Etapele de dezvoltare a sistemelor de programe

2.1. Ciclul de viaţă

Există patru faze fundamentale ale metodologiilor ingineriei software:

- analiza (ce dorim să construim);
- proiectarea (cum vom construi);
- implementarea (construirea propriu-zisă);
- testarea (asigurarea calităţii).

Deşi aceste faze se referă în mod special la *ciclul de viaţă* al produsului software, ele pot fi aplicate şi altor stadii de existenţă prin care trece un program de la „naştere” până la „moarte”: lansare, întreţinere, ieşire din uz.

2.1.1. Faza de analiză

- defineşte *cerinţele* sistemului, independent de modul în care acestea vor fi îndeplinite.
- se defineşte problema pe care clientul doreşte să o rezolve.

- Rezultatul = *documentul care conţine specificarea cerinţelor* şi care trebuie să precizeze clar *ce* trebuie construit.

> formularea problemei, aşteptările clientului sau criteriile pe care trebuie să le îndeplinească produsul.

Documentul cerinţelor, numit *specificarea cerinţelor* poate fi realizat într-o manieră formală, bazată pe logică matematică, sau poate fi exprimat în limbaj natural.

el descrie *obiectele* din sistem şi *acţiunile* care pot fi realizate cu ajutorul obiectelor.

Descrierea obiectelor şi acţiunilor trebuie să fie generală si să nu depindă de o anumită tehnologie.

Desigur, într-o abordare POO, descrierile vor lua forma obiectelor şi metodelor, însă în alte abordări, obiectele pot fi de exemplu servicii care accesează baze de date.

Descrierile nu implică proiectarea arhitecturii aplicaţiei, ci enumerarea părţilor componente şi a modului în care acestea se comportă.

Mai târziu, în faza de proiectare, acestea vor fi transformate în primitive informatice, precum liste, stive, arbori, grafuri, algoritmi şi structuri de date.

Mai concret, documentul trebuie să conţină descrieri pentru următoarele categorii:

- ***Obiecte****:* se defineşte mai întâi ontologia sistemului, care este bazată pe construcţii substantivale pentru identificarea pieselor, părţilor componente, constantelor, numelor şi a relaţiilor dintre acestea;
- ***Acţiuni****:* se definesc acţiunile pe care trebuie să le îndeplinească sistemul şi care sunt sugerate în general de construcţii verbale. Exemple de acţiuni sunt: metodele, funcţiile sau procedurile;
- ***Stări****:* Sunt definite ca mulţimi de setări şi valori care disting sistemul între două ipostaze spaţio-temporale. Exemple de stări sunt: starea iniţială, cea finală sau stările de eroare.
- ***Scenarii tipice****:* Un scenariu este o secvenţă de paşi urmaţi pentru îndeplinirea unui scop. Când sistemul este terminat şi aplicaţia este disponibilă, clientul trebuie să poată utiliza, într-o

manieră cât mai facilă si clar specificată, toate scenariile tipice ale aplicaţiei. Scenariile tipice trebuie să reprezinte majoritatea scenariilor de utilizare ale aplicaţiei.

- ***Scenarii atipice****:* Un scenariu atipic trebuie să fie îndeplinit de sistem numai în cazuri speciale. Clientul poate să spere, de exemplu, că o eroare neprevăzută este un eveniment atipic. Totuşi, sistemul trebuie să gestioneze un număr cât mai mare de categorii de erori, prin tehnici stabilite, precum tratarea excepţiilor, monitorizarea proceselor etc.;
- ***Cerinţe incomplete sau nemonotone***: O enumerare completă a cerinţelor, pentru toate situaţiile care pot apărea în condiţii de lucru reale, nu este posibilă. Procesul de stabilire a cerinţelor are o natură iterativă si nemonotonă. Noile cerinţe pot infirma soluţiile vechi. Pe măsură ce un sistem creşte în dimensiuni şi complexitate, stabilirea cerinţelor devine din ce în ce mai dificilă, mai ales când procesul de colectare a cerinţelor este distribuit, fiind realizat de indivizi cu specializări diferite.

2.1.2. Faza de proiectare

Pe baza cerinţelor din faza de analiză, acum se stabileşte ***arhitectura*** sistemului: componentele sistemului, ***interfeţele*** şi modul lor de ***comportare***:

- ***Componentele*** sunt elementele constructive ale produsului. Acestea pot fi create de la zero sau reutilizate dintr-o bibliotecă de componente. Componentele rafinează si capturează semnificaţia detaliilor din documentul cerinţelor;
- ***Interfeţele*** ajută la îmbinarea componentelor. O interfaţă reprezintă graniţa dintre două componente, utilizată pentru comunicarea dintre acestea. Prin intermediul interfeţei, componentele pot interacţiona;
- ***Comportamentul***, determinat de interfaţă, reprezintă răspunsul unei componente la stimulii acţiunilor altor componente.

Se identifică detaliile privind limbajele de programare, mediile de dezvoltare, dimensiunea memoriei, platforma, algoritmii, structurile de date, definiţiile globale de tip, interfeţele etc.

- trebuie indicate şi *priorităţile critice* pentru implementare. Acestea sugerează sarcinile care, dacă nu sunt executate corect, conduc la eşecul sistemului. Totuşi, chiar dacă priorităţile critice sunt îndeplinite, acest fapt nu duce automat la succesul sistemului, însă creşte nivelul de încredere că produsul va fi o reuşită.

Folosind scenariile tipice şi atipice, trebuie realizate compromisurile inerente între performanţă si complexitatea implementării.

Analiza performanţelor presupune studierea modului în care diferitele arhitecturi conduc la diferite caracteristici de performanţă pentru fiecare scenariu tipic.

În funcţie de frecvenţa de utilizare a scenariilor, fiecare arhitectură va avea avantaje şi dezavantaje.

Un răspuns rapid la o acţiune a utilizatorului se realizează deseori pe baza unor costuri de resurse suplimentare: indecşi, managementul cache-ului, calcule predictive etc.

Dacă o acţiune este foarte frecventă, ea trebuie realizată corect şi eficient.

O acţiune mai rară trebuie de asemenea implementată corect, dar nu este evident care e nivelul de performanţă necesar în acest caz.

O situaţie în care o astfel de acţiune trebuie implementată cu performanţe maxime este închiderea de urgenţă a unui reactor nuclear.

Planul de implementare stabileşte programul după care se va realiza implementarea şi resursele necesare (mediul de dezvoltare, editoarele, compilatoarele etc.).

Planul de test defineşte testele necesare pentru stabilirea calităţii sistemului. Dacă produsul trece toate testele din planul de test, este declarat terminat.

Cu cât testele sunt mai amănunţite, cu atât este mai mare încrederea în sistem şi deci creşte calitatea sa. Un anume test va verifica doar o porţiune a sistemului. *Acoperirea testului* este procentajul din produs verificat prin testare. În mod ideal, o acoperire de 100% ar fi excelentă, însă ea este rareori îndeplinită. De obicei, un test cu o acoperire de 90% este simplu, însă ultimele 10% necesită o perioadă de timp semnificativă.

În general, este suficient ca testele să cuprindă scenariile tipice şi atipice, fără să verifice întregul sistem, cu absolut toate firele de execuţie. Acesta poate conţine ramificaţii interne, erori sau întreruperi care conduc la fire de execuţie netestate. Majoritatea sistemelor sunt pline de bug-uri

nedescoperite. De obicei, clientul participă în mod logic la testarea sistemului şi semnalează erori care vor fi îndepărtate în versiunile ulterioare.

2.1.3. Faza de implementare

În această fază, sistemul este construit, ori plecând de la zero, ori prin asamblarea unor componente pre-existente. Pe baza documentelor din fazele anterioare, echipa de dezvoltare ar trebui să ştie exact ce trebuie să construiască, chiar dacă rămâne loc pentru inovaţii şi flexibilitate.

Echipa trebuie să gestioneze problemele legate de calitate, performanţă, biblioteci şi depanare. Scopul este producerea sistemului propriu-zis.

O problemă importantă este *îndepărtarea erorilor critice*.

Într-un sistem există trei tipuri de erori:

- *Erori critice:* Împiedică sistemul să satisfacă în mod complet scenariile de utilizare. Aceste erori trebuie corectate înainte ca sistemul să fie predat clientului şi chiar înainte ca procesul de dezvoltare ulterioară a produsului să poată continua;

- *Erori necritice:* Sunt cunoscute, dar prezenţa lor nu afectează în mod semnificativ calitatea observată a sistemului. De obicei aceste erori sunt listate în notele de lansare şi au modalităţi de ocolire bine cunoscute;
- *Erori necunoscute:* Există întotdeauna o probabilitate mare ca sistemul să conţină un număr de erori nedescoperite încă. Efectele acestor erori sunt necunoscute. Unele se pot dovedi critice, altele pot fi rezolvate cu patch-uri sau eliminate în versiuni ulterioare.

2.1.4. Faza de testare

Calitatea produsului software este foarte importantă. Multe companii nu au învăţat însă acest lucru şi produc sisteme cu funcţionalitate extinsă, dar cu o calitate scăzută. Totuşi, e mai simplu să-i explici clientului de ce lipseşte o anumită funcţie decât să-i explici de ce produsul nu este performant. Un client satisfăcut de calitatea produsului va rămâne loial firmei şi va aştepta noile funcţii în versiunile următoare.

În multe metodologii ale ingineriei software, faza de testare este o fază separată, realizată de o echipă *diferită* după ce implementarea s-a terminat. Motivul este faptul că un programator nu-şi poate descoperi foarte uşor propriile greşeli.

O persoană nouă care priveşte codul poate descoperi greşeli evidente care scapă celui care citeşte şi reciteşte materialul de multe ori.

Din păcate, această practică poate determina o atitudine indiferentă faţă de calitate în echipa de implementare.

Tehnicile de testare sunt abordate preponderent din perspectiva producătorului sistemului. În mod ideal, şi clientul trebuie să joace un rol important în această fază.

Testele de regresiune (engl. „regression test") sunt colecţii de programe care testează una sau mai multe trăsături ale sistemului. Rezultatele testelor sunt adunate şi dacă există erori, eroarea este corectată. Un test de regresiune valid generează rezultate verificate, numite „standardul de aur".

Validitatea rezultatului unui test ar trebui să fie determinată de documentul cerinţelor. În practică, echipa de implementare este responsabilă de interpretarea validităţii.

Testele sunt colectate, împreună cu rezultatele standardelor de aur, într-un pachet de test de regresiune. Pe măsură ce dezvoltarea continuă, sunt adăugate mai multe teste noi, iar testele vechi pot rămâne valide sau nu. Dacă un test vechi nu mai este valid, rezultatele sale sunt modificate în standardul de aur, pentru a se potrivi aşteptărilor curente. Pachetul de test este rulat din nou şi

generează noi rezultate. Acestea sunt comparate cu rezultatele standardelor de aur. Dacă sunt diferite, înseamnă că în sistem a apărut o eroare. Eroarea este corectată si dezvoltarea continuă. Acest mecanism detectează situaţiile când starea curentă de dezvoltare a produsului invalidează o stare existentă. Astfel, se previne regresiunea sistemului într-o stare de eroare anterioară.

Există patru puncte de interes în testele de regresiune pentru asigurarea calităţii.

Testarea internă tratează implementarea de nivel scăzut. Fiecare funcţie sau componentă este testată de către echipa de implementare. Aceste teste se mai numesc teste „clear-box” sau „white-box”, deoarece toate detaliile sunt vizibile pentru test.

Testarea unităţilor testează o unitate ca un întreg. Aici se testează interacţiunea mai multor funcţii, dar numai în cadrul unei singure unităţi.

Testarea este determinată de arhitectură. De multe ori sunt necesare aşa-numitele „schele”, adică programe special construite pentru stabilirea mediului de test.

Numai când mediul este realizat se poate executa o evaluare corectă.

Programul schelă stabileşte stări şi valori pentru structurile de date şi asigură funcţii externe fictive.

De obicei, programul schelă nu are aceeaşi calitate ca produsul software testat şi adesea este destul de fragil.

O schimbare mică în test poate determina schimbări importante în programul schelă.

Aceste teste se mai numesc teste „black-box” deoarece numai detaliile interfeţei sunt vizibile pentru test.

Testarea internă si a unităţilor poate fi automatizată cu ajutorul instrumentelor de acoperire (engl. „coverage tools”), care analizează codul sursă şi generează un test pentru fiecare alternativă a firelor de execuţie.

Depinde de programator combinarea acestor teste în cazuri semnificative care să valideze rezultatelor fiecărui fir de execuţie.

De obicei, instrumentul de acoperire este utilizat într-un mod oarecum diferit: el urmăreşte liniile de cod executate într-un test şi apoi raportează procentul din cod executat în cadrul testului.

Dacă acoperirea este mare şi liniile sursă netestate nu prezintă mare importanţă pentru calitatea generală a sistemului, atunci nu mai sunt necesare teste suplimentare.

Testarea aplicaţiei testează aplicaţia ca întreg şi este determinată de scenariile echipei de analiză.

Aplicaţia trebuie să execute cu succes toate scenariile pentru a putea fi pusă la dispoziţia clientului.

Spre deosebire de testarea internă si a unităţilor, care se face prin program, testarea aplicaţiei se face de obicei cu scripturi care rulează sistemul cu o serie de parametri şi colectează rezultatele.

În trecut, aceste scripturi erau create manual. În prezent, există instrumente care automatizează si acest proces. Majoritatea aplicaţiilor din zilele noastre au interfeţe grafice (GUI).

Testarea interfeţei grafice pentru asigurarea calităţii poate pune anumite probleme.

Cele mai multe interfeţe, dacă nu chiar toate, au bucle de evenimente, care conţin cozi de mesaje de la mouse, tastatură, ferestre etc.şi care asociate cu fiecare eveniment sunt coordonatele ecran.

Testarea interfeţei presupune deci memorarea tuturor acestor informaţii şi elaborarea unei modalităţi prin care mesajele să fie trimise din nou aplicaţiei, la un moment ulterior.

Testarea la stres determină calitatea aplicaţiei în mediul său de execuţie.

Ideea este crearea unui mediu mai solicitant decât cel în care aplicaţia va rula în mod obişnuit.

Aceasta este cea mai dificilă si complexă categorie de teste.

Sistemul este supus unor cerinţe din ce în ce mai numeroase, până când acesta cade.

Apoi produsul este reparat şi testul de stres se repetă până când se atinge un nivel de stres mai ridicat decât nivelul aşteptat de pe staţia clientului.

Deseori apar aici conflicte între teste. Fiecare test funcţionează corect atunci când este făcut separat.

Când două teste sunt rulate în paralel, unul sau ambele teste pot eşua. Cauza este de obicei managementul incorect al accesului la resurse critice.

Mai apar şi probleme de memorie, când un test îşi alocă memorie şi apoi nu o mai eliberează.

Testul pare să funcţioneze corect, însă după ce este rulat de mai multe ori, memoria disponibilă se reduce iar sistemul cade.

2.2. Cerinţe – Specificaţii

În mod logic, prima etapă în realizarea unui produs software constă în stabilirea cu precizie a ceea ce utilizatorul doreşte de la sistem.

Dominanta în această etapă o constituie comunicarea dintre beneficiar şi inginerul software.

Inginerul care se ocupă de stabilirea cerinţelor utilizatorului, ceea ce de fapt reprezintă analiza sistemului, se va numi simplu *analist.*

utilizatorului colaborează cu analistul pentru definirea cerinţelor şi specificaţiilor de proiectare ale sistemului.

Utilizatorul poate să aibă o idee foarte vagă despre ceea ce doreşte sau, din contra, el poate să ştie foarte exact.

În mod foarte categoric, stabilirea specificaţiilor de proiectare este o etapă deosebit de importantă în dezvoltarea ulterioară a sistemului.

2.2.1. Noţiunea de "cerinţă"

Cerinţa reprezintă *ceea ce doreşte* de fapt utilizatorul, fără a se preciza de fapt cum se realizează acel lucru.

Una dintre marile controverse din Computer Science se enunţă astfel:

« Este sau nu este necesar să se specifice şi *cum* realizează sistemul o anumită solicitare». Aceasta este relaţia dintre specificare şi implementare.

De cele mai multe ori, utilizatorul nu agreează să i se explice cum va fi implementat sistemul, sau mai mult, nici nu-l interesează.

Pentru analist însă acest lucru este important pentru că, în funcţie de varianta aleasă, ştie cum să-şi orienteze investigaţiile şi, de asemenea, ce restricţii are.

Analistul trebuie:

- să verifice dacă o anumită solicitare este tehnic posibilă.
- este vital să ia în considerare implementarea pentru a putea estima costul şi data de livrare a sistemului.

- dacă specificaţiile pot să constituie un ghid privind modul în care sistemul răspunde dorinţelor utilizatorilor.

Câteodată, specificaţiile suferă de deficienţe majore, cum ar fi: sunt incomplete, nu există nici o menţiune despre cost sau termen de livrare.

2.2.2. Procesul de alegere a cerinţelor

Activitatea de selectare a cerinţelor presupune colaborarea şi lucrul împreună atât al analistului cât şi al utilizatorului.

Se pot distinge trei feluri de activităţi în cadrul acestui proces de specificare a cerinţelor, şi anume:

- ascultarea (sau selectarea cererilor);
- analizarea cererilor;
- scrierea (sau definirea cererilor).

Selectarea presupune ascultarea nevoilor utilizatorilor, punând întrebări astfel încât utilizatorii să-şi poată clarifica cât mai bine cererile, ca în final să se înregistreze punctul de vedere al utilizatorului privind specificaţiile sistemului.

Analiza este etapa în care efectiv inginerul software se gândeşte cum poate transforma punctul de vedere al utilizatorului, privind sistemul, într-o reprezentare care să poată fi implementată în sistem.

Acest lucru poate fi adesea dificil din cauza existenţei unor puncte de vedere diferite ale diverşilor utilizatori.

Definirea cererilor înseamnă scrierea într-o manieră clară, adesea în limbaj natural, a ceea ce sistemul trebuie să furnizeze utilizatorului. Această informaţie se numeşte ***specificarea cererilor***.

Rezumând, rolul analistului este:

1. Să achiziţioneze şi să selecteze cererile de la utilizatori.
2. Să ajute la rezolvarea diferenţelor posibile dintre utilizatori.
3. Să avizeze utilizatorul despre ceea ce este tehnic posibil sau imposibil.
4. Să clarifice cererile utilizatorilor.

5. Să documenteze solicitările (se va vedea în secţiunea următoare).

6. Să negocieze şi în final să pună de acord diferitele opinii al utilizatorilor pentru formularea specificaţiilor.

2.2.4. Specificarea cerinţelor

Specificarea este documentul de referinţă prin care este asigurată dezvoltarea sistemului.

Cei trei factori importanţi care sunt luaţi în consideraţie sunt:

1) nivelul de detaliere - ; nevoia de a restrânge specificaţiile, pe cât posibil numai la ceea *ce* "face sistemul" şi nu la *cum* va realiza cerinţa respectivă
2) cui este adresat documentul - utilizatori şi proiectanţi;
3) notaţiile utilizate.

Utilizatorii preferă o descriere netehnică, exprimată în limbaj natural, improprie pentru specificaţii precise, consistente şi neambiguii.

Pe de altă parte, analistul având o orientare tehnică, va dori probabil să utilizeze o notaţie precisă (chiar matematică) pentru a specifica sistemul.

Există câteva notaţii pentru scrierea specificaţiilor:

- *informale* - scrise în limbaj natural, utilizate cât mai clar şi cu cât mai multă grijă posibilă;
- *formale* - utilizând notaţii matematice, cu rigoare şi consistenţă;
- *semiformale* - utilizând o combinaţie de limbaj natural cu diferite diagrame şi notaţii tabelare.

În momentul de faţă, cele mai multe specificaţii sunt scrise în limbaj natural, la care se adaugă notaţii formale, cum ar fi diagramele de flux, diagramele UML pentru a clarifica textul.

Varianta modernă este de a elabora două documente şi anume:

1. Specificaţiile cererilor scrise în primul rând pentru utilizatori, pentru a descrie punctul lor de vedere asupra sistemului, exprimat în limbaj natural. Acestea constituie substanţa contractului dintre utilizatori şi proiectanţi.
2. O specificare tehnică, care este folosită în primul rând de proiectanţi, exprimată într-o formă de notaţii formale şi descriind numai o parte a informaţiei în toate specificaţiile cererilor.

O lista de verificare a conţinutului pentru specificaţiile solicitărilor trebuie să arate astfel:

- solicitări funcţionale;
- solicitări de date;
- restricţii;
- ghid de aplicare.

Solicitări funcţionale

sunt esenţa specificaţiilor cererilor. Ele indică sistemului "ceea ce trebuie să facă" şi sunt caracterizate de verbe şi realizează acţiuni.

Exemple:

- Sistemul va afişa titlurile şi toate cărţile scrise de acelaşi autor.
- Sistemul va afişa permanent temperaturile tuturor maşinilor.

Solicitări de date - au două componente:

1. Date de intrare sau ieşire pentru sistem, cum ar fi de exemplu dimensiunile ecranului;
2. Date care sunt memorate în sistem, de obicei în fişiere pe disc. De exemplu, informaţia despre cărţile dintr-o bibliotecă publică.

Restricţiile

Acestea au de regulă influenţe asupra implementării sistemului. Exemple:

"Sistemul trebuie scris în limbajul de programare ADA", " Sistemul va răspunde utilizatorului în timp de o secundă". Principalele restricţii care pot apare sunt:

1). Costul
2). Termenul de livrare
3). Echipamentul hardware necesar
4). Capacitatea internă a memoriei
5). Capacitatea memoriei externe
6). Timpul de răspuns
7). Limbajul de programare care trebuie utilizat
8). Volumul de date (ex. sistemul trebuie să memoreze informaţii despre 10000 angajaţi)
9). Nivelul de încărcare pentru terminale pe unitatea de timp
10). Fiabilitatea solicitărilor (ex. sistemul trebuie să aibă un anumit timp mediu între defecţiuni pentru o perioadă de şase luni).

Restricţiile se adresează adesea implementării, de aceea trebuie făcute cu mare grijă.

Ghidul de aplicare

- furnizează informaţii utile pentru implementare, în situaţia în care există mai multe strategii de implementare.

De exemplu: “ Timpul de răspuns al sistemului la cererile sosite de la tastatură trebuie să fie minimizat. “ Sau o altă alternativă: “Utilizarea memoriei interne trebuie să fie cât mai redusă. “

deficienţe:

- imprecizia informaţiei. De exemplu: “ Interfaţa va fi prietenoasă.”
- contradicţiile: “ Rezultatele vor fi memorate pe suport magnetic” , sau, “ Sistemul va răspunde în mai puţin de o secundă. “
- omisiunea sau incompletitudinea : tratarea datelor de intrare eronate, furnizate de utilizator.

În concluzie, realizarea cu succes a specificaţiilor este o activitate necesară şi care solicită implicarea cu competenţă a unui mare număr de persoane.

2.3. Concepte ale specificaţiilor de programare

Orice program se exprimă în două forme:

- una fiind o mulţime obişnuită de comenzi şi structuri de control caracteristice limbajului de programare utilizat, care exprimă CUM se rezolvă problema;
- a doua este o serie de propoziţii (secvenţe) într-un formalism matematic, care exprimă CE face programul.

Presupunând că una dintre aceste forme este o reprezentare satisfăcătoare a intenţiilor programatorului, dacă se poate demonstra că ele exprimă acelaşi lucru, sau au aceeaşi semnificaţie, atunci se poate concluziona că programul este corect. Această abordare a verificării programului nu implică nici un alt concept de corectitudine absolută, ci doar demonstrează consistenţa celor două descrieri ale aceleiaşi probleme.

Atunci când se solicită descrierea semnificaţiei unei piese de software, de regulă se descrie interacţiunea programului cu mediul utilizatorilor. Exprimarea semnificaţiei programului în termenii utilizatorilor prin descrierea execuţiilor sale posibile se numeşte **descriere operaţională.** Această descriere nu este unică, o altă cale fiind descrierea în termenii entităţilor unui limbaj de programare.

Utilizarea unui formalism pentru descrierea programului permite verificarea mai multor implementări ale aceleaşi probleme.

Exprimarea semnificaţiei programului (ale specificaţiilor sale) într-un formalism al calculului cu predicate de ordinul întâi, chiar dacă raţionamentul este încă în termenii entităţilor utilizate de limbajul de programare, este efectiv independentă de orice concept al execuţiei comenzilor pe un calculator real sau abstract. Acest concept este important deoarece permite ca, în principiu să se exprime semnificaţia programului într-o astfel de manieră încât consistenţa mai multor implementări să fie verificată.

Se ştie că una dintre cele mai importante surse de erori în programe este generată de absenţa unei porţiuni din proiectul logic, deci un program trebuie exprimat (descris) în aşa fel încât să se reducă această sursă de erori.

2.3.1.Variabilele şi stările unui program

Dacă examinăm un program scris în orice limbaj de programare putem identifica două aspecte: unul este **controlul** execuţiei instrucţiunilor (if-then, etc.), al doilea este **execuţia** comenzilor (ca asignări de valori rezultate în urma unor evaluări).

Ne putem astfel imagina că pentru fiecare variabilă a programului există o stare care conţine informaţia curentă la orice moment de timp. Această stare este identificată de valoarea variabilei care poate fi schimbată de execuţia unei comenzi.

Să presupunem că fiecărei variabile a unui program îi asociem una din axe într-un spaţiu cartezian multidimensional, care se va numi **spaţiul stărilor unui program.** Un punct în acest spaţiu este o stare a programului. Execuţia unei comenzi elementare este vizualizată în acest spaţiu ca un segment ce leagă starea programului înainte de execuţie de cea de după execuţie. În acest context, **spaţiul stărilor** este o mulţime a tuturor valorilor posibile ale variabilelor programului.

Expresia condiţiilor care trebuie satisfăcute, în spaţiul stărilor, pentru o execuţie corectă a programului se numeşte **precondiţie.** Specificaţiile stărilor la terminarea programului se numesc **postcondiţii.**

Operaţiile dintr-un program ne permit să reorganizăm diferitele reprezentări ale aceluiaşi concept. Un tip de date este o mulţime de valori şi operaţii definite pe ea însăşi.

Exemple:

" mai mare decât " : G x G $\rightarrow$ {true, false}

>

"adunare" : G x G $\rightarrow$ G

+

În acest context, un sistem este o colecţie de mecanisme care realizează unul sau mai multe tipuri de date.

Tipurile de date pot fi definite prin **enumerare**, prin **operaţii** sau prin **proceduri.** Depinde de limbajul de programare modul în care acestea sunt implementate în compilatoare.

În conformitate cu Dijkstra programatorii pot utiliza trei modalităţi de proiectare a programelor: enumerarea, inducţia matematică şi abstractizarea.

Instrumentul mental al enumerării este principala metodă folosită de programatori. Utilizăm gândirea enumerativă ori de câte ori vrem să instruim pe cineva despre execuţia mai multor activităţi.

A doua metodă indicată de Dijkstra, inducţia matematică, este foarte utilizată pentru proiectarea structurilor repetitive (cicluri). De remarcat că, inducţia matematică nu este un proces mental natural ca enumerarea.

În general, principiile inducţiei pot fi definite astfel:

1. **Baza inducţiei** este de a stabili că un anumit element ***p*** al unei mulţimi ***S*** satisface o relaţie ***R***.
2. **Pasul inducţiei** este de a arăta că un element generic ***y*** a lui ***S*** satisface ***R***, adică există ***T(y)***, unde ***T*** este orice fel de transformare în care alt element a lui ***S*** satisface ***R***.
3. Dacă baza şi pasul inducţiei sunt demonstrate, atunci orice element derivând din ***p*** printr-un număr de aplicaţii ale lui ***T***, de asemenea satisfac ***R***.

Principiul inducţiei este fundamentarea teoretică a metodei de verificare propusă de Floyd. Importanţa acestui concept de bază este aceea că nu cere o execuţie mentală a unei secvenţe de comandă, permiţând proiectarea şi verificarea ciclurilor care trebuie executate de mai multe ori, în funcţie de valori particulare ale datelor.

Atunci când inducţia matematică este aplicată asupra stărilor descrise de relaţia R, această metodă este un instrument efectiv pentru proiectarea secvenţelor de iteraţii în general şi executarea lor de ori câte ori.

În proiectarea programelor, aşa cum am amintit deja, se utilizează şi limbajul logicii predicatelor. Fiecare predicat defineşte un subset al spaţiului stărilor pentru care el este adevărat.

Precondiţia unui program este un predicat care defineşte un subset al stărilor acceptat ca intrare legitimă. Constanta predicat T, care este adevărată peste tot, defineşte toate stările posibile ale spaţiului stărilor. Constanta predicat F, care este falsă peste tot, nu defineşte nici o stare în spaţiul stărilor.

Cu alte cuvinte, dacă spaţiul stărilor reprezintă întregul univers al obiectelor care pot fi manipulate de program, T defineşte în întregime acest univers , iar F este mulţimea vidă. Dacă un program a fost specificat cu o precondiţie care acceptă orice stare din spaţiul stărilor, T este în mod clar predicatul care reprezintă corect această condiţie.

Predicatele definesc submulţimi ale spaţiului stărilor. În acest context, verificarea corectitudinii programului în totalitate poate fi definită.

Dându-se un program S şi o precondiţie r, vom găsi "cea mai slabă precondiţie", adică condiţia care defineşte numai şi numai acele stări iniţiale care asigură terminarea în stările care satisfac r.

Această precondiţie, "cea mai slabă" este un predicat funcţie atât de S cât şi de r, indicat prin ***wp(S,r)***.

Bineînţeles că se poate defini şi problema duală. Dându-se o precondiţie p a unui program S, vom găsi "cea mai puternică" postcondiţie, adică predicatul care defineşte numai şi numai acele stări pentru care programul va satisface terminarea când este iniţializat într-o stare care satisface p.

Această "cea mai puternică" postcondiţie este o funcţie de S şi p definită ca ***Sp(S,p)***.

Conceptele "cea mai slabă" precondiţie şi "cea mai puternică" postcondiţie au fost introduse de Dijkstra în 1976. Acestea au fost necesare pentru a descrie tehnica de proiectare a lui Dijkstra numită **calculul programării.**

Ideea de bază este de a utiliza tehnici de demonstrare a corectitudinii **nu** prin verificarea programului deja creat printr-un mijloc sau altul , ci prin a ghida pas cu pas proiectarea programului în mod explicit şi conştient.

Calculul Dijkstra este direct îndreptat spre realizarea unei tehnici care, la sfârşitul programelor, să furnizeze verificarea proiectării lor printr-o analiză "intelectuală" înainte de testarea produsului.

Pe scurt, calculul Dijkstra constă în construirea de expresii scrise în limbajul calculului predicatelor care să descrie precondiţiile şi postcondiţiile problemei şi care să se verifice ca adevărate în contextul problemei. Puterea acestui calcul este evidentă în cazul problemelor greu de algoritmat.

O tehnică destul de comună în programare este abstractizarea procedurală. Ea constă din abstractizarea unei piese din program (de regulă nescrisă încă) prin substituirea în textul său a precondiţiei şi postcondiţiei. În acest fel întreaga structură a programului poate fi definită şi se demonstrează corectitudinea lui prin utilizarea unui text mai scurt.

Această tehnică poate fi efectiv aplicată pentru algoritmi a căror complexitate şi dimensiune nu sunt prea mari. Nu este de imaginat să se aplice această metodă pentru proiectarea unui sistem software mare. Totuşi, este posibil de conceput că fiecare sistem mare este compus din mai multe părţi componente, fiecare dintre acestea nefiind prea mare, care poate fi proiectată folosind abstractizarea procedurală.

Problemele specifice limbajelor de programare, cum ar fi pointerii şi transmiterile de parametri între module, limitează aplicabilitatea calculului Dijkstra. Totuşi, exprimarea specificaţiilor programelor în termenii limbajului calculului predicativ ajută la reducerea erorilor.

2.4. Specificarea formală

Obiectivul acestui capitol este de a înţelege şi scrie specificaţiile unui sistem software. S-a amintit deja că ieşirile din activitatea de analiză sunt constituite din specificaţii de proiectare, care conţin atât solicitări funcţionale cât şi solicitări de date, cu alte cuvinte se detaliază cu precizie ceea ce sistemul va executa având în vedere restricţiile practice de care proiectantul va ţine cont. Ne vom referi la componenta funcţională a specificaţiilor de proiectare ca la o *specificaţie software* .

Comportarea unui sistem este influenţată de dezvoltarea lui în cele trei faze, şi anume: specificarea, implementarea şi verificarea. O abordare convenţională este specificarea sistemului în limbaj natural (în limba româna), scrierea programului într-un limbaj de programare (ex. Pascal) şi verificarea prin testare. S-a argumentat că o astfel de abordare este susceptibilă la tot felul de erori şi în ultimă instanţă generează software incorect.

Utilizând abordarea convenţională se pune întrebarea: cum şi de ce au fost scrise programe incorecte? Există trei cauze esenţiale:

1. Specificarea este greşită - ea poate fi ambiguă, vagă, inconsistentă şi/sau incompletă.

2. Programul este greşit - nu execută ceea ce există în specificaţii.

3. Verificarea este incompletă - testarea nu este exhaustivă, mai mult decât atât, nici nu poate fi exhaustivă .

Există următoarele aserţiuni pentru cele trei faze importante din dezvoltarea software-ului :

1). Specificarea este o *descriere* - adică spune CE reprezintă problema;

2). Problema este o *realizare* a specificaţiei - adică spune CUM poate fi rezolvată problema;

3). Verificarea este o *justificare* - spune DE CE realizarea satisface specificaţia.

Specificarea într-o notaţie formală, mai mult decât într-un limbaj natural, aduce beneficii imediate. "Formal" înseamnă scrierea în întregime într-un limbaj cu o sintaxă explicită şi precisă şi cu o semantică definită. Limbajul matematic este mai apropiat pentru acest scop.

Avantajele utilizării unei notaţii formale pot fi rezumate astfel:

- Specificaţiile formale pot fi studiate matematic - cu alte cuvinte, o specificaţie poate fi judecată utilizând tehnicile matematice. De exemplu, diverse forme de inconsistenţă sau incompletitudine în specificaţii pot fi detectate automat.
- Specificaţiile formale pot fi întreţinute cu mai multă uşurinţă decât dacă ar fi scrise în limbaj natural. Aceasta face ca specificaţiile să prezinte mai multă siguranţă şi să fie asigurat controlul posibilelor consecinţe ale schimbărilor.
- Notaţia utilizată pentru exprimarea specificaţiilor formale este extensibilă.

Utilizând un limbaj de specificare formală avem posibilitatea de a mecaniza transformarea specificaţiilor în programe. Prototipizarea sistemelor (în general ineficientă) poate fi generată automat din specificaţii mai adecvate pentru viabilitatea sistemului propus.

Şi mai important este că în acest fel, avem posibilitatea potenţială de a dovedi corectitudinea programului în raport cu specificaţiile sale.

2.5. Exemplu de specificare formală

În mod curent, cele mai cunoscute limbaje de specificare formală sunt Z şi VDM. Ambele utilizează o abordare pe bază de model, adică reprezintă tipurile de date şi structurile necesare pentru a putea descrie problema folosind entităţi cum ar fi: mulţimi, funcţii şi propoziţii.

Limbajul Z a fost prima dată introdus de Jean-Raymond Abriel în 1979 şi dezvoltat în cadrul Programming Research Group la Universitatea Oxford. Partea "puternică" a lui Z este aceea că specificaţiile structurate pot fi achiziţionate folosind "schema de calcul" ; aceasta permite specificaţiilor să fie construite în trei blocuri mai mici, iar programele pot fi proiectate top-down sau bottom-up, pentru componentele lor procedurale.

Metoda VDM (Vienna Development Method) a fost iniţiată în laboratoarele de cercetare IBM din Viena, în 1970. Această metodă se bazează pe notaţiile semantice, o abordare a definirii limbajelor de programare făcută de Scott şi Strachey. VDM reprezintă mai mult decât un limbaj semantic, el punând la dispoziţia celor care îl utilizează reguli şi proceduri care vor fi urmărite în diferite stadii de dezvoltare a sistemului.

Z şi VDM au notaţii vaste şi complexe. Este interesant poate de subliniat că un set relativ mic de instrumente permite specificarea cu uşurinţă a unei clase largi de probleme tipice.

Pentru a putea înţelege mai bine exemplul următor, să reluăm câteva notaţii matematice cum ar fi: mulţimile, secvenţele şi funcţiile.

O *mulţime* este o colecţie de obiecte, exprimată de obicei prin enumerarea elementelor incluse între acolade. De exemplu:

{Anton, Vasile, Ioana, Alice}

este o mulţime de nume. Se poate utiliza semnul "$\in$" pentru a indica apartenenţa unui element la o mulţime.

Astfel Anton $\in$ {Anton, Vasile, Ioana, Alice}, dar

Sandu $\notin$ {Anton, Vasile, Ioana, Alice}.

O *secvenţă* este o colecţie ordonată de obiecte exprimate în mod normal prin enumerarea elementelor sale, incluse în paranteze drepte. De exemplu:

[Anton, Vasile, Ioana, Alice]

este o secvenţă de nume. *Capătul (capul)* secvenţei este Anton, iar *corpul* este [Vasile, Ioana, Alice]. Dacă S este o secvenţă, atunci elementele lui S marchează mulţimea de elemente din secvenţa S. Dacă S este o secvenţă a mulţimii, atunci elemente lui S = {Anton, Vasile, Ioana, Alice}. O secvenţă poate avea elemente care se repetă, în timp ce o mulţime, nu.

O *funcţie* exprimă o relaţie între două elemente ale unei mulţimi. De exemplu, *director* poate asocia nume cu numere de telefon. Dacă vom utiliza *Nume* pentru a reprezenta mulţimea tuturor numelor posibile şi *Număr* pentru a reprezenta mulţimea tuturor numerelor de telefon posibile, atunci putem descrie *director* astfel:

director : Nume →Număr

Vom spune că *director* este o funcţie de tip *Nume* → *Număr* . Orice funcţie de acest tip va avea ca argument un membru din *Nume* şi va returna un membru din mulţimea *Număr.* Totuşi, *directorul* nostru particular este parţial, în sensul că este definit numai pentru unele elemente din *Nume* deoarece nu toate persoanele au telefon.

Submulţimea (subsetul) pe care este definită funcţia se numeşte *domeniul de definiţie al funcţiei.* De exemplu, să presupunem că *Nume* este mulţimea {Anton, Vasile, Ioana, Alice} şi *Număr* este mulţimea {421563, 123456}. În acest caz *directorul* va fi definit explicit astfel:

director(Anton) = 421563

director(Alice) = 123456

Dacă Vasile şi Ioana nu au telefon atunci *director (Vasile)* şi *director (Ioana)* nu sunt definite. Domeniul funcţiei este mulţimea {Anton, Alice}, care este un subset al mulţimii *Nume* .

Funcţia *director* mai poate fi definită prin explicitarea enumerativă a *Numelui* legat de un *Număr,* astfel:

director = {Anton $\rightarrow$ 421563, Alice $\rightarrow$ 123456}

Anton $\rightarrow$ 421563 şi Alice $\rightarrow$ 123456 se numesc "corespondenţe" (maplets). *Director* este o funcţie finită deoarece domeniul său este finit, cu alte cuvinte conţine un număr finit de corespondenţe.

Funcţiile exprimă mai mult decât o relaţie. Astfel, dacă luăm ca model funcţia telefon, mai multe persoane pot avea acelaşi număr de telefon, dar o anumită persoană nu poate avea decât un singur număr de telefon.

Să considerăm acum următorul exemplu pentru a descrie specificaţiile formale simple pentru o problemă descrisă informal astfel :

O bancă are numai un ghişeu unde în mod normal clienţii aşteaptă la coadă. Fiecare client se identifică prin numărul de cont şi sunt permise numai tranzacţii de adăugare sau scoatere din cont. Clientului i se refuză eliberarea sumei cerute dacă contul său este vid. Dacă el doreşte mai mult decât are în cont atunci va primi numai suma pe care o mai are în cont. După ce s-a efectuat tranzacţia, clientul părăseşte banca.

Primul lucru care trebuie făcut este să se stabilească un model matematic adecvat pentru sistemul propus. Conturile din bancă pot fi modelate printr-o funcţie parţială care face corespondenţa între numerele de cont şi balanţa băncii:

bancă : Cont $\rightarrow$ *Balanţă*

Cont este mulţimea de numere de conturi posibile, iar *Balanţă* este mulţimea tuturor balanţelor (disponibilului) băncii. Deci o funcţie *bancă* poate arăta astfel :

{a1$\rightarrow$ 23, a2 $\rightarrow$ 87, a3 $\rightarrow$ 45},

care indică de exemplu, că persoana care are numărul de cont a1 are 45000 lei în cont, ş.a.m.d. Domeniul funcţiei *bancă* (scris dom_bancă) este o submulţime {a1,a2,a3}, care identifică conturile.

Coada la ghişeul băncii poate fi modelată ca o secvenţă de numere de cont astfel:

coadă : secvCont

În acest fel [a2,a5,a1] reprezintă o coadă tipică. Vom adopta convenţia că prima persoană din linie reprezintă *capul* cozii. De notat că putem avea elemente care se repetă, adică o persoană poate să apară de mai multe ori în coadă. Ne vom asigura că acest lucru nu este posibil, introducând restricţii suplimentare ori de câte ori se schimbă sau actualizează coada.

Avem acum specificarea formală a operaţiilor care se vor efectua în sistemul modelat. Fiecare operaţie va fi specificată prin trei componente:

1. Intrările operaţiei;

2. Condiţia în care operaţia poate fi aplicată (*precondiţia*);

3. Expresia care arată relaţia dintre starea sistemului înainte de operaţie şi starea sistemului după operaţie (*post-condiţia*).

Vom adopta convenţia de a folosi numele obişnuit pentru a desemna starea iniţială şi &nume, pentru starea finală. Exemplu : coadă reprezintă starea cozii înainte de sosirea unui client, iar &coadă, după.

Să luăm acum specificaţia care introduce un client în coadă:

sosire (client : Cont)

precondiţie

client $\in$ *dom_bancă*

client $\notin$ *elem_coadă*

post-condiţie

&coadă = adaug_ coadă [client]

Numele operaţiei este *sosire.* Ea are o intrare numită *client*, de tip *Cont.* Precondiţia indică faptul că atunci când un client soseşte la coadă el trebuie să aibă cont în bancă, altfel nu va fi acceptat în coada de aşteptare. Post-condiţia indică faptul că după operaţie coada se măreşte cu o persoană. (Observaţie: *adaug* este un operator de concatenare a două secvenţe).

De exemplu, să presupunem:

bancă = {a1→ 23, a2→ 87, a3 → 45}

coadă = [a2]

client = [a1]

atunci

client ∈ {a1,a2,a3}

client ∉ {a2}

şi

&*coadă* = [a2,a1]

Vom scrie acum o operaţie care acordă credit unui client care are cont şi se află în capul cozii (este primul în coadă):

credit (*suma* : *Balanţă*)

precondiţia

coadă ≠ [ ]

post-condiţia

&bancă = *bancă* ⊕ {*cap coadă* → *bancă(cap coadă*) + *suma*}

&coadă = *elimin coadă*

Operaţia *credit* are o intrare numită ***sumă***, de tip ***Balanţă*** şi o precondiţie care precizează că lista (coada) nu trebuie să fie vidă, adică trebuie să fie măcar o persoană la coadă.

⊕ este un operator de scriere adiţională, peste ceea ce era înainte. El creează o nouă funcţie din alte două date ca operanzi. De exemplu, să presupunem că:

bancă = {a1 → 23, a2 → 87, a3 → 45}

coadă = [a2,a1]

suma = 10

Atunci

capul cozii = a2

bancă(*capul cozii*) = 87

astfel încât

&bancă = {a1 → 23, a2 → 87, a3 → 45} ⊕ {a2 → 87+10} = {a1 → 23, a2 →97, a3 →45}

Corespondenţa care are a2 în stânga a fost scrisă peste vechea corespondenţă, definind astfel o nouă funcţie care modelează noua stare a băncii.

Ultima componentă a post-condiţiei arată că după efectuarea tranzacţiei, clientul din faţă părăseşte coada :

&*coadă* = *elimin* [a2 a1]

= [a1]

În final, vom defini operaţia care permite efectuarea retragerii de bani din cont :

debit (*suma* : *Balanţă*)

precondiţie

coadă $\neq$ [ ]

bancă (*capul cozii*) $\geq$ *suma*

post-condiţia

&*bancă* = *bancă* $\oplus$ { *capul cozii* $\rightarrow$ *bancă*(*capul cozii*) - *suma*}

&*coadă* = *elimin coadă*

Prima componentă din precondiţie arată mai întâi că nu trebuie să fie vidă coada şi apoi presupune că:

bancă = {a1 $\rightarrow$ 23, a2 $\rightarrow$ 87, a3 $\rightarrow$ 45}

coadă = [a2,a1]

suma = 10

Capul cozii este a2 iar *bancă* (*capul cozii*) = 87

A doua componentă a precondiţiei specifică faptul că balanţa relativă la contul clientului trebuie să fie mai mare sau egală cu suma pe care acesta intenţionează s-o scoată din bancă.

În contextul validării specificaţiilor (ceea ce înseamnă de fapt înglobarea cerinţelor), vom specifica acum un *invariant* .

Un *invariant* este o aserţiune despre componentele modelului matematic, pe care se vor baza apoi specificaţiile noastre. Dacă invariantul se află înainte de operaţiile pe care le-am specificat, atunci el trebuie să apară şi după ce operaţia este completă.

El exprimă ceva ce este întotdeauna adevărat. De exemplu, există un invariant pentru bancă:

($\forall$ a $\in$ dom_bancă) *bancă*(a) $\geq$ 0

elem_coadă $\subseteq$ dom_bancă

coadă $\leq$ 10

Acest invariant stipulează că :

1. Oricare ar fi *a* din domeniul băncii, valoarea lui *bancă*(a) este mai mare sau egală cu zero. Aceasta înseamnă că toţi clienţii băncii au credit.

2. Mulţimea de persoane aflată în coadă este o submulţime a domeniului băncii. Aceasta înseamnă că fiecare persoană din coadă are un cont în bancă.

3. Numărul de elemente din coadă este mai mic sau egal cu 10, considerând că nu vor fi niciodată mai mult de 10 persoane la coadă.

Intuitiv, se poate vedea că debitul şi creditul conservă acest invariant, dar *sosire* nu, de aceea nu a existat nici o restricţie cu privire la lungimea cozii. Totuşi *sosire* va conserva invariantul dacă vom adăuga un extra predicat : # *coadă* < 10

în precondiţia sa.

Invarianţii furnizează un instrument de valoare pentru asigurarea consistenţei peste operaţiile care se fac în cadrul specificaţiilor.

Concluzii:

Rezumând, un mediu ideal de inginerie software (cu sublinierea cuvântului inginerie), ca scenariu tipic pentru dezvoltarea unui program, trebuie să arate astfel:

- Să existe o specificare formală, derivată dintr-o descriere informală, care să ajute mai bine la înţelegerea problemei.
- Să existe o dovadă care să demonstreze că specificarea reală se poate realiza printr-un algoritm.
- Să existe un prototip (chiar dacă este ineficient) care să poată fi prezentat celui care trebuie să valideze proiectul.
- Să se scrie programul sau programele derivate din specificaţii.
- Să existe o dovadă că programul este corect prin prisma specificaţiilor sale.

Dezvoltarea unor instrumente integrate de software care să susţină fiecare dintre aceste faze este considerată vitală pentru acceptarea unui astfel de scenariu.

2.6. Exerciţii

1) Care sunt informaţiile necesare care trebuie colectate şi înregistrate ca specificaţii software?

2) Explicaţi dificultăţile limbajului natural pentru descrierea specificaţiilor software?

3) Luaţi ca exemplu un sistem informatic mic (care doriţi dv.). Identificaţi componentele funcţionale şi de date din sistem. Identificaţi problemele legate de specificaţiile software cum ar fi: ambiguitate, inconsistenţă şi informaţii vagi.

4) Există solicitarea pentru un sistem informatic care să gestioneze informaţiile despre cărţile existente într-o mică bibliotecă de departament.

s1 - sistemul trebuie să funcţioneze pe un calculator standard PC;

s2 - pentru fiecare carte informaţiile standard sunt:

- titlul
- autorul
- codul de clasificare (cota cărţii)
- anul apariţiei
- dacă este împrumutată
- data solicitării

s3 - calculatorul trebuie să memoreze informaţiile pentru 1000 de cărţi

s4 - sistemul trebuie să răspundă la următoarele comenzi de la tastatură:

(a) solicitarea de împrumut a unei cărţi

(b) înapoierea unei cărţi împrumutate

(c) crearea unei înregistrări pentru o carte nou achiziţionată;

s5 - comenzile vor fi accesibile printr-o selecţie cu ajutorul cursorului dintr-un meniu;

s6 - calculatorul trebuie să răspundă în maximum 30 de secunde de la formularea cererii;

s7 - calculatorul trebuie să fie capabil să tipărească întregul catalog al cărţilor, cu un cap de tabel corespunzător. Acest lucru trebuie să se facă în paralel cu altă comandă primită;

s8 - măsuri de securitate: sistemul va iniţializa informaţiile din bibliotecă numai dacă el conţine zero cărţi;

s9 - când o carte este ştearsă sistemul va afişa informaţiile despre ea;

s10 - sistemul trebuie livrat la data D, trebuie să nu depăşească costul C, să fie în întregime documentat şi uşor de întreţinut.

5. " Scrieţi un program de sortare ". Este această modalitate, adecvată ca specificaţie ?

6. Un aspect necesar al specificaţiilor este discuţia cu potenţialii utilizatori. Cei mai mulţi dintre ei nu înţeleg notaţiile matematice. Este acesta un dezavantaj? Ce aduc în plus notaţiile formale pentru a face această muncă mai productivă?

7. Enumeraţi câteva cerinţe non-funcţionale din implementarea programului de împrumut la bancă.

8. Scrieţi specificaţiile pentru deschiderea şi închiderea unui cont bancar.

9. Modificaţi invariantul pentru bancă astfel încât să ilustreze faptul că o persoană nu poate să apară de mai multe ori în coadă.

10. Pentru cine sunt mai importante atributele unei specificaţii, pentru programator sau pentru cel care implementează programul ?