea informațiilor disponibile la nivel global.

O altă caracteristică a funcțiilor este că atunci când un program apelează o funcție, funcția schimbă starea programului, ceea ce uneori nu este acceptat de alte module. Proiectarea orientată pe funcții funcționează bine acolo unde starea sistemului nu contează și programul / funcțiile funcționează pe intrare mai degrabă decât pe o stare.

Proces de design

• Întregul sistem este văzut ca modul în care datele circulă în sistem prin intermediul unei diagrame de flux de date.

• DFD descrie modul în care funcțiile modifică datele și starea întregului sistem.

• Întregul sistem este defalcat logic în unități mai mici cunoscute sub numele de funcții pe baza funcționării lor în sistem.

• Fiecare funcție este apoi descrisă în general.

12.3 Proiectare orientată pe obiecte

Proiectarea orientată pe obiecte (OOD) funcționează în jurul entităților și caracteristicilor acestora în locul funcțiilor implicate în sistemul software. Aceste strategii de proiectare se concentrează pe entități și caracteristicile sale. Întregul concept de soluție software se învârte în jurul entităților angajate.

Să vedem conceptele importante ale proiectării orientate pe obiecte:

• **Obiecte** - Toate entitățile implicate în proiectarea soluției sunt cunoscute sub numele de obiecte. De exemplu, persoana, băncile, compania și clienții sunt tratați ca obiecte. Fiecare entitate are asociate niște atribute și are câteva metode de efectuat asupra atributelor.

• **Clase** - O clasă este o descriere generalizată a unui obiect. Un obiect este o instanță a unei clase. Clasa definește toate atributele pe care le poate avea un obiect și metodele care definesc funcționalitatea obiectului. În proiectarea soluției, atributele sunt stocate ca variabile și funcționalitățile sunt definite prin intermediul metodelor sau procedurilor.

• **Incapsulare** - În OOD, atributele (variabilele de date) și metodele (operația pe date) sunt grupate împreună se numește încapsulare. Incapsularea nu numai că grupează informații importante despre un

obiect, ci și restricționează accesul la date și metode din lumea exterioară. Aceasta se numește ascunderea informațiilor.

• **Moștenire** - OOD permite claselor similare să se acumuleze într-un mod ierarhic în care clasele inferioare sau secundare pot importa, implementa și reutiliza variabilele și metodele permise din superclasele lor imediate. Această proprietate a OOD este cunoscută sub numele de moștenire. Acest lucru face mai ușor să definiți anumite clase și să creați clase generalizate din clase specifice.

• **Polimorfism** - limbajele OOD oferă un mecanism în care metodelor care îndeplinesc sarcini similare, dar variază în argumente, li se poate atribui același nume. Aceasta se numește polimorfism, care permite unei singure interfețe să efectueze sarcini pentru diferite tipuri. În funcție de modul în care este invocată funcția, porțiunea respectivă a codului este executată.

12.4 Proces de design

Procesul de proiectare software poate fi perceput ca o serie de pași bine definiți. Deși variază în funcție de abordarea de proiectare (orientată spre funcții sau orientată spre obiect, totuși poate avea următorii pași implicați:

• O soluție de proiectare este creată din cerințe sau din sistemul anterior utilizat și / sau din diagrama de secvență a sistemului.

• Obiectele sunt identificate și grupate în clase în numele similarității caracteristicilor atributelor.

• Ierarhia claselor și relația dintre acestea este definită.

• Cadrul de aplicație este definit.

13. Abordări de Proiectare Software

Iată două abordări generice pentru proiectarea software-ului:

13.1 Design de sus în jos

Știm că un sistem este compus din mai multe subsisteme și conține o serie de componente. Mai mult, aceste subsisteme și componente pot avea propriul set de subsisteme și componente și creează o structură ierarhică în sistem.

Proiectarea de sus în jos ia întregul sistem software ca o singură entitate și apoi îl descompune pentru a obține mai mult de un subsistem sau componentă pe baza unor caracteristici. Fiecare subsistem sau componentă este apoi tratat ca un sistem și descompus în continuare. Acest proces continuă să ruleze până la atingerea celui mai scăzut nivel al sistemului din ierarhia de sus în jos.

Proiectarea de sus în jos începe cu un model generalizat de sistem și continuă să definească partea mai specifică a acestuia. Când toate componentele sunt alcătuite, întregul sistem apare.

Designul de sus în jos este mai potrivit atunci când soluția software trebuie proiectată de la zero și nu se cunosc detalii specifice.

13.2 Design de jos în sus

Modelul de proiectare de jos în sus începe cu majoritatea componentelor specifice și de bază. Se continuă cu compunerea unui nivel superior de componente utilizând componente de bază sau de nivel inferior.

Continuă să creeze componente de nivel superior până când sistemul dorit nu este dezvoltat ca o singură componentă. Cu fiecare nivel superior, cantitatea de abstractizare este crescută.

Strategia de jos în sus este mai potrivită atunci când trebuie creat un sistem dintr-un sistem existent, unde primitivele de bază pot fi utilizate în sistemul mai nou.

Ambele abordări de sus în jos și de jos în sus nu sunt practice individual. În schimb, se folosește o combinație bună a ambelor.

14. Proiectarea Interfeței Software cu Utilizatorul

Interfața cu utilizatorul este vizualizarea aplicației front-end la care utilizatorul interacționează pentru a utiliza software-ul. Utilizatorul poate manipula și controla software-ul, precum și hardware-ul prin intermediul interfeței cu utilizatorul. Astăzi, interfața cu utilizatorul se găsește în aproape orice loc în care există tehnologia digitală, chiar de la computere, telefoane mobile, mașini, playere muzicale, avioane, nave etc.

Interfața cu utilizatorul face parte din software și este concepută în așa fel încât este de așteptat să ofere utilizatorului informații despre software. UI oferă o platformă fundamentală pentru interacțiunea om-computer.

UI poate fi grafică, bazată pe text, audio-video, în funcție de combinația hardware și software care stau la baza. UI poate fi hardware sau software sau o combinație a ambelor.

Software-ul devine mai popular dacă interfața sa de utilizator este:

• Atractiv

• Simplu de utilizat

• Răspunde în scurt timp

• Clar de înțeles

• În conformitate cu toate ecranele de interfață, interfața de utilizare este în general împărțită în două categorii:

• Linia de comandă

• Interfață grafică pentru utilizator

14.1 Interfață linie de comandă (CLI)

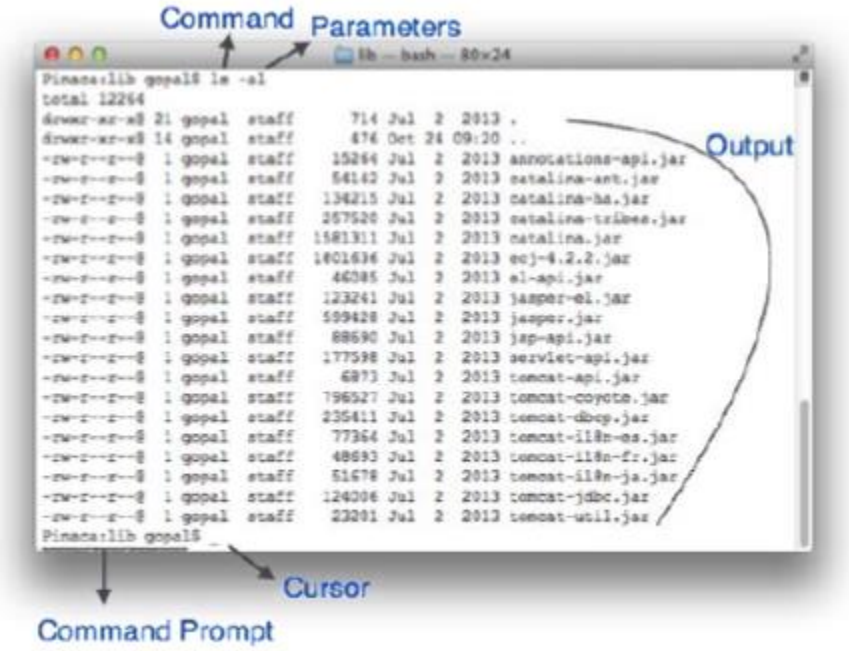
CLI a fost un instrument excelent de interacțiune cu computerele până când au apărut monitoarele de afișare video. CLI este prima alegere a multor utilizatori și programatori tehnici. Este interfața minimă pe care un software o poate oferi utilizatorilor săi.

CLI oferă un prompt de comandă, locul în care utilizatorul tastează comanda și alimentează sistemul. Utilizatorul trebuie să rețină sintaxa comenzii și utilizarea acesteia. CLI anterioare nu erau programate pentru a gestiona în mod eficient erorile utilizatorului.

O comandă este o referință bazată pe text la setul de instrucțiuni, care se așteaptă să fie executate de sistem. Există metode cum ar fi macrocomenzile, scripturi care facilitează utilizarea utilizatorului.

CLI utilizează o cantitate mai mică de resurse pentru computer în comparație cu GUI.

Elemente CLI



O interfață de linie de comandă bazată pe text poate avea următoarele elemente:

• **Prompter de comandă** - Este un notificator bazat pe text care arată în principal contextul în care lucrează utilizatorul. Este generat de sistemul software.

• **Cursor** - Este o linie orizontală mică sau o bară verticală a înălțimii liniei, pentru a reprezenta poziția caracterului în timp ce tastați. Cursorul se găsește mai ales în stare intermitentă. Se mișcă pe măsură ce utilizatorul scrie sau șterge ceva.

• **Comandă** - O comandă este o instrucțiune executabilă. Poate avea unul sau mai mulți parametri. Ieșirea la executarea comenzii este afișată în linie pe ecran. Când se produce ieșire, promptul de comandă este afișat pe linia următoare.

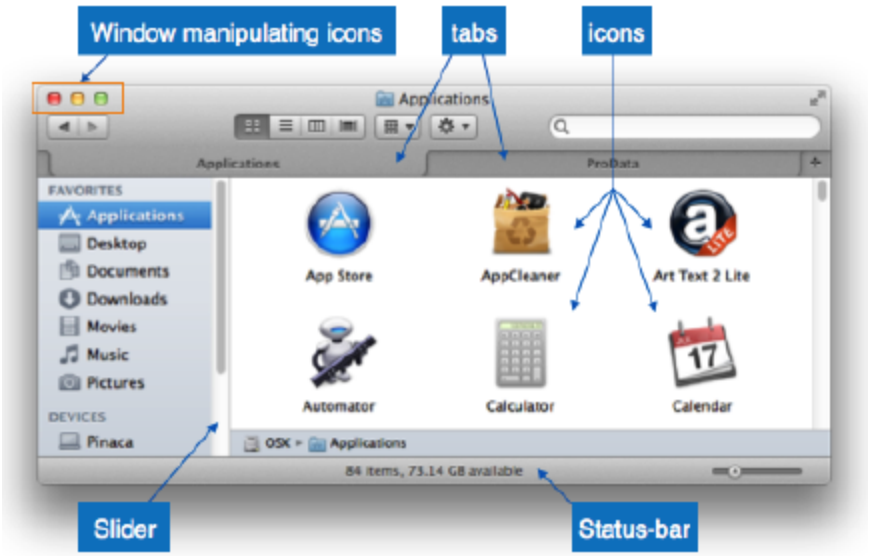
14.2 Interfață grafică pentru utilizator

Interfața grafică de utilizator (GUI) oferă utilizatorului mijloace grafice pentru a interacționa cu sistemul. GUI poate fi o combinație atât de hardware, cât și de software. Utilizând GUI, utilizatorul interpretează software-ul.

De obicei, GUI consumă mai multe resurse decât cea a CLI. Cu o tehnologie avansată, programatorii și designerii creează modele GUI complexe care funcționează cu mai multă eficiență, precizie și viteză.

Elemente GUI

GUI oferă un set de componente pentru a interacționa cu software sau hardware.



Fiecare componentă grafică oferă o modalitate de a lucra cu sistemul. Un sistem GUI are următoarele elemente, cum ar fi:

Fereastra - O zonă în care este afișat conținutul aplicației. Conținutul unei ferestre poate fi afișat sub formă de pictograme sau liste, dacă fereastra reprezintă structura fișierului. Este mai ușor pentru un utilizator să navigheze în sistemul de fișiere într-o fereastră de explorare. Windows-ul poate fi minimizat, redimensionat sau maximizat la dimensiunea ecranului. Ele pot fi mutate oriunde pe ecran. O fereastră poate conține o altă fereastră a aceleiași aplicații, numită fereastră copil.

• Tabs - Dacă o aplicație permite executarea mai multor instanțe, ele apar pe ecran ca ferestre separate. Interfața cu documente cu file a apărut pentru a deschide mai multe documente în aceeași fereastră. Această interfață ajută, de asemenea, la vizualizarea panoului de preferințe în aplicație. Toate browserele web moderne folosesc această caracteristică.

• Menu- Meniul este o serie de comenzi standard, grupate împreună și plasate într-un loc vizibil (de obicei sus) în fereastra aplicației. Meniul poate fi programat să apară sau să se ascundă la clicurile mouse-ului.

• Icon - O pictogramă este o imagine mică care reprezintă o aplicație asociată. Când se fac clic sau se face dublu clic pe aceste pictograme, se deschide fereastra aplicației. Pictograma afișează aplicația și programele instalate pe un sistem sub formă de imagini mici.

• Cursor - Dispozitivele care interacționează, cum ar fi mouse-ul, touch pad-ul, stiloul digital sunt reprezentate în GUI ca cursori. Cursorul de pe ecran urmează instrucțiunile din hardware aproape în timp real. Cursorii sunt, de asemenea, numiți indicatori în sistemele GUI. Acestea sunt folosite pentru a selecta meniuri, ferestre și alte caracteristici ale aplicației.

14.3 Componente GUI specifice aplicației

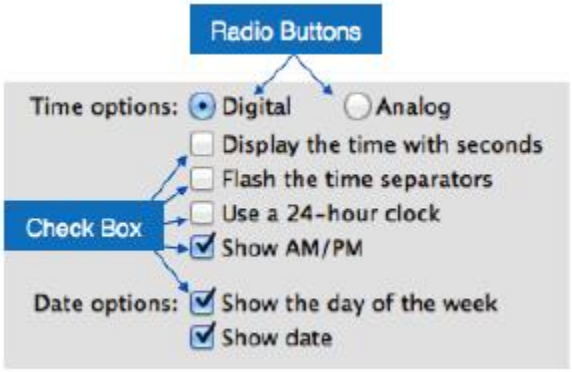
Un GUI al unei aplicații conține unul sau mai multe dintre elementele GUI enumerate:

• **Fereastra aplicației** - Majoritatea ferestrelor aplicației utilizează construcțiile furnizate de sistemele de operare, dar multe utilizează propriile ferestre create de clienți pentru a conține conținutul aplicației.

• **Casetă de dialog** - Este o fereastră copil care conține un mesaj pentru utilizator și solicită o acțiune. De exemplu: aplicația generează un dialog pentru a primi confirmarea de la utilizator pentru a șterge un fișier.

• **Casetă text** - Oferă o zonă în care utilizatorul poate introduce și introduce date bazate pe text.

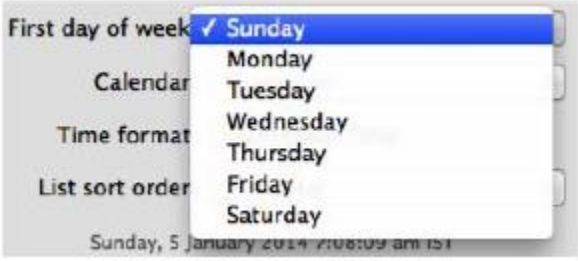
• **Butoane** - Acestea imită butoanele din viața reală și sunt utilizate pentru a trimite intrări către software.



• **Buton radio** - Afișează opțiunile disponibile pentru selectare. Doar unul poate fi selectat dintre toate ofertele.

• **Casetă de selectare** - Funcții similare casetei de listă. Când este selectată o opțiune, caseta este marcată ca bifată. Pot fi selectate mai multe opțiuni reprezentate de casete de selectare.

• **List-box** - Oferă o listă a articolelor disponibile pentru selecție. Pot fi selectate mai multe elemente.



Alte componente GUI impresionante sunt:

• Sliders

• Combo-box

• Data-grid

• Lista Drop-Down

14.4 Activități de proiectare a interfeței utilizatorului

Există o serie de activități efectuate pentru proiectarea interfeței cu utilizatorul. Procesul de proiectare și implementare a interfeței grafice este similar CVDS. Orice model poate fi utilizat pentru implementarea GUI în cascadă, model iterativ sau spiralat.

Un model utilizat pentru proiectarea și dezvoltarea GUI ar trebui să îndeplinească acești pași specifici GUI.

• **Centralizarea cerințelor GUI** - Designerii pot dori să aibă o listă cu toate cerințele funcționale și nefuncționale ale GUI. Acest lucru poate fi preluat de la utilizator și de la soluția software existentă.

• **Analiza utilizatorului** - Proiectantul studiază cine va folosi software-ul GUI. Publicul țintă contează, deoarece detaliile de proiectare se modifică în funcție de cunoștințele și nivelul de competență al utilizatorului. Dacă utilizatorul are cunoștințe tehnice, pot fi încorporate GUI complexe și avansate. Pentru un utilizator începător, sunt incluse mai multe informații despre modul de utilizare a software-ului.

• **Analiza sarcinilor** - Proiectanții trebuie să analizeze ce sarcină trebuie realizată de soluția software. Aici în GUI, nu contează cum se va face. Sarcinile pot fi reprezentate în mod ierarhic luând o sarcină majoră și împărțind-o în continuare în sub-sarcini mai mici. Sarcinile oferă obiective pentru prezentarea GUI. Fluxul de informații între sub-sarcini determină fluxul de conținut GUI în software.

• **Proiectare și implementare GUI** - Proiectanții, după ce au informații despre cerințe, sarcini și mediul utilizatorului, proiectează GUI și implementează în cod și încorporează GUI cu software-ul de lucru sau fals în fundal. Apoi este auto-testat de către dezvoltatori.

• **Testare** - Testarea GUI se poate face în diferite moduri. Organizația poate avea inspecții interne, implicarea directă a utilizatorilor și lansarea versiunii beta sunt puține dintre acestea. Testarea poate include uzabilitate, compatibilitate, acceptarea utilizatorului etc.

14.5 Instrumente de implementare GUI

Există mai multe instrumente disponibile cu ajutorul cărora proiectanții pot crea GUI întregi printr-un clic de mouse. Unele instrumente pot fi încorporate în mediul software (IDE).

Instrumentele de implementare GUI oferă o gamă puternică de controale GUI. Pentru personalizarea software-ului, proiectanții pot modifica codul în consecință.

Există diferite segmente de instrumente GUI în funcție de utilizarea și platforma lor diferită.

Exemplu

Mobile GUI, Computer GUI, Touch-Screen GUI etc. Iată o listă cu câteva instrumente care sunt utile pentru a construi GUI:

• FLUID

• AppInventor (Android)

• LucidChart

• Wavemaker

• Visual Studio

14.6 Interfața utilizatorului - Regulile de aur

Următoarele reguli sunt menționate a fi regulile de aur pentru proiectarea GUI, descrise de Shneiderman și Plaisant în cartea lor (Designing the User Interface).

• **Străduiți-vă pentru consecvență** - În situații similare ar trebui să fie necesare secvențe consecvente de acțiuni. Terminologia identică trebuie utilizată în solicitări, meniuri și ecrane de ajutor. Comenzile coerente ar trebui folosite pe tot parcursul.

• **Permiteți utilizatorilor frecvenți să utilizeze comenzile rapide** - Dorința utilizatorului de a reduce numărul de interacțiuni crește odată cu frecvența de utilizare. Abrevierile, tastele funcționale, comenzile ascunse și facilitățile macro sunt foarte utile pentru un utilizator expert.

• **Oferiți feedback informativ** - Pentru fiecare acțiune a operatorului, ar trebui să existe un feedback al sistemului. Pentru acțiunile frecvente și minore, răspunsul trebuie să fie modest, în timp ce pentru acțiunile rare și majore, răspunsul trebuie să fie mai substanțial.

• **Creați un dialog pentru a obține închiderea** - Secvențele acțiunilor ar trebui organizate în grupuri cu început, mijloc și sfârșit. Feedback-ul informativ la finalizarea unui grup de acțiuni oferă operatorilor satisfacția realizării, un sentiment de ușurare, semnalul de a renunța la planurile și opțiunile de urgență

din mintea lor, iar acest lucru indică faptul că calea de urmat este clară pentru pregătirea pentru următoarea grup de acțiuni.

• **Oferiți o gestionare simplă a erorilor** - Pe cât posibil, proiectați sistemul astfel încât utilizatorul să nu comită o eroare gravă. Dacă se face o eroare, sistemul ar trebui să fie capabil să o detecteze și să ofere mecanisme simple și ușor de înțeles pentru gestionarea erorii.

• **Permiteți inversarea ușoară a acțiunilor** - Această caracteristică ameliorează anxietatea, deoarece utilizatorul știe că erorile pot fi anulate. Inversarea ușoară a acțiunilor încurajează explorarea opțiunilor necunoscute. Unitățile de reversibilitate pot fi o singură acțiune, o introducere de date sau un grup complet de acțiuni.

• **Sprijinirea locusului intern de control** - Operatorii experimentați doresc cu tărie sa simta că sunt responsabili de sistem și că sistemul răspunde acțiunilor lor. Proiectați sistemul pentru a face utilizatorii mai degrabă inițiatorii acțiunilor decât respondenții.

• **Reduceți încărcarea memoriei pe termen scurt** - Limitarea procesării informațiilor umane în memoria pe termen scurt necesită menținerea simplă a afișajelor, consolidarea afișajelor cu mai multe pagini, reducerea frecvenței mișcării ferestrei și alocarea unui timp suficient de antrenament pentru coduri, mnemonice și secvențe de acțiuni.

15. Complexitatea Proiectarii Software

Termenul de complexitate se refera la starea evenimentelor sau a lucrurilor, care au mai multe legături interconectate și structuri extrem de complicate. În programarea software, pe măsură ce se realizează proiectarea software-ului, numărul de elemente și interconectările lor treptat devin imense, ceea ce devine prea dificil de înțeles deodată.

Complexitatea proiectării software-ului este dificil de evaluat fără utilizarea metricilor și măsurilor de complexitate. Să vedem trei măsuri importante de complexitate software.

15.1 Măsurile de complexitate ale lui Halstead

În 1977, Maurice Howard Halstead a introdus metrici pentru a măsura complexitatea software-ului. Metricele Halstead depind de implementarea efectivă a programului și măsurile sale, care sunt calculate direct de la operatori și operanzi din codul sursă, în mod static. Permite evaluarea timpului de testare, vocabularului, dimensiunii, dificultății, erorilor și eforturilor pentru codul sursă C / C ++ / Java.

Potrivit lui Halstead, „Un program de calculator este o implementare a unui algoritm considerat a fi o colecție de jetoane care pot fi clasificate fie ca operatori, fie ca operanzi". Metricele Halstead consideră un program ca o succesiune de operatori și operanzi asociați lor.

El definește diverși indicatori pentru a verifica complexitatea modulului. Următorul tabel indică parametrii și semnificațiile:

Parametrul	**Semnificația**
n1	Numărul de operatori unici

n2	Numărul de operanzi unici
N1	Numărul apariției totale a operatorilor
N2	Numărul apariției totale a operanzilor

Când selectăm fișierul sursă pentru a vizualiza detaliile complexității acestuia în Metric Viewer, următorul rezultat este văzut în Metric Report:

Metrică	Semnificatie	Reprezentare matematică
n	Vocabular	n1 + n2
N	Dimensiune	N1 + N2
V	Volum	Lungime * Log2 Vocabular
D	Dificultate	(n1 / 2) * (N1 / n2)
E	Efort	Dificultate * Volum
B	Erori	Volum / 3000
T	Timp de testare	Timp = Eforturi / S, unde S = 18 secunde

16. Măsuri de complexitate ciclomatică

Fiecare program cuprinde instrucțiuni de executat pentru a îndeplini anumite sarcini și alte instrucțiuni decizionale care decid, ce instrucțiuni trebuie executate. Aceste structuri decizionale schimbă fluxul programului.

Dacă comparăm două programe de aceeași dimensiune, cel cu mai multe declarații decizionale va fi mai complex pe măsură ce controlul programului sare frecvent.

McCabe, în 1976, a propus măsură de complexitate ciclomatică pentru a cuantifica complexitatea unui anumit software. Este un model bazat pe grafic care se bazează pe structuri de luare a deciziilor de program, cum ar fi afirmațiile if-else, do-while, repeat-until, switch-case și goto.

Procesul de realizare a graficului de control al fluxului:

• Pauză program în blocuri mai mici, delimitate de construcții decizionale.

• Creați noduri reprezentând fiecare dintre aceste noduri.

• Conectați nodurile după cum urmează:

- Dacă controlul se poate ramifica de la blocul i la blocul j

  Desenați un arc

- De la nodul de ieșire la nodul de intrare

  Desenați un arc.

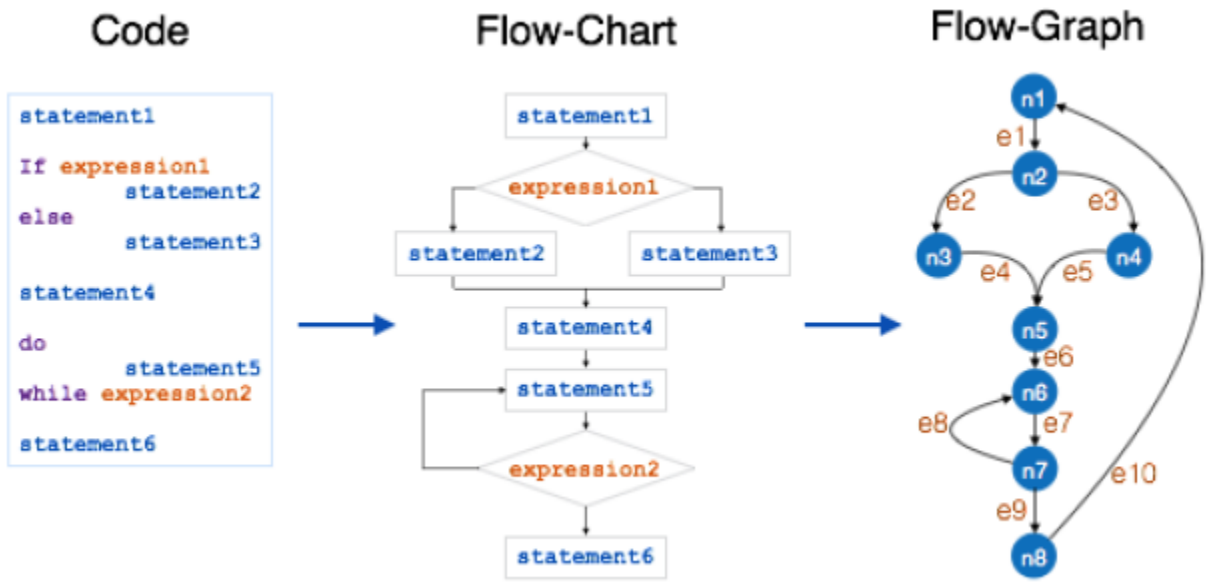
Pentru a calcula complexitatea ciclomatică a unui modul de program, folosim formula:

V (G) = e - n + 2

Unde:

e este numărul total de muchii

n este numărul total de noduri



Complexitatea ciclomatică a modulului de mai sus este

e = 10

n = 8

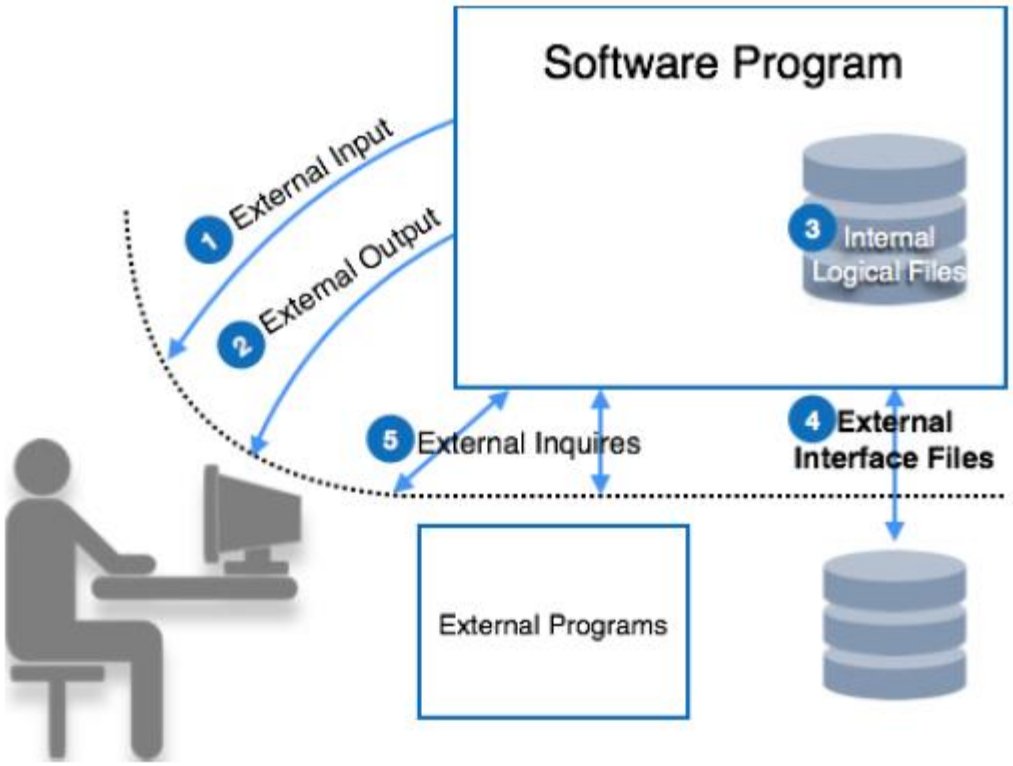
Complexitate ciclomatică = 10 - 8 + 2 = 4

Potrivit lui P. Jorgensen, complexitatea ciclomatică a unui modul nu ar trebui să depășească valoarea 10.

16.1 Punct de funcție

Este utilizat pe scară largă pentru a măsura dimensiunea software-ului. Function Point se concentrează pe funcționalitatea oferită de sistem. Caracteristicile și funcționalitatea sistemului sunt utilizate pentru a măsura complexitatea software-ului.

Punctul funcțional se bazează pe cinci parametri, numiți Intrare externă, Ieșire externă, Fișiere logice interne, Fișiere de interfață externe și Cerere externă. Pentru a lua în considerare complexitatea software-ului, fiecare parametru este clasificat în continuare ca simplu, mediu sau complex.



Să vedem parametrii punctului funcțional:

Intrare externă

Fiecare intrare unică a sistemului, din exterior, este considerată intrare externă. Unicitatea intrării este măsurată, deoarece nu există două intrări care să aibă aceleași formate.

Aceste intrări pot fi fie date, fie parametri de control.

• **Simplu** - dacă numărul de intrări este redus și afectează mai puține fișiere interne

• **Complex** - dacă numărul de intrări este mare și afectează mai multe fișiere interne

• **Mediu** - între simplu și complex.

Ieșire externă

Toate tipurile de ieșire furnizate de sistem sunt numărate în această categorie. Ieșirea este considerată unică dacă formatul de ieșire și / sau procesarea lor sunt unice.

• **Simplu** - dacă numărul de ieșiri este redus

• **Complex** - dacă numărul de ieșiri este mare

• **Medie** - între simplu și complex.

Fișiere interne logice

Fiecare sistem software menține fișiere interne pentru a-și menține informațiile funcționale și pentru a funcționa corect. Aceste fișiere conțin date logice ale sistemului.

Aceste date logice pot conține atât date funcționale, cât și date de control.

• **Simplu** - dacă numărul de tipuri de înregistrări este redus

• **Complex** - dacă numărul de tipuri de înregistrări este mare

• **Medie** - între simplu și complex.

Fișiere de interfață externe

Este posibil ca sistemul software să trebuiască să-și partajeze fișierele cu un software extern sau poate fi necesar să treacă fișierul pentru procesare sau ca parametru la o anumită funcție. Toate aceste fișiere sunt considerate fișiere de interfață externe.

• **Simplu** - dacă numărul de tipuri de înregistrări din fișierul partajat este scăzut • Complex - dacă numărul de tipuri de înregistrări din fișierul partajat este mare

• **Medie** - între simplu și complex.

Anchetă externă

O anchetă este o combinație de intrare și ieșire, în care utilizatorul trimite câteva date pentru a întreba despre intrare și sistemul răspunde utilizatorului cu ieșirea de anchetă procesată. Complexitatea unei interogări este mai mult decât intrarea externă și ieșirea externă. Se spune că interogarea este unică dacă intrarea și ieșirea acesteia sunt unice în ceea ce privește formatul și datele.

• Simplu - dacă interogarea necesită o procesare redusă și produce o cantitate mică de date de ieșire

• Complex - dacă interogarea are nevoie de un proces ridicat și produce o cantitate mare de date de ieșire

Media - între simplu și complex.

Fiecare dintre acești parametri din sistem primește pondere în funcție de clasa și complexitatea lor. Tabelul de mai jos menționează ponderarea dată fiecărui parametru:

Parametru	**Simplu**	**Mediu**	**Complex**
Intrări	3	4	6
Ieșiri	4	5	7
Cerere	3	4	6
Fisiere	7	10	15
Interfete	5	7	10

Tabelul de mai sus produce puncte de funcție brute. Aceste puncte funcționale sunt ajustate în funcție de complexitatea mediului. Sistemul este descris folosind paisprezece caracteristici diferite:

• Comunicări de date

• Procesare distribuită

• Obiective performante

• Sarcina de configurare a operației

• Rata tranzacției

• Introducere de date online,

• Eficiența utilizatorului final

• Actualizare online

• Logică de procesare complexă

• Reutilizabilitate

• Ușurința de instalare

• Ușurința operațională

• Site-uri multiple

• Dorința de a facilita schimbările

Acești factori de caracteristici sunt apoi evaluați de la 0 la 5, după cum se menționează mai jos:

• Fără influență

• Întâmplător

• Moderat

• In medie

• Semnificativ

• Esențial

Toate ratingurile sunt apoi însumate ca N. Valoarea N variază de la 0 la 70 (14 tipuri de caracteristici x 5 tipuri de rating). Se utilizează pentru a calcula factorii de ajustare a complexității (CAF), utilizând următoarele formule:

```
CAF = 0,65 + 0,01N
```

Atunci,

```
Puncte funcționale livrate (FP) = CAF x Raw FP
```

Acest FP poate fi apoi utilizat în diferite valori, cum ar fi:

• **Cost** = $ / FP

• **Calitate** = Erori / FP

• **Productivitate** = FP / persoană-lună

Implementarea software-ului

În acest capitol, vom studia metodele de programare, documentația și provocările în implementarea software-ului.

16.2 Programare structurată

În procesul de codificare, liniile de cod continuă să se înmulțească, astfel, dimensiunea software-ului crește. Treptat, devine aproape imposibil să ne amintim fluxul programului. Dacă uităm cum sunt construite software-ul și programele, fișierele, procedurile care stau la baza acestuia, devine foarte dificil să partajezi, să depanezi și să modifici programul. Soluția la aceasta este programarea structurată. Îi încurajează pe dezvoltator să utilizeze subrutine și bucle în loc să folosească salturi simple în cod, aducând astfel claritate în cod și îmbunătățind eficiența programării structurate ajută programatorul să reducă timpul de codare și să organizeze corect codul.

Programarea structurată indică modul în care programul va fi codat. Folosește trei concepte principale:

1. **Analiza de sus în jos** - Un software este întotdeauna creat pentru a efectua o muncă rațională. Această lucrare rațională este cunoscută sub numele de problemă în limbajul software. Astfel, este foarte important să înțelegem cum să rezolvăm problema. În cadrul analizei de sus în jos, problema este împărțită în bucăți mici, unde fiecare are o anumită semnificație. Fiecare problemă este rezolvată individual și pașii sunt precizați cu privire la modul de rezolvare a problemei.

2. **Programare modulară** - În timpul programării, codul este împărțit în grupuri mai mici de instrucțiuni. Aceste grupuri sunt cunoscute sub numele de module, subprograme sau subrutine. Programare modulară bazată pe înțelegerea analizei de sus în jos. Descurajează săriturile folosind instrucțiunile „goto” din program, ceea ce face adesea ca fluxul programului să nu poată fi urmărit. Salturile sunt interzise și formatul modular este încurajat în programarea structurată.

3. **Codificare structurată** - În referință cu analiza de sus în jos, codificarea structurată subdivizează modulele în alte unități de cod mai mici din ordinea executării lor. Programarea structurată folosește structura de control, care controlează fluxul programului, în timp ce codificarea structurată folosește structura de control pentru a-și organiza instrucțiunile în modele definibile.

16.3 Programare funcțională

Programarea funcțională este un stil de limbaj de programare, care folosește conceptele de funcții matematice. O funcție în matematică ar trebui să producă întotdeauna același rezultat la primirea aceluiași argument. În limbajele procedurale, fluxul programului rulează prin proceduri, adică controlul programului este transferat procedurii apelate. În timp ce fluxul de control se transferă de la o procedură la alta, programul își schimbă starea.

În programarea procedurală, este posibil ca o procedură să producă rezultate diferite atunci când este apelată cu același argument, deoarece programul în sine poate fi în stare diferită în timp ce îl apelează. Aceasta este o proprietate, precum și un dezavantaj al programării procedurale, în care succesiunea sau calendarul execuției procedurii devine importantă.

Programarea funcțională oferă mijloace de calcul ca funcții matematice, care produc rezultate indiferent de starea programului. Acest lucru face posibilă prezicerea comportamentului programului.

Programarea funcțională utilizează următoarele concepte:

Funcții de primă clasă și de înaltă ordine - Aceste funcții au capacitatea de a accepta o altă funcție ca argument sau returnează alte funcții ca rezultate.

• **Funcții pure** - Aceste funcții nu includ actualizări distructive, adică nu afectează I / O sau memorie și, dacă nu sunt utilizate, pot fi ușor eliminate, fără a împiedica restul programului.

• **Recursivitate** - Recursivitatea este o tehnică de programare în care o funcție se numește singură și repetă codul programului din ea, cu excepția cazului în care se potrivește o condiție predefinită. Recursivitatea este modalitatea de a crea bucle în programarea funcțională.

• **Evaluare strictă** - Este o metodă de evaluare a expresiei transmise unei funcții ca argument. Programarea funcțională are două tipuri de metode de evaluare, stricte (dornice) sau nestricte (leneșe). Evaluarea strictă evaluează întotdeauna expresia înainte de a invoca funcția. Evaluarea non-strictă nu evaluează expresia decât dacă este necesară.

• **λ-calculus** - Majoritatea limbajelor de programare funcționale folosesc λ-calcul ca sisteme de tip. Expresiile λ sunt executate evaluându-le pe măsură ce apar.

Common Lisp, Scala, Haskell, Erlang și F # sunt câteva exemple de limbaje funcționale de programare.

16.4 Stil de programare

Stilul de programare este un set de reguli de codare urmate de toți programatorii pentru a scrie codul. Când mai mulți programatori lucrează la același proiect software, trebuie să lucreze frecvent cu codul programului scris de un alt dezvoltator. Acest lucru devine obositor sau uneori imposibil, dacă toți dezvoltatorii nu respectă un anumit stil de programare standard pentru a codifica programul.

Un stil de programare adecvat include folosirea numelor de funcții și variabile relevante pentru sarcina intenționată, utilizarea indentării bine plasate, comentarea codului pentru confortul cititorului și prezentarea generală a codului. Acest lucru face ca codul programului să fie lizibil și ușor de înțeles de către toți, ceea ce, la rândul său, facilitează depanarea și rezolvarea erorilor. De asemenea, stilul de codare adecvat ajută la ușurarea documentării și actualizării.

Linii directoare de codificare

Practica stilului de codare variază în funcție de organizații, sisteme de operare și limbajul de codificare în sine.

Următoarele elemente de codificare pot fi definite în cadrul ghidurilor de codificare ale unei organizații:

• **Convenții de denumire** - Această secțiune definește modul de denumire a funcțiilor, variabilelor, constantelor și variabilelor globale.

• **Indentare** - Acesta este spațiul lăsat la începutul liniei, de obicei 2-8 spații albe sau o singură filă.

• **Spațiu alb** - În general, este omis la sfârșitul liniei.

• **Operatori** - Definește regulile de scriere a operatorilor matematici, de atribuire și logici. De exemplu, operatorul de atribuire '=' ar trebui să aibă spațiu înainte și după acesta, ca în „x = 2".

• **Structuri de control** - regulile de scriere if-then-else, case-switch, în timp ce și pentru instrucțiunile de flux de control numai și în mod imbricat.

• **Lungimea și înfășurarea liniei** - Definește câte caractere ar trebui să fie acolo într-o singură linie, în mare parte o linie are 80 de caractere. Înfășurarea definește modul în care ar trebui să fie înfășurată o linie, dacă este prea lungă.

• **Funcții** - Aceasta definește modul în care funcțiile ar trebui declarate și invocate, cu și fără parametri.

• **Variabile** - Aceasta menționează modul în care variabile de diferite tipuri de date sunt declarate și definite.

• **Comentarii** - Aceasta este una dintre componentele importante de codificare, deoarece comentariile incluse în cod descriu ce face codul de fapt și toate celelalte descrieri asociate. Această secțiune ajută, de asemenea, la crearea documentațiilor de ajutor pentru alți dezvoltatori.

16.5 Documentație software

Documentația software este o parte importantă a procesului software. Un document bine scris oferă un instrument excelent și un mijloc de stocare a informațiilor necesare pentru a ști despre procesul software. Documentația software oferă, de asemenea, informații despre modul de utilizare a produsului.

O documentație bine întreținută ar trebui să includă următoarele documente:

• **Documentație privind cerințele** - Această documentație funcționează ca instrument cheie pentru proiectantul de software, dezvoltatorul și echipa de testare pentru a-și îndeplini sarcinile respective. Acest document conține toate descrierile funcționale, nefuncționale și comportamentale ale software-ului intenționat.

Sursa acestui document poate fi stocată anterior date despre software, care rulează deja software la sfârșitul clientului, interviul clientului, chestionare și cercetare. În general, este stocat sub formă de foaie de calcul sau document de procesare a textului la echipa de management software de ultimă generație.

Această documentație funcționează ca bază pentru software-ul care urmează să fie dezvoltat și este utilizată în principal în fazele de verificare și validare. Majoritatea testelor sunt construite direct din documentația cerințelor.

• **Documentație de proiectare software** - Aceste documentații conțin toate informațiile necesare, necesare pentru a construi software-ul. Conține: (a) Arhitectură software la nivel înalt, (b) Detalii despre proiectarea software-ului, (c) Diagramele fluxului de date, (d) Proiectarea bazei de date

Aceste documente funcționează ca depozit pentru dezvoltatori pentru a implementa software-ul. Deși aceste documente nu oferă niciun detaliu cu privire la modul de codificare a programului, ele oferă toate informațiile necesare care sunt necesare pentru codificare și implementare.

• **Documentație tehnică** - Aceste documentații sunt întreținute de dezvoltatori și de programatorii reali. Aceste documente, în ansamblu, reprezintă informații despre cod. În timp ce scriu codul, programatorii menționează și obiectivul codului, cine l-a scris, unde va fi necesar, ce face și cum o face, ce alte resurse folosește codul etc.

Documentația tehnică sporește înțelegerea între diferiți programatori care lucrează la același cod. Îmbunătățește capacitatea de reutilizare a codului. Face depanarea ușoară și trasabilă.

Există diverse instrumente automate disponibile, iar unele sunt livrate cu limbajul de programare în sine. De exemplu java vine instrumentul JavaDoc pentru a genera documentația tehnică a codului.

• **Documentație pentru utilizator** - Această documentație este diferită de toate cele explicate mai sus. Toate documentațiile anterioare sunt menținute pentru a furniza informații despre software și procesul de dezvoltare al acestuia. Dar documentația utilizatorului explică modul în care ar trebui să funcționeze produsul software și cum ar trebui utilizat pentru a obține rezultatele dorite.

Aceste documentații pot include, proceduri de instalare a software-ului, ghiduri, ghiduri de utilizare, metode de dezinstalare și referințe speciale pentru a obține mai multe informații, cum ar fi actualizarea licenței etc.

16.6 Provocări de implementare software

Există câteva provocări cu care se confruntă echipa de dezvoltare în timpul implementării software-ului. Unele dintre ele sunt menționate mai jos:

• **Reutilizarea codului** - Interfețele de programare ale limbajelor actuale sunt foarte sofisticate și sunt dotate cu funcții imense de bibliotecă. Totuși, pentru a reduce costul produsului final, conducerea organizației preferă să reutilizeze codul, care a fost creat mai devreme pentru alte programe software. Există probleme uriașe cu care se confruntă programatorii pentru verificarea compatibilității și pentru a decide cât de mult cod să refolosească.

• **Managementul versiunilor** - De fiecare dată când un nou software este trimis clientului, dezvoltatorii trebuie să păstreze versiunea și configurația documentației. Această documentație trebuie să fie foarte precisă și disponibilă la timp.

• **Țintă-gazdă** - Programul software, care este dezvoltat în organizație, trebuie să fie conceput pentru mașini gazdă la sfârșitul clienților.

Dar uneori, este imposibil să proiectezi un software care să funcționeze pe mașinile țintă.

17. Prezentare Generală a Testării Software-ului

Testarea software-ului este evaluarea software-ului în raport cu cerințele colectate de la utilizatori și specificațiile sistemului. Testarea se efectuează la nivel de fază în ciclul de viață al dezvoltării software-ului sau la nivel de modul în codul programului. Testarea software-ului cuprinde validare și verificare.

Validare software

Validarea este procesul de examinare dacă software-ul satisface sau nu cerințele utilizatorului. Se efectuează la sfârșitul CVDS. Dacă software-ul corespunde cerințelor pentru care a fost creat, acesta este validat.

• Validarea asigură că produsul în curs de dezvoltare este conform cerințelor utilizatorului.

• Validarea răspunde la întrebarea - „Dezvoltăm produsul care încearcă tot ce are nevoie utilizatorul de la acest software?”.

• Validarea pune accentul pe cerințele utilizatorilor.

17.1 Verificare software

Verificarea este procesul de confirmare dacă software-ul îndeplinește cerințele de afaceri și este dezvoltat respectând specificațiile și metodologiile corespunzătoare.

• Verificarea asigură faptul că produsul dezvoltat este conform specificațiilor de proiectare.

• Verificarea răspunde la întrebarea - „Dezvoltăm acest produs urmând ferm toate specificațiile de proiectare?”

• Verificările se concentrează asupra proiectării și specificațiilor sistemului.

Ținta testului este -

• **Erori** - Acestea sunt greșeli reale de codare făcute de dezvoltatori. În plus, există o diferență de ieșire a software-ului și de ieșire dorită, este considerat ca o eroare.

• **Defect** - Când există eroare, apare defectul. Un defect, cunoscut și sub numele de eroare, este rezultatul unei erori care poate provoca eșecul sistemului.

• **Eșec** - se spune că eșecul este incapacitatea sistemului de a efectua sarcina dorită. Eșecul apare atunci când există erori în sistem.

17.2 Testare Manuala vs Automată

Testarea se poate face fie manual, fie utilizând un instrument automat de testare:

• **Manual** - Această testare se efectuează fără a lua ajutorul instrumentelor de testare automate. Software-ul de testare pregătește cazuri de testare pentru diferite secțiuni și niveluri ale codului, execută testele și raportează rezultatul managerului.

Testarea manuală consumă timp și resurse. Testatorul trebuie să confirme dacă sunt utilizate sau nu cazuri de testare corecte. O mare parte a testării implică testarea manuală.

• **Automat** Această testare este o procedură de testare realizată cu ajutorul instrumentelor de testare automată. Limitările cu testarea manuală pot fi depășite folosind instrumente de testare automate.

Un test trebuie să verifice dacă o pagină web poate fi deschisă în Internet Explorer. Acest lucru se poate face cu ușurință prin testarea manuală. Dar pentru a verifica dacă serverul web poate prelua încărcătura a 1 milion de utilizatori, este destul de imposibil să testați manual.

Există instrumente software și hardware care ajută testerul să efectueze testarea sarcinii, testarea stresului, testarea de regresie.

17.3 Abordări de testare

Testele pot fi efectuate pe baza a două abordări -

1. Testarea funcționalității

2. Testarea implementării

Atunci când funcționalitatea este testată fără a lua în considerare implementarea efectivă, este cunoscut sub numele de testare cutie neagră. Cealaltă parte este cunoscută sub numele de testare în cutie albă, în care nu doar funcționalitatea este testată, ci și modul în care este implementată.

Testele exhaustive sunt metoda cea mai dorită pentru o testare perfectă. Fiecare valoare posibilă din intervalul valorilor de intrare și ieșire este testată. Nu este posibil să testați fiecare valoare în scenariul real dacă gama de valori este mare.

Testarea cutiei negre

Se efectuează pentru a testa funcționalitatea programului și se numește și testare „comportamentală”. În acest caz, testerul are un set de valori de intrare și rezultatele dorite respective. La furnizarea intrării, dacă ieșirea se potrivește cu rezultatele dorite, programul este testat „ok” și problematic altfel.

În această metodă de testare, proiectantul și structura codului nu sunt cunoscute de tester, iar inginerii de testare și utilizatorii finali efectuează acest test pe software.

Tehnici de testare a cutiei negre:

• **Clasa de echivalență** - Intrarea este împărțită în clase similare. Dacă un element al unei clase trece testul, se presupune că toată clasa este trecută.

• **Valori limită** - Intrarea este împărțită în valori finale superioare și inferioare. Dacă aceste valori trec testul, se presupune că toate valorile intermediare pot trece și ele.

• **Graficarea efectului cauzei** - În ambele metode anterioare, se testează o singură valoare de intrare la un moment dat. Cauză (intrare) - Efectul (ieșirea) este o tehnică de testare în care combinațiile de valori de intrare sunt testate într-un mod sistematic.

• **Testare pereche** - Comportamentul software-ului depinde de mai mulți parametri. În testarea în perechi, parametrii multipli sunt testați în perechi pentru valorile lor diferite.

• **Testare bazată pe stare** - Sistemul modifică starea la furnizarea de intrare.

Aceste sisteme sunt testate pe baza stărilor și a intrării lor.

Testarea cutiei albe

Este realizat pentru a testa programul și implementarea acestuia, pentru a îmbunătăți eficiența sau structura codului. Este, de asemenea, cunoscut sub numele de testare „structurală”.

În această metodă de testare, proiectantul și structura codului sunt cunoscute de tester. Programatorii codului efectuează acest test pe cod.

Mai jos sunt câteva tehnici de testare a cutiei albe:

• **Testarea fluxului de control** - Scopul testării debitului de control de a stabili cazuri de testare care acoperă toate declarațiile și condițiile ramurii. Condițiile sucursalei sunt testate pentru a fi adevărate și false, astfel încât toate declarațiile să poată fi acoperite.

• **Testarea fluxului de date** - Această tehnică de testare pune accentul pe toate variabilele de date incluse în program. Testează unde variabilele au fost declarate și definite și unde au fost utilizate sau modificate.

17.4 Nivelurile de testare

Testarea în sine poate fi definită la diferite niveluri ale CVDS. Procesul de testare se desfășoară paralel cu dezvoltarea de software. Înainte de a trece la etapa următoare, o etapă este testată, validată și verificată.

Testarea separată se face doar pentru a vă asigura că software-ul nu rămâne niciun bug ascuns sau probleme. Software-ul este testat la diferite niveluri -

Testarea unitara

În timpul codificării, programatorul efectuează câteva teste pe acea unitate de program pentru a ști dacă nu are erori. Testarea se efectuează în cadrul abordării de testare a cutiei albe. Testarea unităților îi ajută pe dezvoltatori să decidă dacă unitățile individuale ale programului funcționează conform cerințelor și nu au erori.

Testarea integrării

Chiar dacă unitățile software funcționează bine individual, este necesar să aflăm dacă unitățile, dacă sunt integrate împreună, ar funcționa și fără erori. De exemplu, trecerea argumentelor și actualizarea datelor etc.

Testarea sistemului

Software-ul este compilat ca produs și apoi este testat în ansamblu. Acest lucru poate fi realizat folosind unul sau mai multe dintre următoarele teste:

• **Testarea funcționalității** - Testează toate funcționalitățile software-ului în funcție de cerință.

• **Testarea performanței** - Acest test demonstrează cât de eficient este software-ul. Acesta testează eficiența și timpul mediu al software-ului pentru îndeplinirea sarcinii dorite. Testarea performanței se face prin testarea sarcinii și testarea stresului în care software-ul este supus unei sarcini ridicate de utilizator și de date în diferite condiții de mediu.

• **Securitate și portabilitate** - Aceste teste se fac atunci când software-ul este destinat să funcționeze pe diverse platforme și accesat de numărul de persoane.

Testarea de acceptare

Când software-ul este gata să fie predat clientului, acesta trebuie să treacă prin ultima fază de testare, unde este testat pentru interacțiunea cu utilizatorul și răspunsul. Acest lucru este important, deoarece chiar dacă software-ul corespunde tuturor cerințelor utilizatorului și dacă utilizatorului nu îi place modul în care apare sau funcționează, acesta poate fi respins.

• **Testarea alfa** - Echipa de dezvoltatori efectuează testarea alfa folosind sistemul ca și cum ar fi utilizat în mediul de lucru. Ei încearcă să afle cum va reacționa utilizatorul la o acțiune din software și cum ar trebui să răspundă sistemul la intrări.

• **Testarea beta** - După ce software-ul este testat intern, acesta este predat utilizatorilor să-l folosească în mediul lor de producție numai în scopul testării. Acesta nu este încă produsul livrat. Dezvoltatorii se așteaptă ca utilizatorii din această etapă să aducă probleme minuscule, care au fost omise pentru a participa.

Testarea regresiei

Ori de câte ori un produs software este actualizat cu un nou cod, caracteristică sau funcționalitate, acesta este testat cu atenție pentru a detecta dacă există vreun impact negativ al codului adăugat.

Acest lucru este cunoscut sub numele de testare de regresie.

17.5 Documentație de testare

Documentele de testare sunt pregătite în diferite etape -

Înainte de testare

Testarea începe cu generarea cazurilor de testare. Următoarele documente sunt necesare pentru referință -

• **Document SRS** - Document de cerințe funcționale

• **Document privind politica de testare** - Acesta descrie cât de mult ar trebui să aibă loc testarea înainte de lansarea produsului.

• **Document de strategie de testare** - Acesta menționează aspectele detaliate ale echipei de testare, matricea de responsabilitate și drepturile / responsabilitatea managerului de testare și a inginerului de testare.

• **Document de matrice de trasabilitate** - Acesta este documentul CVDS, care este legat de procesul de colectare a cerințelor. Pe măsură ce vin noi cerințe, acestea se adaugă la această matrice. Aceste matrice îi ajută pe testatori să cunoască sursa cerințelor. Ele pot fi urmărite înainte și înapoi.

În timp ce fii testat

Următoarele documente pot fi necesare în timpul începerii și în curs de testare:

• **Document ptr caz de testare** - Acest document conține lista testelor necesare pentru a fi efectuate. Acesta include planul de testare unitară, planul de testare a integrării, planul de testare a sistemului și planul de testare a acceptării.

• **Descrierea testului** - Acest document este o descriere detaliată a tuturor cazurilor de testare și a procedurilor de executare a acestora.

• **Raport de caz de testare** - Acest document conține raport de caz de testare ca rezultat al testului.

• **Jurnalele de testare** - Acest document conține jurnalele de testare pentru fiecare raport de caz de testare.

După testare

Următoarele documente pot fi generate după testare:

• **Sumarul testului** - Acest rezumat al testului este o analiză colectivă a tuturor rapoartelor și jurnalelor de testare. Rezumă și concluzionează dacă software-ul este gata de lansare. Software-ul este lansat sub sistemul de control al versiunilor dacă este gata de lansare.

17.6 Testare vs. Control de calitate și asigurare și audit

Trebuie să înțelegem că testarea software-ului este diferită de asigurarea calității software-ului, controlul calității software-ului și auditul software-ului.

• **Asigurarea calității software** - Acestea sunt mijloace de monitorizare a procesului de dezvoltare a software-ului, prin care se asigură că toate măsurile sunt luate conform standardelor de organizare. Această monitorizare se face pentru a vă asigura că au fost urmate metode adecvate de dezvoltare software.

• **Controlul calității software** - Acesta este un sistem pentru menținerea calității produsului software. Poate include aspecte funcționale și nefuncționale ale produsului software, care sporesc bunăvoința organizației. Acest sistem asigură faptul că clientul primește produse de calitate pentru cerințele sale și că produsul este certificat ca „potrivit pentru utilizare”.

• **Auditul software** - Aceasta este o revizuire a procedurii utilizate de organizație pentru a dezvolta software-ul. O echipă de auditori, independentă de echipa de dezvoltare examinează procesul software, procedura, cerințele și alte aspecte ale CVDS. Scopul auditului software-ului este de a verifica dacă software-ul și procesul său de dezvoltare, respectă standardele, regulile și reglementările.

18. Prezentare generală a întreținerii software

Întreținerea software-ului este o parte acceptată pe scară largă a CVDS acum câteva zile. Reprezintă toate modificările și actualizările efectuate după livrarea produsului software. Există mai multe motive pentru care sunt necesare modificări, unele dintre ele fiind menționate pe scurt mai jos:

• **Condiții de piață** - Politicile, care se modifică în timp, cum ar fi impozitarea și constrângerile nou introduse, cum ar fi, modul de menținere a contabilității, pot declanșa necesitatea modificării.

• **Cerințe client** - De-a lungul timpului, clientul poate solicita noi funcții sau funcții în software.

• **Modificări gazdă** - Dacă se modifică orice hardware și / sau platformă (cum ar fi sistemul de operare) al gazdei țintă, sunt necesare modificări software pentru a menține adaptabilitatea.

• **Modificări ale organizației** - Dacă există o schimbare la nivelul afacerii la sfârșitul clientului, cum ar fi reducerea puterii organizației, achiziționarea unei alte companii, organizație care se aventurează într-o afacere nouă, poate apărea necesitatea modificării în software-ul original.

18.1 Tipuri de întreținere

În timpul duratei de viață a software-ului, tipul de întreținere poate varia în funcție de natura sa. Poate fi doar o sarcină de întreținere de rutină, deoarece o eroare descoperită de un utilizator sau poate fi un eveniment mare în sine, bazat pe dimensiunea sau natura întreținerii. Următoarele sunt câteva tipuri de întreținere pe baza caracteristicilor lor:

• **Întreținere corectivă** - Aceasta include modificări și actualizări efectuate pentru a corecta sau remedia problemele, care sunt fie descoperite de utilizator, fie încheiate prin rapoarte de erori ale utilizatorului.

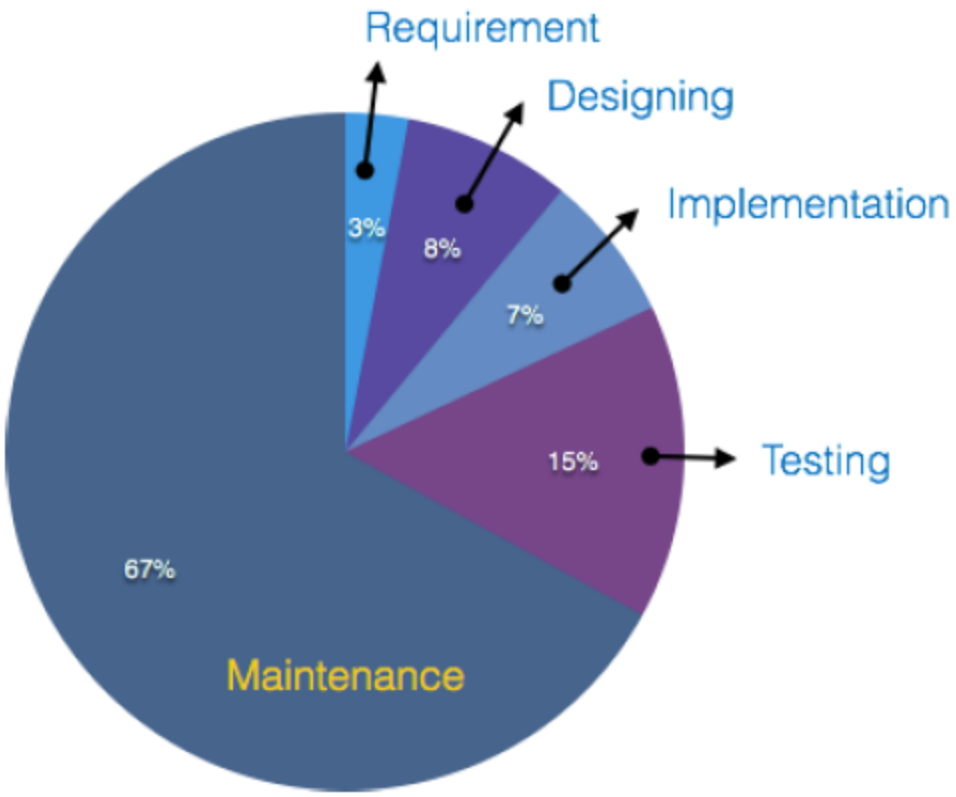
• **Întreținere adaptivă** - Aceasta include modificări și actualizări aplicate pentru a menține produsul software actualizat și adaptat la lumea în continuă schimbare a tehnologiei și a mediului de afaceri.

• **Întreținere perfectivă** - Aceasta include modificări și actualizări efectuate pentru a menține software-ul utilizabil pe o perioadă lungă de timp. Acesta include noi caracteristici, noi cerințe ale utilizatorilor pentru rafinarea software-ului și îmbunătățirea fiabilității și performanței acestuia.

• **Întreținere preventivă** - Aceasta include modificări și actualizări pentru a preveni problemele viitoare ale software-ului. Acesta își propune să participe la probleme, care nu sunt semnificative în acest moment, dar care pot provoca probleme grave în viitor.

18.2 Costul întreținerii

Rapoartele sugerează că costul întreținerii este ridicat. Un studiu privind estimarea întreținerii software a constatat că costul întreținerii este de până la 67% din costul întregului ciclu de proces software.



În medie, costul întreținerii software-ului este mai mare de 50% din toate fazele CVDS. Există diferiți factori care declanșează costurile de întreținere crescute, cum ar fi:

Factori din lumea reală care afectează costurile de întreținere

• Vârsta standard a oricărui software este considerată de până la 10 până la 15 ani.

• Software-urile mai vechi, care erau menite să funcționeze pe mașini lente, cu memorie și capacitate de stocare mai reduse, nu se pot menține provocatoare împotriva software-urilor îmbunătățite nou venite pe hardware-ul modern.

• Pe măsură ce tehnologia avansează, devine costisitoare menținerea unui software vechi.

• Majoritatea inginerilor de întreținere sunt începători și folosesc metoda de încercare și eroare pentru a remedia problema.

• Adesea, modificările făcute pot afecta cu ușurință structura originală a software-ului, făcându-l dificil pentru orice modificări ulterioare.

• Modificările sunt adesea lăsate nedocumentate, ceea ce poate provoca mai multe conflicte în viitor.

Factorii de finalizare a software-ului care afectează costul de întreținere

• Structura programului software

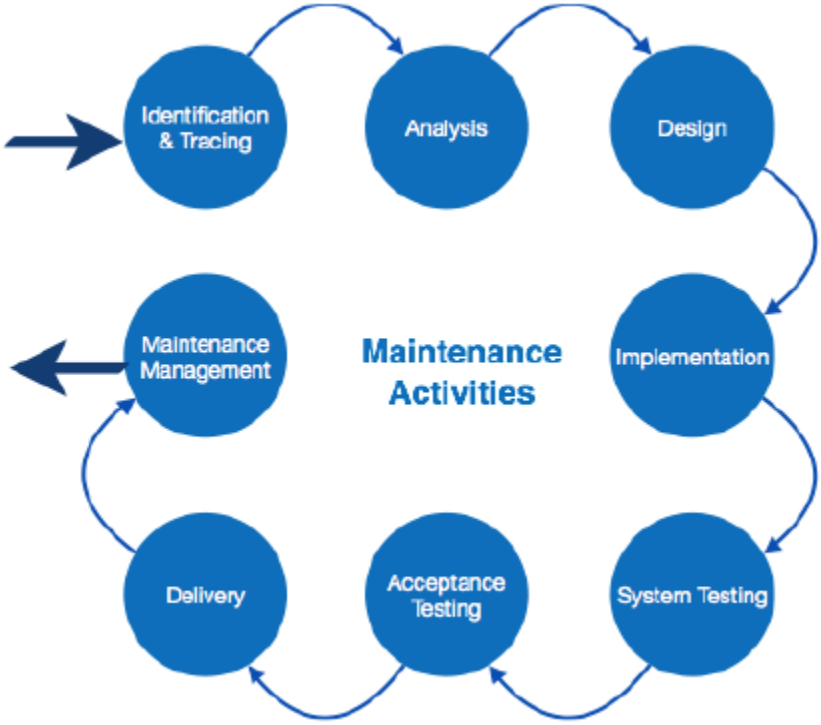
• Limbaj de programare

• Dependența de mediul extern

• Fiabilitatea și disponibilitatea personalului

18.3 Activități de întreținere

IEEE oferă un cadru pentru activitățile procesului de întreținere secvențială. Poate fi folosit în mod iterativ și poate fi extins astfel încât să poată fi incluse articole și procese personalizate.



Aceste activități merg mână în mână cu fiecare dintre următoarele faze:

• **Identificare și urmărire** - implică activități legate de identificarea cerinței de modificare sau întreținere. Acesta este generat de utilizator sau sistemul poate raporta el însuși prin jurnale sau mesaje de eroare. Aici, tipul de întreținere este, de asemenea, clasificat.

• **Analiză** - Modificarea este analizată pentru impactul său asupra sistemului, inclusiv implicațiile de siguranță și securitate. Dacă impactul probabil este sever, se caută o soluție alternativă. Un set de modificări necesare este apoi materializat în specificațiile cerințelor. Costul modificării / întreținerii este analizat și estimarea este încheiată.

• **Proiectare** - Modulele noi, care trebuie înlocuite sau modificate, sunt proiectate conform specificațiilor cerințelor stabilite în etapa anterioară. Testele sunt create pentru validare și verificare.

• **Implementare** - Noile module sunt codificate cu ajutorul proiectării structurate create în etapa de proiectare. Fiecare programator este de așteptat să facă teste unitare în paralel.

• **Testarea sistemului** - Testarea integrării se face printre modulele create recent. Testarea integrării este, de asemenea, efectuată între noi module și sistem. În cele din urmă, sistemul este testat ca întreg, urmând proceduri de testare regresive.

• **Testarea acceptării** - După testarea internă a sistemului, acesta este testat pentru acceptare cu ajutorul utilizatorilor. Dacă se află în această stare, utilizatorii reclamă anumite probleme cărora li se adresează sau se notează că vor fi abordate în următoarea iterație.

• **Livrare** - După testul de acceptare, sistemul este implementat în întreaga organizație, fie printr-un pachet mic de actualizare, fie prin instalarea proaspătă a sistemului. Testarea finală are loc la sfârșitul clientului după livrarea software-ului.

Instalația de instruire este oferită, dacă este necesar, în plus față de copia tipărită a manualului de utilizare.

• **Managementul întreținerii** - Managementul configurației este o parte esențială a întreținerii sistemului. Este ajutat cu instrumente de control al versiunilor pentru a controla versiunile, semi-versiunea sau gestionarea patch-urilor.

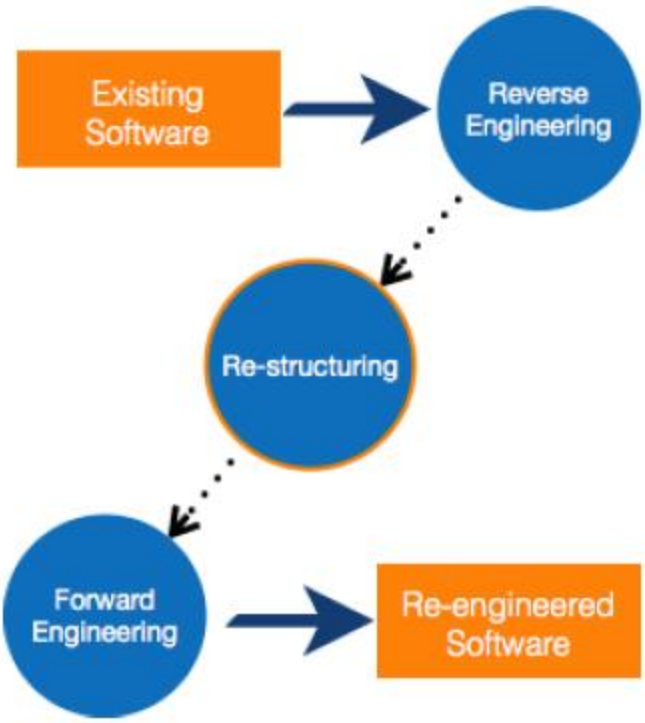
18.4 Re-inginerie software

Când trebuie să actualizăm software-ul pentru a-l menține pe piața actuală, fără a afecta funcționalitatea acestuia, acesta se numește re-inginerie software. Este un proces minuțios în care designul software-ului este modificat și programele sunt rescrise.

Software-ul vechi nu poate continua să se adapteze la cea mai recentă tehnologie disponibilă pe piață. Pe măsură ce hardware-ul devine depășit, actualizarea software-ului devine o durere de cap. Chiar dacă software-ul îmbătrânește cu timpul, funcționalitatea sa nu.

De exemplu, inițial Unix a fost dezvoltat în limbaj de asamblare. Când a apărut limbajul C, Unix a fost reproiectat în C, deoarece lucrul în limbajul asamblării era dificil.

În afară de aceasta, uneori programatorii observă că puține părți ale software-ului necesită mai multă întreținere decât altele și, de asemenea, au nevoie de re-inginerie.



Proces de re-inginerie

• **Decideți** ce să reproiectați. Este un software întreg sau o parte din el?

• **Efectuați** Reverse Engineering, pentru a obține specificații ale software-ului existent.

• **Programul de restructurare**, dacă este necesar. De exemplu, schimbarea programelor orientate funcțional în programe orientate obiect.

• **Re-structurați datele** după cum este necesar.

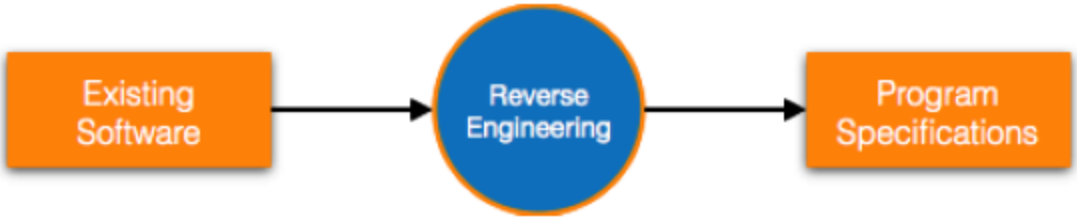
• **Aplicați concepte de inginerie forward** pentru a obține software-ul reproiectat.

Există puțini termeni importanți utilizați în re-proiectarea software-ului

Inginerie inversă

Este un proces de realizare a specificațiilor sistemului, analizând, înțelegând cu atenție sistemul existent. Acest proces poate fi văzut ca un model CVDS invers, adică încercăm să obținem un nivel mai ridicat de abstractizare analizând niveluri mai mici de abstractizare.

Un sistem existent este implementat anterior în proiectare, despre care nu știm nimic. Proiectanții fac apoi inginerie inversă uitându-se la cod și încearcă să obțină designul. Cu designul în mână, încearcă să încheie specificațiile. Astfel, mergând invers de la cod la specificațiile sistemului.



Restructurarea programului

Este un proces de restructurare și reconstruire a software-ului existent. Este vorba despre rearanjarea codului sursă, fie în același limbaj de programare, fie de la un limbaj de programare la altul. Restructurarea poate avea fie restructurarea codului sursă, cât și restructurarea datelor sau ambele.

Re-structurarea nu afectează funcționalitatea software-ului, ci sporește fiabilitatea și mentenabilitatea. Componentele programului, care cauzează erori foarte frecvent, pot fi modificate sau actualizate cu restructurarea.

Fiabilitatea software-ului pe platforma hardware învechită poate fi eliminată prin restructurare.

Inginerie Forward

Ingineria directă este un proces de obținere a software-ului dorit din specificațiile avute, care au fost reduse prin intermediul ingineriei inverse. Se presupune că a existat o anumită inginerie software deja realizată în trecut.

Ingineria directă este aceeași cu procesul de inginerie software cu o singură diferență - se realizează întotdeauna după inginerie inversă.

Reutilizarea componentelor

O componentă este o parte a codului programului software, care execută o activitate independentă în sistem. Poate fi un modul mic sau un subsistem în sine.

Exemplu

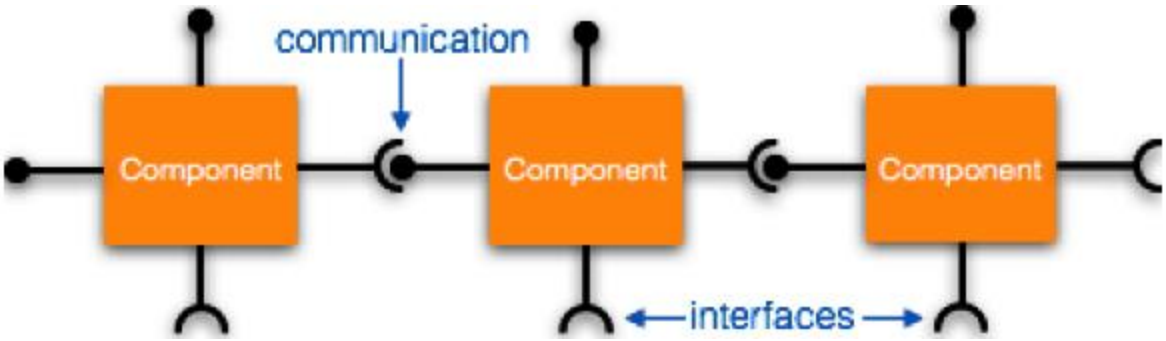
Procedurile de conectare utilizate pe web pot fi considerate componente, sistemul de imprimare în software poate fi văzut ca o componentă a software-ului.

Componentele au o coeziune ridicată a funcționalității și o rată mai mică de cuplare, adică funcționează independent și pot îndeplini sarcini fără a depinde de alte module.

În OOP, obiectele sunt proiectate sunt foarte specifice preocupărilor lor și au șanse mai mici de a fi utilizate în alte programe.

În programarea modulară, modulele sunt codificate pentru a îndeplini sarcini specifice care pot fi utilizate în numeroase alte programe software.

Există o nouă verticală, care se bazează pe reutilizarea componentelor software și este cunoscută sub numele de Inginerie software bazată pe componente (CBSE).



Reutilizarea se poate face la diferite niveluri

• **Nivelul aplicației** - În cazul în care o aplicație întreagă este utilizată ca subsistem de software nou.

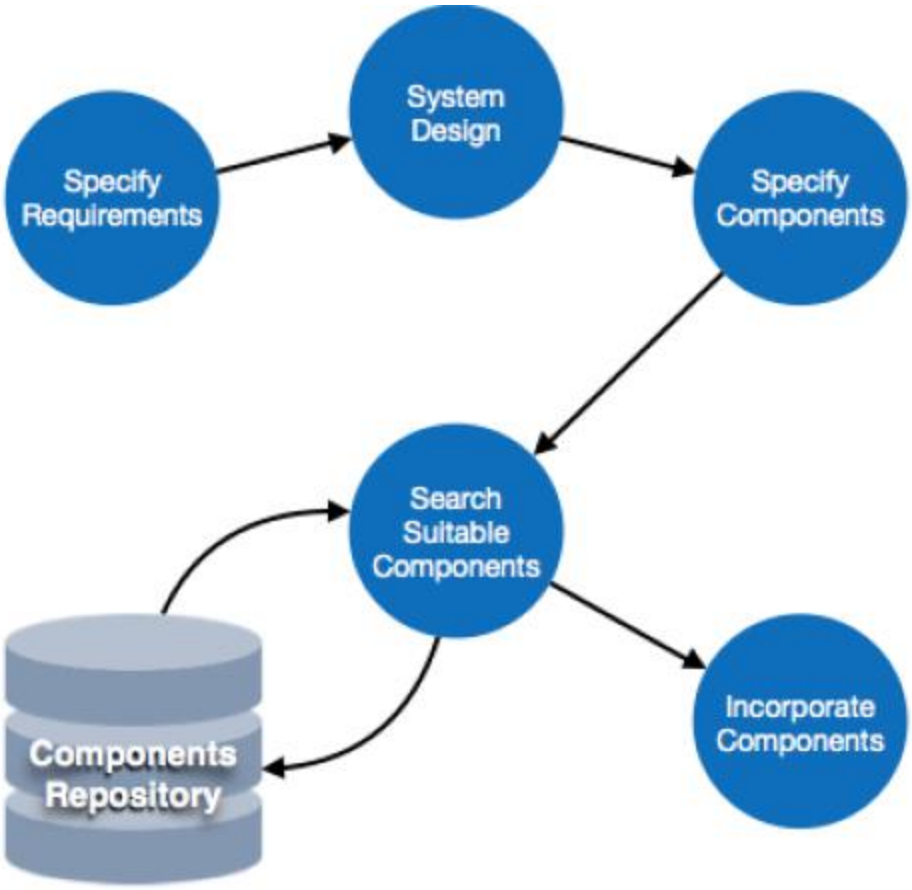
• **Nivel componen**ta - Unde este utilizat subsistemul unei aplicații.

• **Nivelul modulelor** - În cazul în care modulele funcționale sunt refolosite.

Componentele software oferă interfețe, care pot fi utilizate pentru a stabili comunicarea între diferite componente.

Procesul de reutilizare

Două tipuri de metode care pot fi adoptate: fie prin menținerea acelorași cerințe și ajustarea componentelor, fie prin menținerea componentelor la fel și modificarea cerințelor.



• **Specificații cerințe** - Sunt specificate cerințele funcționale și nefuncționale, pe care trebuie să le respecte un produs software, cu ajutorul sistemului existent, a intrării utilizatorului sau a ambelor.

• **Proiectare** - Acesta este, de asemenea, un pas standard al procesului CVDS, unde cerințele sunt definite în termeni de limbaj software. Se creează arhitectura de bază a sistemului ca întreg și subsistemele acestuia.

• **Specificați componente** - Prin studierea proiectării software-ului, proiectanții separă întregul sistem în componente sau subsisteme mai mici. Un design complet al software-ului se transformă într-o colecție de un set imens de componente care lucrează împreună.

• **Căutare componente adecvate** - Depozitul de componente software este trimis de către proiectanți pentru a căuta componenta potrivită, pe baza funcționalității și a cerințelor software intenționate ..

• **Incorporați componente** - Toate componentele potrivite sunt ambalate împreună pentru a le forma ca software complet.

19. Prezentare generală a Software CASE Tools

CASE înseamnă Computer Aided Software Engineering. Aceasta reprezinta dezvoltarea și întreținerea de proiecte software cu ajutorul diferitelor instrumente software automatizate.

19.1 Instrumente CASE

Instrumentele CASE sunt un set de programe de aplicații software, care sunt utilizate pentru automatizarea activităților CVDS. Instrumentele CASE sunt utilizate de managerii de proiect software, analiști și ingineri pentru a dezvolta un sistem software.

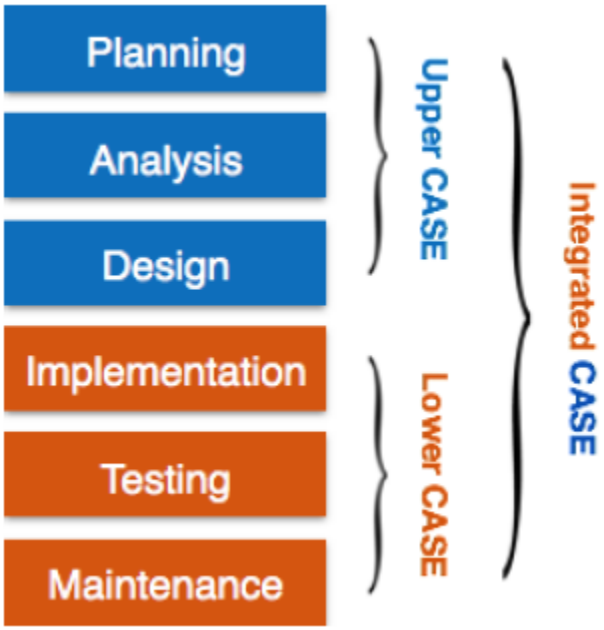
Există mai multe instrumente CASE disponibile pentru simplificarea diferitelor etape ale ciclului de viață al dezvoltării software, cum ar fi instrumentele de analiză, instrumentele de proiectare, instrumentele de gestionare a proiectelor, instrumentele de gestionare a bazelor de date, instrumentele de documentare sunt câteva dintre acestea.

Utilizarea instrumentelor CASE accelerează dezvoltarea proiectului pentru a produce rezultatul dorit și ajută la descoperirea defectelor înainte de a trece la următoarea etapă a dezvoltării software-ului.

19.2 Componentele instrumentelor CASE

Instrumentele CASE pot fi împărțite în general în următoarele părți, pe baza utilizării lor într-un anumit stadiu CVDS:

• **Depozit central** - instrumentele CASE necesită un depozit central, care poate servi ca sursă de informații comune, integrate și consistente. Depozitul central este un loc central de stocare în care sunt stocate specificațiile produsului, documentele de cerințe, rapoartele și diagramele aferente, alte informații utile privind gestionarea. Depozitul central servește și ca dicționar de date.



• **Instrumente Upper CASE** - Instrumentele cu majuscule CASE sunt utilizate în etapele de planificare, analiză și proiectare ale CVDS.

• **Instrumente Lower CASE** - Uneltele minuscule CASE sunt utilizate în implementare, testare și întreținere.

• **Integreted Case Tools** - Instrumentele CASE integrate sunt utile în toate etapele CVDS, de la colectarea cerințelor până la testare și documentare.

Instrumentele CASE pot fi grupate împreună dacă au funcționalități similare, activități de proces și capacitatea de a se integra cu alte instrumente.

19.3 Domeniul de aplicare al instrumentelor de caz

Domeniul de aplicare al instrumentelor CASE acoperă CVDS. Acum trecem pe scurt prin diferite instrumente CASE

Instrumente pentru diagrame

Aceste instrumente sunt utilizate pentru a reprezenta componentele sistemului, datele și fluxul de control între diferite componente software și structura sistemului într-o formă grafică.

De exemplu, instrumentul Flow Chart Maker pentru crearea diagramelor de flux de ultimă generație.

Instrumente de modelare a proceselor

Modelarea proceselor este o metodă de creare a unui model de proces software, care este utilizat pentru dezvoltarea software-ului. Instrumentele de modelare a proceselor îi ajută pe manageri să aleagă un model de proces sau să îl modifice conform cerințelor produsului software. De exemplu, EPF Composer

Instrumente de gestionare a proiectelor

Aceste instrumente sunt utilizate pentru planificarea proiectelor, estimarea costurilor și eforturilor, planificarea proiectelor și planificarea resurselor. Managerii trebuie să respecte cu strictețe execuția proiectului cu fiecare pas menționat în gestionarea proiectelor software. Instrumentele de gestionare a proiectelor ajută la stocarea și partajarea informațiilor despre proiect în timp real în întreaga organizație. De exemplu, Creative Pro Office, Trac Project, Basecamp.

Instrumente de documentare

Documentarea într-un proiect software începe înainte de procesul software, trece în toate fazele CVDS și după finalizarea proiectului.

Instrumentele de documentare generează documente pentru utilizatorii tehnici și utilizatorii finali. Utilizatorii tehnici sunt în mare parte profesioniști interni ai echipei de dezvoltare care se referă la manualul de sistem, manualul de referință, manualul de instruire, manualele de instalare etc. De exemplu, Doxygen, DrExplain, Adobe RoboHelp pentru documentare.

Instrumente de analiză

Aceste instrumente ajută la colectarea cerințelor, verifică automat dacă există inconsecvențe, inexactități în diagrame, redundanțe de date sau omisiuni eronate. De exemplu, Accept 360, Accompa, CaseComplete pentru analiza cerințelor, Analist vizibil pentru analiza totală.

Instrumente de proiectare

Aceste instrumente îi ajută pe proiectanții de software să proiecteze structura bloc a software-ului, care poate fi descompusă în continuare în module mai mici folosind tehnici de rafinare. Aceste instrumente oferă detalii despre fiecare modul și interconectări între module. De exemplu, Proiectare software animat.

Instrumente de gestionare a configurației

O instanță de software este lansată sub o singură versiune. Instrumentele de gestionare a configurației se ocupă de -

• Managementul versiunilor și reviziilor

• Managementul configurației de bază

• Managementul controlului schimbărilor

Instrumentele CASE ajută în acest sens prin urmărirea automată, gestionarea versiunilor și gestionarea versiunilor. De exemplu, Fossil, Git, Accu REV.

Instrumente de control al modificărilor

Aceste instrumente sunt considerate ca parte a instrumentelor de gestionare a configurației. Acestea se ocupă de modificările aduse software-ului după ce linia de bază a fost reparată sau când software-ul este lansat pentru prima dată. Instrumentele CASE automatizează urmărirea modificărilor, gestionarea fișierelor, gestionarea codului și multe altele. De asemenea, ajută la aplicarea politicii de schimbare a organizației.

Instrumente de programare

Aceste instrumente constau din medii de programare precum IDE (Integrated Development Environment), bibliotecă de module încorporate și instrumente de simulare. Aceste instrumente oferă ajutor cuprinzător în construirea produselor software și includ caracteristici pentru simulare și testare. De exemplu, Cscope pentru a căuta codul în C, Eclipse.

Instrumente de prototipare

Prototipul software este versiunea simulată a produsului software destinat. Prototipul oferă aspectul inițial al produsului și simulează câteva aspecte ale produsului real.

Protejarea instrumentelor CASE vine în esență cu biblioteci grafice. Ele pot crea interfețe de utilizator independente și design. Aceste instrumente ne ajută să construim prototipuri rapide pe baza informațiilor existente. În plus, acestea oferă simularea prototipului software. De exemplu, Serena prototype composer – creatorul de prototipuri Serena, Mockup Builder – constructor de macheta.

Instrumente de dezvoltare web

Aceste instrumente vă ajută să proiectați pagini web cu toate elementele conexe, cum ar fi formulare, text, script, grafic și așa mai departe. Instrumentele web oferă, de asemenea, o previzualizare live a ceea ce se dezvoltă și cum va arăta după finalizare. De exemplu, Fontello, Adobe Edge Inspect, Foundation 3, Brackets.

Instrumente de asigurare a calității

Asigurarea calității într-o organizație software monitorizează procesul de inginerie și metodele adoptate pentru a dezvolta produsul software pentru a asigura conformitatea calității conform standardelor organizației. Instrumentele QA constau în instrumente de configurare și control al modificărilor și instrumente de testare software. De exemplu, SoapTest, AppsWatch, JMeter.

Instrumente de întreținere

Întreținerea software-ului include modificări ale produsului software după livrare. Tehnici de înregistrare automată și raportare a erorilor, generarea automata de tickete de erori și analiza cauzei la radacina sunt câteva instrumente CASE, care ajută organizarea software-ului în faza de întreținere a SDLC. De exemplu, Bugzilla pentru urmărirea defectelor, HP Quality Center.