Paradigmele de dezvoltare a sistemelor de programe

3.1. Etapele dezvoltării programelor

Când pornim la dezvoltarea unui program avem nevoie de:

- înţelegere clară a ceea ce se cere;
- un set de metode şi instrumente de lucru;
- un plan de acţiune.

Planul de acţiune se numeşte ***metodologie de dezvoltare***.

- Dezvoltarea unui anumit program constă într-un set de paşi ce se fac pentru a-l realiza.
- Luând în considerare tipul paşilor ce se efectuează, se creează un model de lucru, ce poate fi aplicat unei serii mai largi de proiecte.
- Acesta este motivul pentru care planul de acţiune este numit **model**: el poate fi privit ca un şablon al dezvoltării de programe.
- În timpul dezvoltării programelor s-a constatat că există anumite tipuri de activităţi care trebuie făcute la un moment dat:

- ***Analiza cerinţelor**:* Se stabileşte ce anume vrea clientul ca programul să facă.
  - înregistrarea cerinţelor într-o manieră cât mai clară (lipsa ambiguităţilor) şi mai fidelă (cuvânt cu cuvânt);
- ***Proiectarea arhitecturală**:* - împarte sistemul într-un număr de module mai mici şi mai simple, care pot fi abordate individual;
- ***Proiectarea detaliată**:* Se realizează proiectarea fiecărui modul al aplicaţiei, în cele mai mici detalii;
- ***Scrierea codului**:* Proiectul detaliat este transpus într-un limbaj de programare. De obicei, aceasta se realizează modular, pe structura rezultată la proiectarea arhitecturală;
- ***Integrarea componentelor**:* Modulele programului sunt combinate în produsul final. Rezultatul este sistemul complet.
  - În modelul numit ***big-bang*** componentele sunt dezvoltate şi testate individual, după care sunt integrate în sistemul final.

- Chiar dacă componentele aufost testate individual şi funcţionează, la asamblare apar erori sau conflicte între anumite componente (de exemplu, conflicte de partajare a resurselor).
- Astfel timpul de testare explodează, proiectul devenind greu de controlat; aceasta justifică denumirea de „big-bang”.

▪ ***Modelul incremental*** propune crearea unui nucleu al aplicaţiei şi integrarea a câte o componentă la un moment dat, urmată imediat de testarea sistemului obţinut. Astfel, se poate determina mai uşor unde anume apare o problema în sistem. Acest tip de integrare oferă de obicei rezultate mai bune decât modelul big-bang;

- ***Acceptarea****:* În procesul de acceptare ne asigurăm că programul îndeplineşte cerinţele utilizatorului. Întrebarea la care răspundem este: construim produsul corect?
  - ▪ Clientul spune dacă este mulţumit cu produsul sau dacă mai trebuie efectuate modificări;
- ***Verificarea****:* ne asigurăm că programul este stabil şi că funcţionează corect din punctul de vedere al dezvoltatorilor. Întrebarea la care răspundem este: construim corect produsul?
- ***Întreţinerea****:* gestionarea : erorilor, schimbarea specificaţiilor, îmbunătăţiri.

Se poate constata uşor că aceste activităţi sunt în strânsă legătură cu cele patru faze ale ingineriei programării: analiza, proiectarea, implementarea şi testarea.

3.2. Paradigmele de dezvoltare software

Paradigmele de dezvoltare a sistemelor de programe se constituie din două categorii de metodologii şi anume:

3.2.1. Metodologii generice	**3.2.2. Metodologii concrete.**
▪ Metodologia secvenţială ▪ Metodologia ciclică ▪ Metodologia hibridă ecluză	▪ Metodologia cascadă ▪ Metodologia spirală ▪ Metodologia spirală WinWin ▪ Prototipizarea ▪ Metodologia Booch ▪ Metode formale ▪ Metoda V ▪ Programarea extremă ▪ Metodologia Open Source ▪ Reverse Engineering ▪ Metodologia de dezvoltare Offshore ▪ Metodologii orientate obiect

3.2.1. Metodologii generice

3.2.1.1. Metodologia secvenţială

În metodologia secvenţială (fig.3.1), cunoscută şi sub numele de metodologia „cascadă”, are loc mai întâi faza de analiză, apoi cea de proiectare, urmată de cea de implementare, iar în final se realizează testarea. Echipele care se ocupă de fiecare fază pot fi diferite, iar la fiecare tranziţie de fază poate fi necesară o decizie managerială.

Fig. 3.1. Metodologia secvenţială

Avantaje

- este potrivită când complexitatea sistemului este mică iar cerinţele sunt statice.

- mai întâi trebuie să ne gândim ce trebuie construit, apoi să stabilim un plan de lucru şi apoi să realizăm proiectul, ţinând cont de standardele de calitate.

- se aliniază metodelor de inginerie hardware. Forţează menţinerea unei discipline de lucru care evită presiunea scrierii codului înainte de a cunoaşte precis ce produs va trebui de fapt construit.

De multe ori, echipa de implementare se află în situaţia de a programa înainte de finalizarea analizei, ceea ce conduce inevitabil la descoperirea unor părţi de cod inutile sau care contribuie foarte puţin (poate chiar şi ineficient) la funcţionalitatea produsului final. Totuşi, acest cod devine un balast foarte costisitor: dificil de abandonat şi greu de schimbat. Această metodologie forţează analiza şi planificarea înaintea implementării, o practică foarte nimerită în multe situaţii.

Un mare număr de sisteme software din trecut au fost construite cu o metodologie secvenţială. Multe companii îşi datorează succesul acestui mod de realizare a programelor. Trebuie spus totuşi şi că presiunea de schimbare din partea surselor externe era destul de limitată la momentul respectiv.

Dezavantaje

- acordă o foarte mare importanţă fazei de analiză.

- Membrii echipei de analiză ar trebui să fie probabil clarvăzători ca să poată defini *toate* detaliile aplicaţiei încă de la început.

- Greşelile nu sunt permise, deoarece nu există un proces de corectare a erorilor după lansarea cerinţelor finale.

- Nu există nici feedback de la echipa de implementare în ceea ce priveşte complexitatea codului corespunzător unei anumite cerinţe.

- O cerinţă simplu de formulat poate creşte considerabil complexitatea implementării.

- În unele cazuri, este posibil să fie chiar imposibil de implementat cu tehnologia actuală.

- Dacă echipa de analiză ar şti că o cerinţă nu poate fi implementată, ei ar putea-o schimba cu o cerinţă diferită care să satisfacă cele mai multe dintre necesităţi şi care să fie mai uşor de efectuat.

Comunicarea dintre echipe este o problemă: cele patru echipe pot fi diferite iar comunicarea dintre ele este limitată.

-- Modul principal de comunicare sunt documentele realizate de o echipă şi trimise următoarei echipe cu foarte puţin feedback.

- Echipa de analiză nu poate avea toate informaţiile privitoare la calitate, performanţă şi motivare.

Într-o industrie în continuă mişcare, metodologia secvenţială poate produce sisteme care, la vremea lansării, să fie deja învechite.

Accentul atât de mare pus pe planificare nu poate determina răspunsuri suficient de rapide la schimbare.

Ce se întâmplă dacă clientul îşi schimbă cerinţele după terminarea fazei de analiză?

Acest lucru se întâmplă însă frecvent; după ce clientul vede prototipul produsului, el îşi poate schimba unele cerinţe.

3.2.1.2. Metodologia ciclică

Metodologia ciclică (fig.3.2), cunoscută şi sub numele de metodologia „***spirală***”, încearcă să rezolve unele din problemele metodologiei secvenţiale.

Şi această metodologie are patru faze, însă în fiecare fază se consumă un timp mai scurt, după care urmează mai multe iteraţii prin toate fazele.

Ideea este de fapt următoarea: gândeşte un pic, planifică un pic, implementează un pic, testează un pic şi apoi ia-o de la capăt.

În mod ideal, fiecărei faze trebuie să i se acorde atenţie şi importanţă egale.

Documentele de la fiecare fază îşi schimbă treptat structura şi conţinutul, la fiecare ciclu sau iteraţie.

Pe măsură ce procesul înaintează, sunt generate din ce în ce mai multe detalii.

În final, după câteva cicluri, sistemul este complet şi gata de lansare.

Procesul poate însă continua pentru lansarea mai multor versiuni ale produsului.

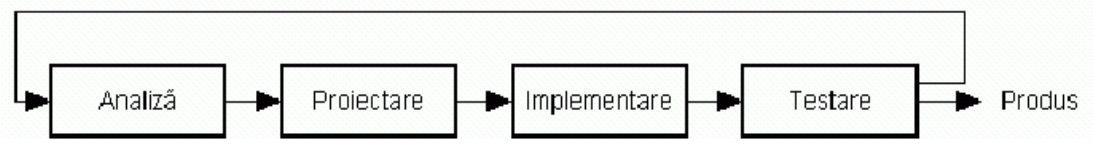

Fig.3.2. Metodologia ciclică

Avantaje

Metodologia ciclică se bazează pe ideea perfecţionării incrementale ale metodologiei secvenţiale.

Permite feedback-ul de la fiecare echipă în ceea ce priveşte complexitatea cerinţelor.

Există etape în care pot fi corectate eventualele greşeli privind cerinţele.

Clientul poate arunca o privire asupra rezultatului şi poate oferi informaţii importante mai ales în faza dinaintea lansării produsului.

Echipa de implementare poate trimite echipei de analiză informaţii privind performanţele şi viabilitatea sistemului.

Acesta se poate adapta mai bine progresului tehnologic: pe măsură ce apar noi soluţii, ele pot fi încorporate în arhitectura produsului.

Dezavantaje

Metodologia ciclică nu are nici o modalitate de supraveghere care să controleze oscilaţiile de la un ciclu la altul.

În această situaţie, fiecare ciclu produce un efort mai mare de muncă pentru ciclul următor, ceea ce încarcă orarul planificat şi poate duce la eliminarea unor funcţii sau la o calitate scăzută.

Lungimea sau numărul de cicluri poate creşte foarte mult.

De vreme ce nu există constrângeri asupra echipei de analiză să facă lucrurile cum trebuie de prima dată, acest fapt duce la scăderea responsabilităţii.

Echipa de implementare poate primi sarcini la care ulterior se va renunţa.

Echipa de proiectare nu are o viziune globală asupra produsului şi deci nu poate realiza o arhitectură completă.

Nu există termene limită precise.

Ciclurile continuă fără o condiţie clară de terminare.

Echipa de implementare poate fi pusă în situaţia nedorită în care arhitectura şi cerinţele sistemului sunt în permanenţă schimbare.

3.2.1.3. Metodologia hibridă ecluză

Metodologia ecluză, propusă de Ronald LeRoi Burback (1998), separă aspectele cele mai importante ale procesului de dezvoltare a unui produs software de detaliile mai puţin semnificative şi se concentrează pe rezolvarea primelor.

- Pe măsură ce procesul continuă, detaliile din ce în ce mai fine sunt rafinate, până când produsul poate fi lansat.

- Această metodologie hibridă (fig.3.3) preia natura iterativă a metodologiei spirală, la care adaugă progresul sigur al metodologiei cascadă.

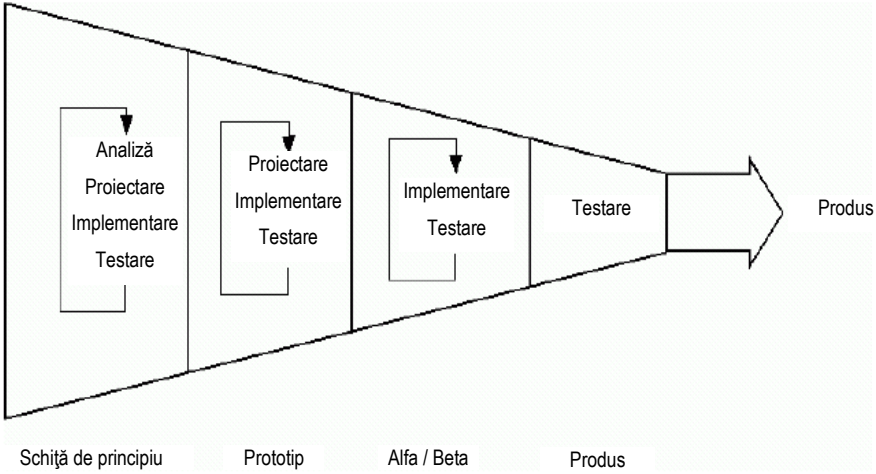

Fig. 3.3. Metodologia hibridă ecluză

- La început, într-un proces iterativ, fazele de analiză, proiectare, implementare şi testare sunt împărţite în mai multe sarcini potenţiale, fiecăruia atribuindu-i-se o prioritate.

- La fiecare moment se execută sarcina cu prioritate maximă.

- În funcţie de dimensiunea echipelor, mai multe sarcini pot fi realizate în paralel.

- Sarcinile rămase, de prioritate minimă, sunt păstrate pentru examinare ulterioară.

- Descompunerea problemei este foarte importantă.

- Cu cât descompunerea şi stabilirea priorităţilor sunt mai bune, cu atât mai eficientă este metodologia.

- Priorităţile se stabilesc pe baza unei *funcţii de prioritate*, care depinde atât de domeniul problemei şi cât şi de normele firmei.

- Ea trebuie să realizeze un compromis între cantitate şi calitate, între funcţionalitate şi constrângerile privind resursele, între aşteptări şi realitate.

-Toate funcţiile de prioritate ar trebuie să aibă ca prim scop lansarea produsului.

- funcţia mai trebuie să gestioneze sarcinile conflictuale şi nemonotone.

- Odată ce o componentă este terminată şi acceptată de echipă, schimbările asupra sa sunt îngheţate.

- Componenta va fi schimbată numai dacă modificările sunt absolut necesare iar echipa este dispusă să întârzie lucrul la restul sistemului pentru a le efectua.

- Schimbările trebuie să fie puţine la număr, bine justificate şi documentate.

Etapele principale ale metodei sunt:

- schiţa de principiu,
- prototipul,
- versiunile alfa/beta şi
- produsul final.

În prima etapă, ***schiţa de principiu***, echipele lucrează simultan la toate fazele problemei.

- Echipa de analiză sugerează cerinţele.

- Echipa de proiectare le discută şi trimite sarcinile critice de implementare echipei de implementare.

- Echipa de testare pregăteşte şi dezvoltă mediul de test în funcţie de cerinţe.

- Echipa de implementare se concentrează asupra sarcinilor critice, care în general sunt cele mai dificile. - contrastează cu practica curentă de realizare mai întâi a sarcinilor simple.

Dacă nu sunt respectate cu stricteţe etapele sistemele pot să eşueze.

Odată ce *componentele critice* au fost realizate, sistemul este gata de a face tranziţia către ***stadiul de prototip***.

Unul din scopurile aceste etapei este de a se convinge echipele că o soluţie poate fi găsită şi pusă în practică.

În cea de a doua etapă, de ***prototip***, cerinţele şi documentul cerinţelor sunt îngheţate.

- Schimbările în cerinţe sunt încă permise, însă ar trebuie să fie foarte rare şi numai dacă sunt absolut necesare, deoarece modificările cerinţelor în acest stadiu al proiectului sunt foarte costisitoare.

- Este posibilă totuşi ajustarea arhitecturii, pe baza noilor opţiuni datorate tehnologiei.

- După ce sarcinile critice au fost terminate, echipa de implementare se poate concentra pe extinderea acestora, pentru definirea cât mai multor aspecte ale aplicaţiei.

- Unul din scopurile acestei etape este de a convinge persoanele din afara echipelor că o soluţie este posibilă.

Acum produsul este gata pentru lansarea versiunilor ***alfa şi beta***.

- Arhitectura este îngheţată, iar accentul cade pe implementare şi asigurarea calităţii.

- Prima versiune lansată se numeşte în general *alfa*.

Produsul este încă imatur; numai sarcinile critice au fost implementate la calitate ridicată.

Numai un număr mic de clienţi sunt în general dispuşi să accepte o versiune alfa şi să-şi asume riscurile asociate.

O a doua lansare reprezintă versiunea *beta*.

Rolul său este de a convinge clienţii că aplicaţia va fi un produs adevărat şi de aceea se adresează unui număr mai mare de clienţi.

Când o parte suficient de mare din sistem a fost construită, poate fi lansat în sfârşit ***produsul***.

În această etapă, implementarea este îngheţată şi accentul cade pe asigurarea calităţii.

Scopul este realizarea unui produs competitiv.

În produsul final nu se acceptă erori critice.

Avantaje:

- Metodologia ecluză recunoaşte faptul că oamenii fac greşeli şi că nici o decizie nu trebuie să fie absolută.

- Echipele nu sunt blocate într-o serie de cerinţe sau într-o arhitectură imobilă care se pot dovedi mai târziu inadecvate sau chiar greşite.

- Totuşi, metodologia impune date de îngheţare a unor faze.

- Există timp suficient pentru corectarea greşelilor decizionale pentru atingerea unui nivel suficient de ridicat de încredere.

- Se pune mare accent pe comunicarea între echipe, ceea ce reduce cantitatea de cod inutil la care ar trebui să se renunţe în mod normal.

- Metodologia încearcă să mute toate erorile la începutul procesului, unde corectarea, sau chiar reînceperea de la zero a lucrului, nu sunt foarte costisitoare.

Dezavantaje

Metodologia presupune asumarea unor responsabilităţi privind delimitarea etapelor şi îngheţarea succesivă a fazelor de dezvoltare.

Ea presupune crearea unui mediu de lucru în care acceptarea responsabilităţii pentru o decizie care se dovedeşte mai târziu greşită să nu se repercuteze în mod negativ asupra individului.

Se doreşte de asemenea schimbarea atitudinii echipelor faţă de testare, care are loc încă de la început, şi faţă de comunicarea continuă, care poate fi dificilă, întrucât cele patru faze reprezintă perspective diferite asupra realizării produsului.

3.2.2. Metodologii concrete

3.2.2.1. Metodologia cascadă

Metodologia cascadă, propusă de Barry Boehm, este una din cele mai cunoscute exemple de metodologie de ingineria programării.

Există numeroase variante ale acestui proces.

Într-o variantă detaliată, metodologia cascadă cuprinde etapele prezentate în fig.3.4.

După fiecare etapă există un pas de acceptare.

Procesul „curge” de la etapă la etapă, ca apa într-o cascadă.

În descrierea originară a lui Boehm, există o întoarcere, un pas înapoi interactiv între fiecare două etape.

Astfel, metoda cascadă este de fapt o combinaţie de metodologie secvenţială cu elemente ciclice.

Totuşi, în practica inginerească, termenul „cascadă” este utilizat ca un nume generic pentru orice metodologie secvenţială.

Acesta este modelul după care de obicei sistemele sunt dezvoltate în practică.

Există o mare atracţie pentru acest model datorită experienţei, tradiţiei în aplicarea sa şi succesului pe care l-a implicat.

O sarcină complexă este împărţită în mai mulţi paşi mici, ce sunt mai uşor de administrat.

Fiecare pas are ca rezultat un produs bine definit (documente de specificaţie, model, etc.)

Modelul cascadă cu feedback propune remedierea problemelor descoperite în pasul precedent.

Problemele la pasul *i* care sunt descoperite la pasul *i* + 3 rămân neremediabile.

Un model realist ar trebui să ofere posibilitatea ca de la un anumit nivel să se poată reveni la oricare dintre nivelele anterioare.

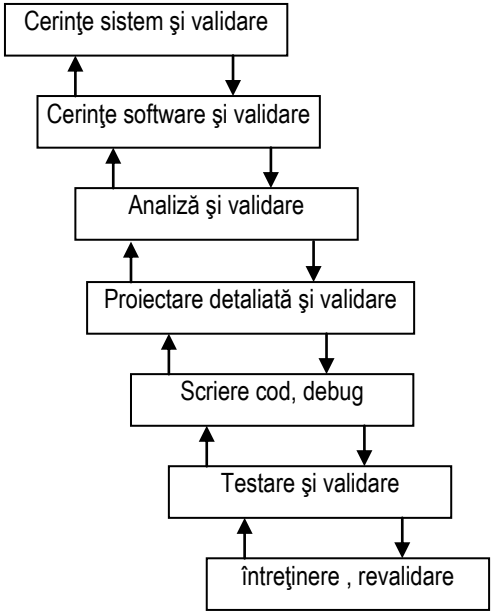

Fig. 3.4. Metodologia cascadă

Dezavantaje:

- clientul obţine o viziune practică asupra produsului doar în momentul terminării procesului de dezvoltare.

- modelul nu are suficientă putere descriptivă, în sensul că nu integrează activităţi ca managementul resurselor sau managementul configuraţiei. Aceasta face dificilă coordonarea proiectului.

- şi modelul cascadă impune îngheţarea specificaţiilor foarte devreme în procesul de dezvoltare pentru a evita iteraţiile frecvente

- reîntoarcerile în fazele anterioare atunci când în faza curentă s-au detectat erori:

  - în timpul analizei se descoperă erori de specificaţii,

  - în timpul implementării se descoperă erori de specificaţii/proiectare etc.,

astfel încât procesul poate implica multiple secvenţe de iteraţii ale activităţilor de dezvoltare).

- Îngheţarea prematură a cerinţelor conduce la :

- obţinerea unui produs prost structurat şi care nu execută ceea ce doreşte utilizatorul.

- obţinerea unei documentaţii neadecvate deoarece schimbările intervenite în iteraţiile frecvente nu sunt actualizate în toate documentele produse.

3.2.2.2. Metodologia spirală

Metodologia spirală, propusă tot de Boehm, este un alt exemplu bine cunoscut de metodologie a ingineriei programării.

Acest model încearcă să rezolve problemele modelului în cascadă, păstrând avantajele acestuia: planificare, faze bine definite, produse intermediare.

El defineşte următorii paşi în dezvoltarea unui produs:

- studiul de fezabilitate;
- analiza cerinţelor;
- proiectarea arhitecturii software;
- implementarea.

Modelul în spirală (fig. 3.5.) recunoaşte că problema principală a dezvoltării programelor este riscul.

Riscul nu mai este eliminat prin aserţiuni de genul: „în urma proiectării am obţinut un model corect al sistemului”, ca în modelul cascadă.

Aici riscul este acceptat, evaluat şi se iau măsuri pentru contracararea efectelor sale negative.

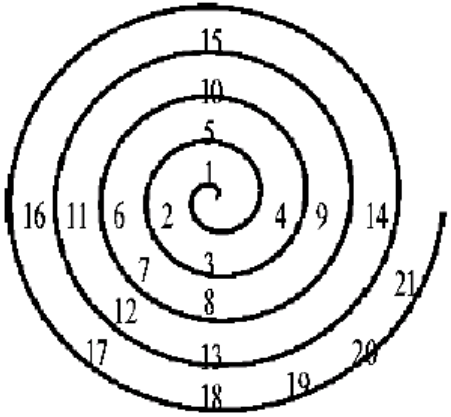

Fig. 3.5. Modelul spirală

Exemple de riscuri:

⇒ în timpul unui proces îndelungat de dezvoltare, cerinţele noi ale clientului sunt ignorate;

⇒ clientul schimbă cerinţele;

⇒ o firmă concurentă lansează un program rival pe piaţă;

⇒ un dezvoltator /arhitect părăseşte echipa de dezvoltare;

⇒ o echipă nu respectă un termen de livrare pentru o anumită componentă

Metodologia spirală cuprinde următoarele etape, grupate pe patru cicluri (Tabelul 3.1)

În modelul spirală se consideră că fiecare pas din dezvoltare conţine o serie de activităţi comune:

- pregătirea: se identifică obiectivele, alternativele, constrângerile;
- gestionarea riscului: analiza şi rezolvarea situaţiilor de risc;
- activităţi de dezvoltare specifice pasului curent (de exemplu analiza specificaţiilor sau scrierea de cod);
- planificarea următorului stadiu: termenele limită, resurse umane, revizuirea stării proiectului.

Procesul începe în centrul spiralei. Fiecare ciclu terminat reprezintă o etapă. Pe măsură ce spirala este parcursă, produsul se maturizează. Cu fiecare ciclu, sistemul se apropie de soluţia finală.

Deşi este considerată ca un exemplu generic pentru metodologia ciclică, metoda are şi elemente secvenţiale, puse în evidenţă de evoluţia constantă de la o etapă la alta.

Tabelul 3.1. Ciclurile metodologiei în spirală

Ciclul 1 – Analiza preliminară:
1. Obiective, alternative, constrângeri 2. Analiza riscului şi prototipul 3. Conceperea operaţiilor 4. Cerinţele şi planul ciclului de viaţă 5. Obiective, alternative, constrângeri 6. Analiza riscului şi prototipul
Ciclul 2 – Analiza finală:
7. Simulare, modele, benchmark-uri

8. Cerinţe software şi acceptare 9. Plan de dezvoltare 10. Obiective, alternative, constrângeri 11. Analiza riscului şi prototipul
Ciclul 3 – Proiectarea:
12. Simulare, modele, benchmark-uri 13. Proiectarea produsului software, acceptare şi verificare 14. Integrare şi plan de test 15. Obiective, alternative, constrângeri 16. Analiza riscului şi prototipul operaţional
Ciclul 4 – Implementarea şi testarea:
17. Simulare, modele, benchmark-uri 18. Proiectare detaliată 19. Cod

20. Integrarea unităţilor şi testarea acceptării
21. Lansarea produsului

3.2.2.4. Metodologia spirală WinWin

Această metodologie extinde spirala Boehm prin adăugarea unui pas de stabilire a priorităţii la începutul fiecărui ciclu din spirală şi prin introducerea unor scopuri intermediare, numite *puncte ancoră*.

- Modelul spirală WinWin identifică un punct de decizie.

- Pentru fiecare punct de decizie, se stabilesc obiectivele, constrângerile şi alternativele.

- Punctele ancoră stabilesc trei scopuri intermediare.

- Primul punct ancoră, numit *obiectivul ciclului de viaţă*, precizează cazurile sigure de funcţionare pentru întregul sistem, arătând că există cel puţin o arhitectură fezabilă (adică posibilă din punct de vedere practic) care satisface scopurile sistemului.

- Primul scop intermediar este stabilit când sunt terminate obiectivele de nivel înalt ale sistemului, arhitectura, modelul ciclului de viaţă şi prototipul sistemului.

- Această primă ancoră spune de ce, ce, când, cine, unde, cum şi estimează costul produsului.

După executarea acestor operaţii, este disponibilă analiza de nivel înalt a sistemului.

- Al doilea punct ancoră defineşte *arhitectura ciclului de viaţă*, iar al treilea – *capacitatea operaţională iniţială*, incluzând mediul software necesar, hardware-ul, documentaţia pentru client şi instruirea acestuia.

Aceste puncte ancoră corespund etapelor majore din ciclul de viaţă al unui produs: dezvoltarea iniţială, lansarea, funcţionarea, întreţinerea şi ieşirea din funcţiune.

3.2.2.5. Prototipizarea

O problemă generală care apare la dezvoltarea unui program este să ne asigurăm că utilizatorul obţine exact ceea ce vrea. Prototipizarea vine în sprijinul rezolvării acestei probleme. Încă din primele faze ale dezvoltării, clientului i se prezintă o versiune funcţională a sistemului.

Această versiune nu reprezintă întregul sistem, însă este o parte a sistemului care cel puţin funcţionează. Prototipul ajută clientul în a-şi defini mai bine cerinţele şi priorităţile. Prin intermediul

unui prototip, el poate înţelege ce este posibil şi ce nu din punct de vedere tehnologic. Prototipul este de obicei produs cât mai repede; pe cale de consecinţă, stilul de programare este de obicei (cel puţin) neglijent. Însă scopul principal al prototipului este de a ajuta în fazele de analiză şi proiectare şi nu folosirea unui stil elegant.

Se disting două feluri de prototipuri:

- jetabil sau de aruncat (throw-away);
- evoluţionar sau lent şi recuperabil.

În cazul realizării unui prototip jetabil, scopul este exclusiv obţinerea unei specificaţii. De aceea nu se acordă nici o importanţă stilului de programare şi de lucru, punându-se accent pe viteza de dezvoltare. Odată stabilite cerinţele, codul prototipului este „aruncat”, sistemul final fiind rescris de la început, chiar în alt limbaj de programare.

În cazul realizării unui prototip evoluţionar, scopul este de a crea un schelet al aplicaţiei care să poată implementa în primă fază o parte a cerinţelor sistemului. Pe măsură ce aplicaţia este dezvoltată, noi caracteristici sunt adăugate scheletului existent. În contrast cu prototipul de aruncat,

aici se investeşte un efort considerabil într-un design modular şi extensibil, precum şi în adoptarea unui stil elegant de programare.

Această metodă are următoarele avantaje:

- permite dezvoltatorilor să elimine lipsa de claritate a specificaţiilor;
- oferă utilizatorilor şansa de a schimba specificaţiile într-un mod ce nu afectează drastic durata de dezvoltare;
- întreţinerea este redusă, deoarece acceptarea se face pe parcursul dezvoltării;
- se poate facilita instruirea utilizatorilor finali înainte de terminarea produsului.

Dintre dezavantajele principale ale prototipizării amintim:

- deoarece prototipul rulează într-un mediu artificial, anumite dezavantaje ale produsului final pot fi scăpate din vedere de clienţi;
- clientul nu înţelege de ce produsul necesită timp suplimentar pentru dezvoltare, având în vedere că prototipul a fost realizat atât de repede;
- deoarece au în fiecare moment şansa de a face acest lucru, clienţii schimbă foarte des specificaţiile;

- poate fi nepopulară printre dezvoltatori, deoarece implică, în cazul prototipului jetabil, renunţarea la propria muncă.

Acest model, cunoscut de asemenea sub numele de prototip rapid, încearcă să rezolve neajunsul modelului în cascadă legat de faptul că până la construirea produsului final nu se ştie cu exactitate ce a dorit într-adevăr utilizatorul. Dezvoltarea evolutivă este similară abordării exploratorii, dar scopul acestei dezvoltări este de a stabili cerinţele utilizatorului, mai ales atunci când este dificil de făcut acest lucru, sau când documentele de specificaţii existente sunt ambigui sau incomplete. Această tehnică este în esenţă o metodă de analiză a specificaţiilor.

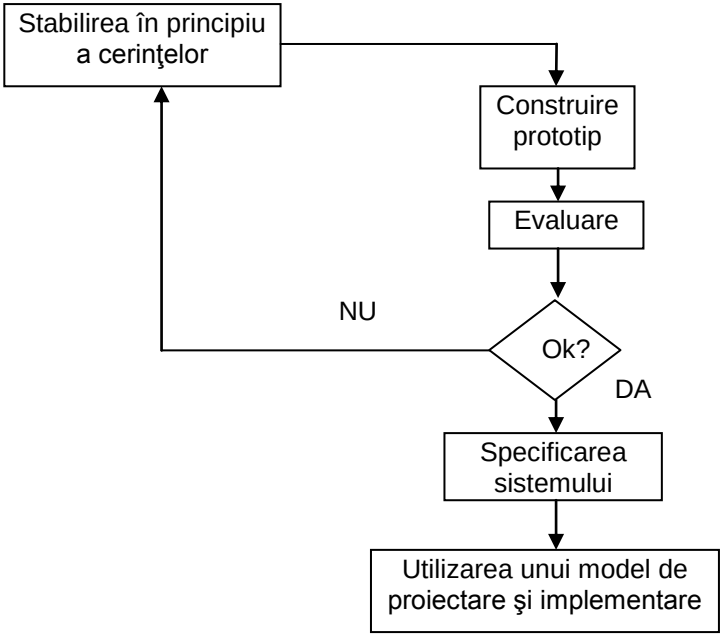

Fig.3.6. Modelul prototipului rapid

Plecând de la stabilirea în mare a specificaţiilor, inginerul software construieşte un prototip rapid şi lasă clientul să-l experimenteze. În momentul în care clientul este satisfăcut, proiectantul poate trece la elaborarea documentului cu specificaţiile produsului, care este apoi folosit în construcţia software-ului utilizând, de exemplu, un model în cascadă (fig.3.7).

Când se construieşte un prototip, proiectarea şi implementarea se realizează în faze diferite de timp.

Cele două abordări pot fi combinate în mod util aşa cum se arată în fig. 3.7. În acest caz, stadiul de analiză a cerinţelor din cadrul modelul în cascadă a fost înlocuit de stadiul necesar al prototipizării iar bucla de feedback a dispărut.

Punctul forte al acestui model combinat este acela că, în acest mod dezvoltarea software-ului devine, în mod esenţial, un proces complet liniar. Datorită completitudinii specificaţiilor utilizatorului, buclele de reacţie (feedback) din stadiile precedente par a fi, din ce în ce mai puţin, necesare.

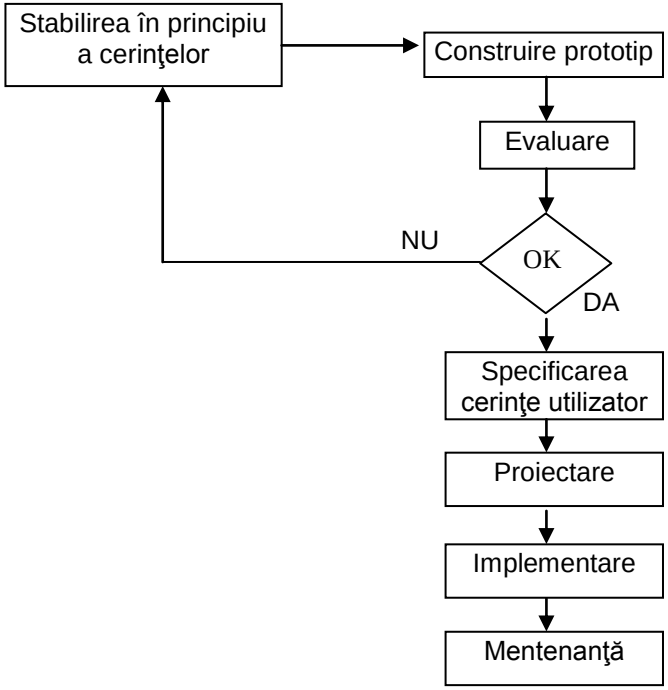

Fig.3.7. Modelul integrat “cascadă şi prototipizare”

3.2.2.6. Metode formale

În acest model de dezvoltare, sunt folosite formalismul şi rigoarea matematicii. În prima fază este construită o specificaţie în limbaj matematic. Apoi, această specificaţie este transformată în programe, de obicei într-un proces incremental.

Avantaje:

- precizia obţinută prin specificarea formală;
- păstrarea corectitudinii în timpul transformării specificaţiei în cod executabil;
- oferă posibilitatea generării automate de cod;
- sunt potrivite pentru sisteme cu cerinţe critice.

Dezavantaje:

- specificarea formală este de obicei o barieră de comunicare între client şi analist;
- necesită personal foarte calificat (deci mai scump);
- folosirea impecabilă a tehnicilor specificării formale nu implică neapărat obţinerea de programe sigure, deoarece anumite aspecte critice pot fi omise din specificaţiile iniţiale.

3.2.2.7. Programarea Extrema

Programarea Extrema (Kent Beck, 1996) este o metodologie care propune rezolvări originale pentru problemele care apar în dezvoltarea de programe.

Este o metodă de programare „agilă”, potrivită dezvoltării rapide de aplicaţii

Fiind o tehnologie nouă (şi extremă) are atât adepţi cât şi critici. XP consideră că dezvoltarea programelor nu înseamnă ierarhii, responsabilităţi şi termene limită, aşa cum se află acestea pe masa administratorului, ci înseamnă colaborarea oamenilor din care este formată echipa. Aceştia sunt încurajaţi să îşi afirme personalitatea, să ofere şi să primească cunoştinţe şi să devină programatori străluciţi.

De asemenea, XP consideră că dezvoltarea de programe înseamnă în primul rând scrierea de programe. Această sintagmă banală se pare că este uitată de multe companii care se ascund în spatele proceselor de dezvoltare stufoase, a şedinţelor şi a rapoartelor de activitate. XP ne aminteşte cu respect ca fişierele PowerPoint nu se pot compila.

De altfel, inspirarea proceselor de dezvoltare a programelor din ingineria construcţiilor se pare că nu este cea mai fericită alegere. Este adevărat că un inginer care vrea să construiască un pod peste un râu face mai întâi măsurători, realizează un proiect şi abia apoi trece la execuţie, toate acestea într-un mod secvenţial şi previzibil. Dar dezvoltarea de programe nu seamănă cu aşa ceva, oricât am vrea să credem asta. Dacă inginerului constructor respectiv i s-ar schimba cerinţele de rezistenţă şi i s-ar muta malurile chiar când a terminat de construit jumătate de pod, putem fi siguri că acel inginer şi-ar schimba modul de lucru. Din păcate însă, nu ştim (încă) cum. Iniţiatorii XP definesc următoarele două carte, ca bază filosofică pentru această metodologie.

Principiile metodelor agile

- Implicarea clientului

  - Clientul trebuie implicat pe tot parcursul procesului de dezvoltare. Rolul său este de a prioritiza noile cerinţe şi de a evalua iteraţiile sistemului

- Livrarea incrementală

  - Programul este dezvoltat incremental, clientul indicând cerinţele care trebuie incluse la fiecare iteraţie

- Oamenii nu procesul

  - Abilităţile echipei de dezvoltare trebuie recunoscute şi exploatate. Echipa trebuie lăsată să-şi contureze propriile modalităţi de lucru, fără a i se da reţete

- Acceptarea schimbării

  - Echipa trebuie să se aştepte ca cerinţele să se schimbe iar proiectarea sistemului trebuie făcută astfel încât să se adapteze uşor la aceste schimbări

- Menţinerea simplităţii

  - Concentrare pe simplitate atât în programele dezvoltate cât şi în procesul de dezvoltare. Oricând este posibil, trebuie eliminată complexitatea din sistem

Problemele metodelor agile

- Este dificilă menţinerea interesului clienţilor implicaţi în proces
- Membrii echipei pot fi incapabili să se adapteze la implicarea intensă caracteristică metodelor agile
- Când există mai mulţi factori de decizie, este dificilă prioritizarea schimbărilor
- Menţinerea simplităţii necesită lucru suplimentar

- Contractele pot fi o problemă în cazul dezvoltării iterative

Carta drepturilor dezvoltatorului:

- Ai dreptul să ştii ceea ce se cere, prin cerinţe clare, cu declaraţii clare de prioritate;
- Ai dreptul să spui cât îţi va lua să implementezi fiecare cerinţă, şi să îţi revizuieşti estimările în funcţie de experienţă;
- Ai dreptul să îţi accepţi responsabilităţile, în loc ca acestea să-ţi fie asignate;
- Ai dreptul să faci treabă de calitate în orice moment;
- Ai dreptul la linişte, distracţie şi la muncă productivă şi plăcută.

Carta drepturilor clientului:

- Ai dreptul la un plan general, să ştii ce poate fi făcut, când, şi la ce preţ;
- Ai dreptul să vezi progresul într-un sistem care rulează şi care se dovedeşte că funcţionează trecând teste repetabile pe care le specifici tu;
- Ai dreptul să te răzgândeşti, să înlocuieşti funcţionalităţi şi să schimbi priorităţile;
- Ai dreptul să fii informat de schimbările în estimări, suficient de devreme pentru a putea reduce cerinţele astfel ca munca să se termine la data prestabilită. Poţi chiar să te opreşti

la un moment dat şi să rămâi cu un sistem folositor care să reflecte investiţia până la acea dată.

Aceste afirmaţii, deşi par de la sine înţelese, conţin semnificaţii profunde. Multe din problemele apărute în dezvoltarea programelor pornesc de la încălcarea acestor principii. Enumerăm pe scurt câteva dintre caracteristicile XP:

- Echipa de dezvoltare nu are o structură ierarhică. Fiecare contribuie la proiect folosind maximul din cunoştinţele sale;
- Scrierea de cod este activitatea cea mai importantă;
- Proiectul este în mintea tuturor programatorilor din echipă, nu în documentaţii, modele sau rapoarte;
- La orice moment, un reprezentant al clientului este disponibil pentru clarificarea cerinţelor;
- Codul se scrie cât mai simplu;
- Se scrie mai întâi cod de test;
- Dacă apare necesitatea rescrierii sau eliminării codului, aceasta se face fără milă;
- Modificările aduse codului sunt integrate continuu (de câteva ori pe zi);

- Se programează în echipă (programare în perechi). Echipele se schimbă la sfârşitul unei iteraţii (1-2 săptămâni);
- Se lucrează 40 de ore pe săptămână, fără lucru suplimentar.

Concluzii

Au fost prezentate aici cele mai importante metodologii de dezvoltare a programelor, mai puţin metodologia orientată obiect care a devenit în momentul de faţă, suverană şi despre care vom vorbi în continuare. Mai întâi au fost descrise metodologiile generice: secvenţială, ciclică şi hibridă, cu avantajele şi dezavantajele fiecăreia. Apoi s-au amintit câteva metode concrete de dezvoltate: modelul cascadă, modelul spirală, WinWin, prototipizarea, metodologia Booch, metodele formale şi aşa-numita „programare extremă”.

3.2.2.8. Metodologia Open Source

Metodologia Open Source înseamnă “sursă la vedere”. Este o abordare recentă, apărută ca urmare a dezvoltării mijloacelor de comunicaţie: FTP, e-mail, grupuri de discuţie Exemple clasice sunt: sistemul de operare Linux, browser-ul Netscape 5.

Codul sursă este transmis utilizatorului final într-o manieră non-proprietară (fără patent), pe baza unei licenţe open-source (gen GNU+ sistem de operare Open Source gen Unix).

Critici

- Nu există documente de proiectare sau alte documentaţii ale proiectului
- Nu se realizează testarea la nivel de sistem
- Nu există cerinţe ale utilizatorilor în afară de funcţionalitatea de bază
- Marketingul produsului este incomplet

O iteraţie tipică pentru o astfel de abordare este:

- Dezvoltatorul realizează proiectarea şi codarea (individual sau în echipă);
- Ceilalţi dezvoltatori sau comunitatea de utilizatori realizează debugging-ul şi testare;
- Noile funcţionalităţi dorite şi erorile depistate sunt trimise iniţiatorului proiectului;
- Se lansează o nouă versiune cu erorile corectate şi se analizează noile cerinţele;
- Se distribuie o listă de sarcini către comunitatea de utilizatori, căutându-se membri voluntari care să execute sarcinile de pe listă;

3.2.2.9. Reverse Engineering - Inginerie inversă

- Se analizează sistemele anterioare şi se încearcă reutilizarea componentelor existente:
  - Software, hardware
  - Documentaţie, metode de lucru
- Unele componente nu pot fi utilizate, altele trebuie modificate (în faza de proiectare);
- Componentele selectate sunt integrate direct în sistem.

3.2.2.10. Metodologia de dezvoltare Offshore

Offshore Înseamnă în limba engleză , "în larg". Companiile consideră ca profitabil outsourcing-ul pentru funcţiile care pot fi dezvoltate mai ieftin în altă ţară

Venituri din outsourcing IT în 2004 au fost:

- India – 43%
- Canada – 32%
- China – 5%
- Europa de Est – 5%

Motivarea externalizării (outsourcing) este:

- Reducerea costurilor;
- Creşterea eficienţei;
- Concentrarea asupra obiectivelor critice ale proiectului;
- Accesarea flexibilă a unor resurse care altfel nu ar fi accesibile (de exemplu personal înalt calificat)
- Dimensiunea mare a proiectului.

Etapele metodologiei sunt:

1. Iniţierea proiectului (local)
2. Analiza cerinţelor (local)
3. Proiectarea de nivel înalt (local)
4. Proiectarea detaliată (offshore)
5. Implementarea (offshore)
6. Testarea (offshore sau local)
7. Livrarea (local).

3.2.2.12. Metodologia orientată pe obiect

Metodologia proiectării orientate pe obiect “inversează” metodologiile anterioare, în special metodologiile funcţionale, aşa cum au fost ele concepute de Yourdan, în sensul că focalizează abordarea pe identificarea obiectelor din domeniul aplicaţiei, “potrivind” apoi procedurile în jurul acestor obiecte. La o eventuală modificare a cerinţelor , nu va mai fi necesar să fie schimbată toată structura obiectelor.

Din start, termenul *orientat pe obiect* înseamnă organizarea software-ului ca o colecţie de obiecte discrete, fiecare obiect încorporând atât structuri de date, cât şi comportament.

Ce este un obiect?

Un obiect poate fi considerat o entitate care încorporează atât structuri de date, numite atribute, cât şi comportament, denumit operaţii.

Un obiect trebuie să aibă caracteristicile următoare:

- **Identitate** : obiectul este o entitate discretă, care se distinge dintre alte entităţi. Exemple: o fereastră pe o staţie de lucru, un triunghi isoscel, o listă de persoane.

- **Clasificare** : obiectele cu aceleaşi atribute şi operaţii se grupează în clase. Fiecare obiect cu aceleaşi atribute şi operaţii poate fi considerat ca o instanţă a unei clase. Exemple: fereastră, triunghi, listă.

- **Polimorfism** : aceeaşi operaţie (cu acelaşi nume) poate să aibă comportament diferit în clase diferite. Exemple: *a muta* o fereastră, *a muta* un triunghi. Operaţia *a muta* înseamnă lucruri diferite, depinzând de obiectul asupra căruia se aplică. Implementarea concretă a unei operaţii într-o clasă se numeşte metodă.

- **Moştenire**: Este o caracteristică a abordării obiectuale şi se referă la transmiterea pe cale ierarhică a atributelor şi metodelor tuturor claselor descendente, într-o relaţie ierarhică.

Modele de lucru în OMT

Metodologia de proiectare orientată pe obiect s-a dezvoltat iniţial ca tehnică de modelare a obiectelor cunoscută şi sub numele de OMT, în engleză Object Modelling Technique şi a evoluat apoi printr-o tehnologie unificată numită UML.

OMT-ul foloseşte trei modele de bază şi anume: modelul obiectelor, modelul dinamic, modelul funcţional.

Modelul obiectelor:

- descrie structura statică a obiectelor din sistem şi relaţiile dintre ele;

- descrie ce se modifică în sistem (adică obiectele);

- este reprezentat cu ajutorul diagramelor de obiecte.

O diagramă de obiecte este un graf ale cărui noduri sunt *obiectele* şi ale cărui arce sunt *relaţiile* dintre obiecte.

Modelul dinamic:

- descrie acele aspecte ale sistemului care se schimbă în timp;

- specifică şi implementează partea de control a sistemului;

- descrie când se modifică sistemul;

- este reprezentat cu ajutorul diagramelor de stare.

Modelul funcţional:

- descrie transformările valorilor datelor în cadrul sistemului;

- descrie cum se modifică sistemul;

- este reprezentat cu ajutorul diagramelor de flux de date.

O diagramă de flux de date este un graf ale cărui noduri sunt procesele şi ale cărui arce sunt fluxurile de date.

Etapele aplicării metodologiei orientate pe obiect

Metodologia OO pentru dezvoltarea software-ului constă din construirea, în etapa de analiză a sistemului, a unui model complet al aplicaţiei, care va cuprinde cele trei modele prezentate anterior.

Ulterior se vor adăuga detaliile de implementare necesare. Etapele sunt: analiza, proiectarea sistemului, proiectarea obiectelor, implementarea sistemului.

- Analiza

Pornind de la specificarea cerinţelor, analistul va concepe un model cu obiecte din domeniul aplicaţiei şi nu al programării (ca de exemplu liste, arbori, etc).

- Proiectarea sistemului

Proiectantul de sistem va lua decizii generale asupra arhitecturii globale a sistemului, va alege o strategie de implementare şi o strategie de alocare a resurselor. De asemenea, va trebui să stabilească împărţirea sistemului mare în subsisteme.

- Proiectarea obiectelor

Proiectantul obiectelor va adăuga detalii de implementare modelului obţinut la analiză, în concordanţă cu strategia aleasă în etapa de proiectare a sistemului. Accentul se va pune acum pe algoritmii şi structurile de date folosite pentru a implementa clasele obţinute în etapa de analiză.

- Implementarea

În cele din urmă, clasele şi relaţiile dintre clase obţinute în celelalte etape vor fi traduse într-un limbaj de programare, într-o bază de date sau vor fi implementate hardware. Este important de remarcat că deşi analiza unui sistem poate să se efectueze folosind noţiuni de obiecte şi clase, nu este neapărat necesar ca implementarea să se facă într-un limbaj de programare orientat pe obiect (cum ar fi Java, C++, Smalltalk, Eiffel sau ADA). Implementarea poate fi făcută în orice limbaj de programare. Un exemplu în acest sens îl constituie biblioteca de elemente de interfaţă pentru X Window, numită Motif, care este scrisă în C, deşi a fost proiectată folosindu-se noţiuni de analiză orientată pe obiect.

Conceptele de bază

A. Abstractizarea

Accentul pentru un obiect se pune pe ce este acesta şi nu pe ce trebuie să facă, înainte de a stabili concret detaliile de implementare. De aceea, etapa esenţială în crearea unei aplicaţii orientate pe obiect este analiza şi nu implementarea, care poate deveni mecanică şi în mare parte automatizată, dacă se folosesc instrumente software specializate (gen CASE).

B. Încapsularea (ascunderea informaţiei)

Încapsularea constă în separarea aspectelor externe ale unui obiect, care sunt accesibile altor obiecte, de aspectele de implementare interne ale obiectului, care sunt ascunse celorlalte obiecte. Utilizatorul obiectului poate accesa doar anumite atribute şi operaţii, în scopul perfecţionării unui algoritm sau eliminării unor erori. Încapsularea ne va împiedica să modificăm toate caracteristicile obiectului, iar aplicaţiile care utilizează obiectul nu vor avea de suferit.

C. Împachetarea datelor şi a comportamentului în acelaşi obiect

Aceasta se referă tocmai la faptul că un obiect conţine atât structuri de date cât şi operaţii, şi permite să se ştie întotdeauna cărei clase îi aparţine o anumită metodă, chiar dacă există mai multe operaţii cu aceeaşi nume, în clase diferite.

D. Partajarea

Tehnicile orientate pe obiect promovează partajarea la diverse niveluri. Partajarea poate însemna transmiterea aceloraşi structuri de date şi respectiv, operaţii de-a lungul unor clase, dintr-o ierarhie de clase. Acest lucru are un efect benefic asupra economisirii spaţiului. Pe de altă parte,

partajarea oferă posibilitatea reutilizării proiectării, respectiv a codului anumitor clase, în proiectele ulterioare.

E. Accentul pus pe structura de obiect, nu pe cea de procedură

Este una dintre caracteristicile principale ale tehnicilor orientate pe obiect. Dacă în tehnicile funcţionale (sau procedurale) accentul în analiza sistemului cade pe descompunerea funcţională a acestuia, deci pe ceea ce trebuie să facă sistemul (în ultimă instanţă pe construirea diagramelor de flux), în tehnicile OOP accentul cade pe înţelegerea sistemului din punct de vedere al descompunerii acestuia în entităţi (obiecte) şi în stabilirea relaţiilor dintre acestea, deci pe ce este un obiect. Mult înainte de a stabili ce face sistemul, problema care se pune constă în a preciza cine execută şi care sunt relaţiile între cei care execută; deci mai importantă este diagrama obiectelor.

Analiza orientată pe obiecte

Analiza este primul pas al metodologiei tehnicii de modelare orientată pe obiecte şi are ca scop construirea unui model precis, concis şi concret al lumii reale.

În figura 3.8 este prezentată o vedere generală asupra procesului de analiză. Analistul, cel care realizează analiza problemei, începe cu definirea problemei, aşa cum este ea formulată de potenţialii

utilizatori. Analistul va construi un model, care poate fi incomplet, pe baza unor cerinţe incomplete. De aceea, modelul trebuie apoi rafinat.

Primul pas în definirea problemei este specificarea cerinţelor. Se va stabili **ceea ce** trebuie să facă aplicaţia şi **nu cum,** şi anume se vor defini:

- scopul problemei;
- necesităţile;
- contextul aplicaţiei;
- diverse presupuneri;
- performanţele necesare;

În partea de proiectare şi implementare se vor urmări:

- algoritmi;
- structurile de date;
- arhitectura;
- optimizările.

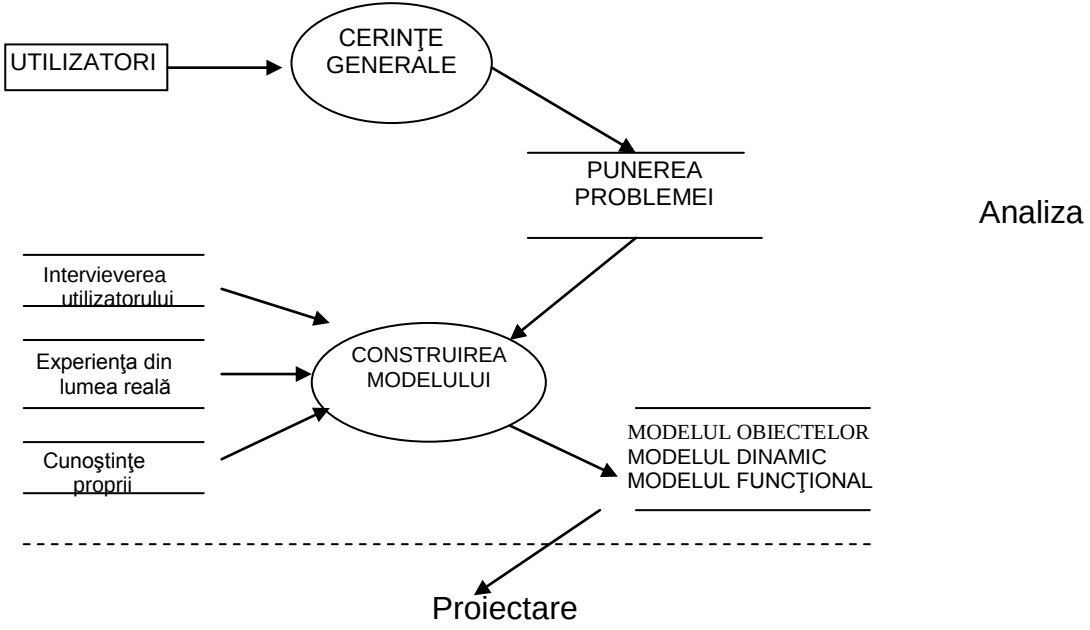

Fig.3.8. Procesul de analiza – vedere generală

Concluzii

1. Simpla adoptare a unei metodologii fără o analiză temeinică a cerinţelor şi contextului de lucru nu este fezabilă.
2. Fără sprijin din partea factorilor de decizie executivă, dezvoltarea proiectului este complexă şi de durată.
3. Metodologia este o tactică ce trebuie considerată numai după determinarea strategiei generale a companiei