Analiza cerinţelor software. Introducere

Această serie de articole este destinată tuturor persoanelor implicate în proiecte de dezvoltare de software: şefi de departament, şefi de proiect, analişti (în primul rând lor), arhitecţi software, programatori sau specialişti în asigurarea calităţii. Un prim argument pentru a fi citite (dar şi pentru a fi scrise :-)) este acela că, în universităţile occidentale aceste lucruri sunt predate tuturor studenţilor care se pregătesc pentru o carieră în IT. Cartea conţine, inevitabil, numeroase elemente de Inginerie software, utile oricărui specialist IT: despre ciclul de dezvoltare în proiectele software, despre interacţiunile dintre disciplinele implicate în proiect etc.

Un al doilea argument este acela că Analiza software este una dintre disciplinele cu impact foarte mare asupra succesului sau insuccesului proiectului software. Statisticile arată peste 60-80% din costurile non-calităţii sunt cauzate de înţelegerea, culegerea şi managementul inadecvate ale cerinţelor. Conform Software Engineering Institute (SEI), primii doi factori care contribuie la eşecul proiectelor, în privinţa respectării bugetelor şi termenelor sunt: specificaţiile inadecvate ale cerinţelor şi schimbările necontrolate ale acestora. Aşa cum vom vedea mai departe, de aceste lucruri se ocupă, în cea mai mare măsură, Analiza.

Argumente

În România, deşi domeniul IT este recunoscut ca fiind unul dintre cele mai dinamice din economie, şi ne place să credem că reprezentăm o forţă pe piaţa de IT, singura disciplină în care excelăm este programarea. Ori programarea nu reprezintă decât maximum 30 – 50% din efortul necesar într-un proiect software.Bunele practici recomandă ca efortul consumat pentru Analiză să fie între 10 şi 30% din efortul total consumat într-un proiect. Pentru aceasta, la nivel de ţară, este desigur nevoie de foarte mulţi oameni specializaţi în domeniul Analizei, este nevoie de cursuri de specialitate, de cărţi sau de articole în revistele de specialitate. Cu toate acestea, în România cărţile despre Inginerie Software, Analiză, Project Management sau Asigurarea calităţii în software se pot număra pe degete, în unele dintre domenii nefiind publicată chiar nici-o carte. Cursuri, despre aceste domenii în facultăţile de specialitate, nu există deloc.

Spre comparaţie, în Statele Unite, lucrurile scrise în această carte se studiază la facultate. Americanii au chiar un standard care conţine recomandări privind specificaţiile software: IEEE Std 830-1993: IEEE Recommended Practice for Software Requirements Specifications.

În 2005, Center for National Software Studies (www.cnsoftware.org) a lansat un raport, în urma celui de-al doilea summit naţional pe probleme legate de domeniul software, denumit „Software 2015: A National Software Strategy to Ensure U.S. Security and Competitiveness”, care arată rolul major al software-ului şi al inovaţiei software asupra succesului economiei americane: „Software-ul şi inovaţia software au o importanţă mare în succesul producţiei de automobile, telecomunicaţii şi de asemenea, asupra tuturor celorlalte industrii. Din perspectiva securităţii naţionale, sistemele militare [...] sunt toate dependente de software-ul inovativ, de înaltă calitate.”trategia propusă prevede sarcini concrete precum: motivarea tinerilor pentru dezvoltarea carierei în software, educarea în domeniul Ingineriei Software, identificarea priorităţilor cercetării şi dezvoltării, creşterea câştigurilor financiare pentru performanţa în inovare.

Vorbind de România, această situaţie poate fi o oportunitate: suntem în faţa unui teren liber, pe care avem ocazia de a construi aşa cum trebuie.

Prin urmare, aceaste articole au fos scrise, în primul rând, pentru că în România există un gol imens în acest domeniu. Au fost scrise cu gândul ca oricine este interesat de domeniu să poată afla de aici:

- ce este Analiza, ce sunt cerinţele, ce sunt Use Case-urile, specificaţiile şi care este rostul existenţei lor;
- care sunt principiile care stau la baza acestei discipline şi cum se lucrează efectiv în Analiză;
- de ce dau companiile bani pe software, ce se aşteaptă să primească şi ce se întâmplă să primească uneori.

Cel mai important lucru din acest articol

Aşa cum ne arată statisticile principala cauză a eşecului proiectelor software este insuficienta implicare a clienţilor. Buna înţelegerea a problemelor clientului, neapărat legată de împărtăşirea unei viziuni comune între echipa de dezvoltare şi client sunt două faţete de bază ale implicării clientului. Fără ele, echipa de dezvoltare nu are referinţele de bază pentru a putea şti dacă ceea ce dezvoltă este corespunzător sau nu şi, prin urmare, implicarea puternică a clientul în proiect şi înţelegerea comună a problemelor lui este vitală.

Altfel, este ca şi cum te-ai deplasa într-o direcţie oarecare, fără să ştii unde vrei să ajungi.

Mesajul acesta este adesea (şi cel mai bine!) ilustrat printr-o caricatură despre diferenţa care apare adesea în practică, între problema clientului, înţelegerea problemei şi ceea ce se realizează de fapt (vezi imaginea din dreapta!)

Despre Analiza cerinţelor

Ce este Analiza cerinţelor software

Analiza cerinţelor software (pe care o vom numi în continuare Analiza Software) este aceea dintre disciplinele existente în domeniul Software care determină ce trebuie făcut, preluând problema clientului şi exprimând-o într-un inteligibil de către dezvoltator.

Mai detaliat, Analiza Software este aceea dintre disciplinele implicate în procesul de dezvoltare a produsului software ale cărei obiective sunt:

- înţelegerea problemelor curente ale organizaţiei client şi a raţiunii pentru care este dispus să investească într-un produs software;
- asigurarea unei viziuni comune asupra problemelor şi a viitoarei soluţii pentru toţi participanţii în proiect;
- culegerea şi actualizarea cerinţelor pentru viitorul produs software şi traducerea cerinţelor clientului în cerinţe software pe care echipa de dezvoltare să le poată înţelege şi transforma în funcţionalităţi efective;
- definirea limitelor viitorului sistem (acesta este un element extrem de important în economia proiectului, având în vedere că în proiectele reale există întotdeauna o presiune crescândă în direcţia extinderii acestor limite).

Activitatea de Analiză este, în acelaşi timp, aceea care furnizează elementele pentru negocierea şi renegocierea bugetului şi pentru planificarea resurselor. Ea conturează produsul şi, prin urmare, proiectul. Impactul Analizei asupra costurilor proiectului şi, în general, asupra succesului acestuia este, aşa cum am prezentat deja, foarte mare.

Pentru a fi foarte concret, în activitatea unui Analist Software intră următoarele mari tipuri de activităţi:

- interviuri la client;
- analiza propriu-zisă a cerinţelor;
- dezvoltare cerinţe (specificaţie);
- managementul cerinţelor.

Toate aceste elemente, care umplu orele de serviciu (şi pe cele suplimentare) ale Analistului vor fi detaliate în paginile următoare.

De asemenea, trebuie precizat şi ce nu înseamnă Analiza Software. Analiza Software nu însemnă UML (Unified Modelling Language) sau Use Case-uri, aşa cum se mai întâmplă să se creadă. De asemenea, nu trebuie confundată şi nici amestecată Analiza Software cu Design-ul, Analiza având ca preocupare de bază găsirea PROBLEMEI iar Design-ul având ca preocupare de bază găsirea SOLUŢIEI.

Analiza este datoare să determine cu precizie boala, cea către care trebuie să se îndrepte tratamentul, fiind de la sine înţeles că este profund greşit să tratezi, din superficialitate sau necunoaştere, simptomele, ignorând boala în sine.

De ce este nevoie de Analiză Software?

The Standish Group arăta că primele trei cauze ale problemelor privind calitatea şi livrarea la timp, în proiectele software, sunt:

- insuficient input (contribuţie) de la utilizatori;
- cerinţe şi specificaţii incomplete;
- schimbarea frecventă a cerinţelor.

Ori, toate acestea sunt lucruri de care se ocupă, în primul rând, Analiza Software. De asemenea, trebuie să fim conştienţi că schimbarea frecventă a cerinţelor, de exemplu, nu este lucru accidental, care într-un proiecte se întâmplă, în altul nu – dimpotrivă, este un lucru care se întâmplă întotdeauna. Prin urmare, nici nu se pune problema ca într-un proiect software să nu îţi propui să controlezi un factor atât de important.

De asemenea, Analiza, prin output-urile sale (adică Specificaţiile software) răspunde la una dintre nevoile fundamentale ale proiectelor: nevoia de comunicare a cerinţelor între membrii echipei de proiect şi de formalizare a acestora.

Despre modul în care Analiza influenţează celelalte discipline şi activităţi implicate în proiect găsiţi detalii într-unul din articolele viitoare.

Axiome ale dezvoltării de software

Cea mai bună cale pentru a-ţi îndeplini visele este să te trezeşti.
Paul Valery

În continuarea răspunsului la întrebarea „De ce este nevoie de Analiză Software?" am să vă prezint o serie de lucruri pe care practica le susţine ca fiind, fără dubiu, nişte adevăruri şi care, aşa cum vom vedea susţin necesitatea existenţei Analizei software.

Axiomele dezvoltării de software:

A1. Întotdeauna cerinţele se schimbă pe parcursul derulării proiectului. Întotdeauna clientul cere mai mult decât la început şi tinde să extindă proiectul peste bugetul iniţial. **Clientul nu ştie cu precizie ce vrea şi este înclinat să îşi modifice cerinţele. Pentru a îşi clarifica propriile gânduri aşteaptă „să vadă mai întâi aplicaţia".**

A2. Întotdeauna, într-un proiect software apar situaţii neprevăzute. Situaţiile neprevăzute nu sunt o abatere de la regulă ci sunt chiar regula.

A3. Niciodată oamenii implicaţi în proiect nu sunt perfecţi. Toţi fac greşeli: programatorii produc bug-uri, analiştii erori de analiză iar project managerii, erori de management.

Cea mai mare parte dintre bug-urile dintr-un produs software de dimensiuni mari (unii spun că peste 70%) sunt introduce în fazele de analiză şi design. Cu cât un bug există pentru mai mult timp într-o aplicaţie, cu atât este mai costisitoare detectarea lui şi rezolvarea va fi mai puţin corespunzătoare.

A4. De regulă, clientul nu citeşte specificaţiile software sau le citeşte superficial. Mai mult decât atât, feed-back-ul primit de la client în faza de dezvoltare a proiectului este insuficient şi incomparabil mai puţin consistent decât feed-back-ul primit după depăşirea termenului final al proiectului.

A5. Nici un proiect software nu dispune de un buget nelimitat. Toate proiectele software au bugete insuficiente.

Dacă acum, aceste axiome, nu sunt suficient de relevante, ele ne vor folosi pe parcursul întregii cărţi pentru a înţelege mai mult.

În continuare (partea a IV-a a seriei), pentru a porni discuţia privind locul exact al Analizei într-un proiect software, voi descrie ciclul de dezvoltare al produsului software şi vom vedea cum se integrează Analiza cu restul disciplinelor implicate într-un proiect.

Ciclul de dezvoltare al produsului software (SDLC)

Deşi în limba engleză este denumit Software Development Life Cycle (SDLC) am ales traducerea „Ciclul de dezvoltare al produsului software”, chiar dacă tendinţa întâlnită cel mai adesea este să se traducă prin „Ciclul de viaţă al produsului software”. (Nu mai vorbim că traducerea mot a mot ar fi şi ea diferită.)

Rămâne la latitudinea dumneavoastră să alegeţi traducerea preferată. Noi, în aceste articole, vom folosi denumirea de „Ciclul de dezvoltare al produsului software” dar îl vom prescurta aşa cum face mai toată lumea: SDLC.

Ciclul de dezvoltare al produsului software este un model al apariţiei produsului software, pornind de la problema originară şi ajungând la un produs concret, care să răspundă acelei probleme.

Cu siguranţă, acest model nu îşi arată utilitatea în condiţiile unor proiecte mici, cu un programator răspunzând la cerinţele ad-hoc ale unui client, şi asta în termen de câteva zile sau săptămâni.

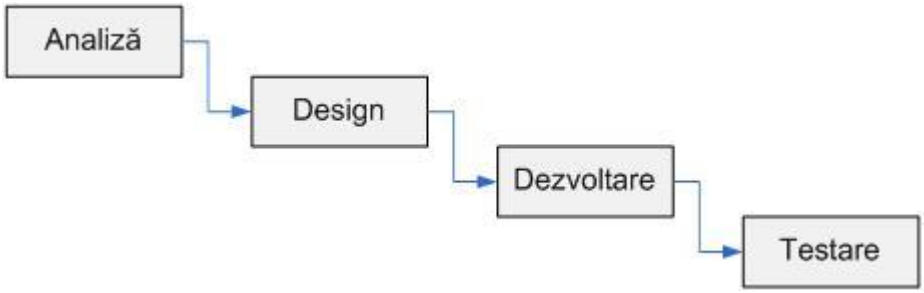
Însă, în condiţiile în care în proiectele software din zilele noastre sunt implicate echipe mari de dezvoltatori, analişti, arhitecţi software, designeri, manageri, în multe cazuri distribuiţi în ţări sau pe continente diferite, pe perioade de timp de ordinul lunilor sau anilor, acest model teoretic începe să aibă utilitate.

El este un prim pas către separarea a ceea ce este în interiorul proiectului, din punct de vedere al tipurilor de activităţi şi din punct de vedere temporal. Este, prin urmare, un prim pas către acel „divide et impera” care, aşa cum vom vedea, permite păstrarea controlului asupra proiectului.

În literatura de specialitate sunt descrise mai multe Cicluri de dezvoltare software. Dintre acestea, două au importanţă asupra a ceea ce dorim să discutăm în această carte şi, din acest motiv, le vom prezenta numai pe acestea două. De asemenea, nu voi intra în toate detaliile acestor cicluri ci voi prezenta numai ceea ce consider a fi util din punct de vedere al Analizei Software.

Ciclul de dezvoltare în cascadă (în V)

Cel mai vechi model şi cel mai cunoscut este ciclul în cascadă (waterfall). El este o secvenţă de stadii în care finalul unui stadiu este începutul altuia. Aceste stadii sunt (de notat că în unele cărţi apar mai multe, fiind incluse studiul de fezabilitate, integrarea sau instalarea, stadii care pentru lucrarea de faţă sunt mai puţin relevante şi pe care nu le vom prezenta):



1. Analiză

În această etapă a proiectului are loc definirea a ceea ce trebuie dezvoltat. Obiectivul aici este să se afle nevoile clientului şi să se definească foarte clar cerinţele privind viitorul produc software. Cartea de faţă este despre această fază.

2. Design

Această etapă are ca obiectiv modelarea viitorului sistem, văzut ca soluţie a problemelor determinate în faza de analiză. Dacă Analiza îşi propunea să determine ce trebuie făcut, faza de Design trebuie să arate cum trebuie făcut.

În această fază sunt proiectate funcţionalităţile pe care viitorul sistem va trebui să le aibă.

3. Dezvoltare

Această fază înseamnă scrierea efectivă a codului, dezvoltarea efectivă a acestuia.

4. Testare

În această fază produsul este testat pentru descoperirea anomaliilor în funcţionare, astfel încât la final să corespundă cerinţelor definite în faza de Analiză.

În afară de aceste faze esenţiale ale procesului de dezvoltare, mai trebuie totuşi menţionate şi deploy-mentul, acceptanţa şi mentenanţa. Acceptanţa este faza (sau momentul) în care clientul recepţionează sistemul software, acceptă că acesta corespunde cerinţelor lui şi îşi dă acordul pentru intrarea în faza de mentenanţă. Intrarea în faza de mentenanţă înseamnă încetarea includerii oricăror noi cerinţe în sistem şi corectarea bug-urilor (anomaliilor în funcţionare). Această fază este importantă pentru că ea constituie adesea o fază costisitoare, dar şi prea adesea ignorată.

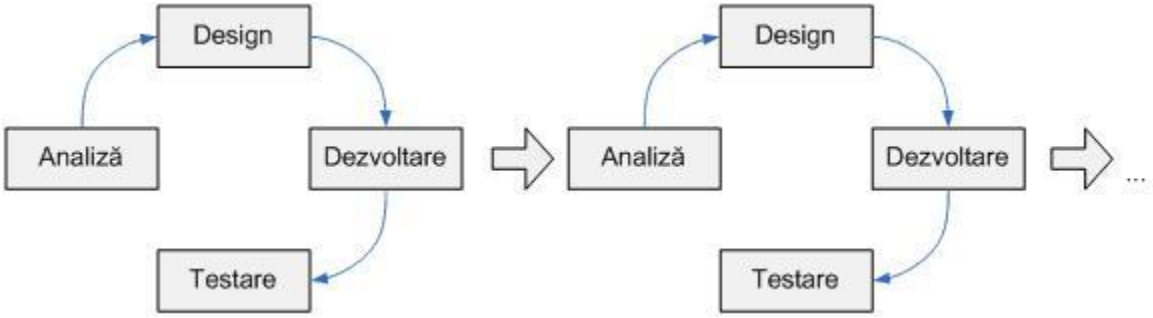
Modelul de mai sus este, cu siguranţă, un model mai degrabă teoretic, el bazându-se pe presupunerea (evident falsă) că cerinţele definite în prima fază nu se vor schimba până la sfârşitul proiectului. În realitate schimbarea cerinţelor este singurul lucru stabil în software. Întotdeauna, în decursul proiectului, cerinţele se vor schimba (axioma A1 – vezi capitolul Axiome ale dezvoltării de software).

Marele rezultat al existenţei acestui model este evident de ordin teoretic. El ne dă imaginea clară a faptului că anumite activităţi nu pot fi executate decât după finalizarea altora, de altă natură – sunt dependente de ele.

Ciclul de dezvoltare iterativ (în spirală)

Asumarea realităţii că întotdeauna, în decursul proiectului, cerinţele se vor schimba a condus la apariţia unui model de dezvoltare realist, care să fie adaptat acestei realităţi şi care să permită înglobarea schimbării în cel mai bun mod cu putinţă.

Ciclul de dezvoltare iterativ este o succesiune de cicluri în cascadă (waterfall), fiecare iteraţie producând o parte din întregul produs, parte care reprezintă intrarea pentru următoarea iteraţie:



Dacă la ciclul în cascadă disciplinele implicate în proiect (Analiză, Design, Implementare, Testare) se confundau cu fazele de proiect, în ciclul iterativ acestea nu mai pot fi privite ca faze ci ca ceea ce sunt ele de fapt: discipline implicate mai mult sau mai puţin în fiecare iteraţie din proiect.

În acest ciclu, fiecare nouă iteraţie înseamnă detalierea (sau clarificarea), adăugarea, modificarea eventual eliminarea unor elemente definite în iteraţiile anterioare. De asemenea, fiecare iteraţie permite şi validarea a ceea ce s-a făcut până în acel moment.

Acest ciclu mai are şi avantajul că permite implicarea clientului chiar şi în fazele avansate de dezvoltare a produsului, astfel încât să se obţină un preţios feed-back. Acest feed-back va putea fi integrat în produs în iteraţiile următoare.

Locul Analizei în proiectul de dezvoltare software

Povestea unui Proiect software

Modelele din articolele precedente sunt nişte modele călăuzitoare pentru lumea reală. Aşa cum rezultă de aici, Analiza Software este o disciplină care se află în relaţie de dependenţă cu celelalte discipline din proiect. Concret, task-urile din proiectul software sunt condiţionate unele de altele, iar task-urile specifice analizei sunt task-uri fără de care ciclurile din cadrul iteraţiilor nu pot avea loc.

Să luăm ca exemplu într-un proiect, pe care îl vom numi în continuare proiectul ALPHA, cu o echipă mare, cu un termen de livrare de cel puţin un an, cu o echipă de dezvoltare de 20 de persoane şi cu un client adesea indecis, care îşi schimbă cerinţele de la o lună la alta (acestea nu sunt condiţii excepţionale, nemaiîntâlnite într-un proiect software, ci dimpotrivă, sunt chiar condiţiile reale cel mai des întâlnite în practică). Proiectul porneşte cu mult entuziasm şi, în ciuda unor experienţe trecute nu prea plăcute, speranţele sunt mari. Şeful de proiect este (ce şansă!) un tip simpatic ales dintre programatorii cei mai experimentaţi un ins serios şi plin de calităţi iar echipa de programatori este o echipă cu experienţă, sudată în proiecte grele din trecut.

Proiectul se startează efectiv printr-o întâlnire faţă în faţă între cei mai valoroşi membrii ai echipei de dezvoltare şi şeful de proiect din partea clientului, însoţit şi el, bineînţeles, de câţiva dintre oamenii care au responsabilităţi legate de proiect. După manual întâlnirea se numeşte kick-off meeting şi trebuie făcută „ca să avem un prim contact cu ei”. Întâlnirea decurge bine pentru că echipa noastră de proiect ştie precis ce trebuie să întrebe şi discuţia este fructuoasă: discutăm despre cum să arate ecranul de introducere a datelor, cam care sunt rapoartele de care au ei nevoie, le explicăm că funcţionalitatea X nu se poate face chiar aşa cum credeau dar putem rezolva altfel, în fine... colaborarea este OK şi asta contează.

Se iau şi notiţe şi urmează ca pe baza lor să începem lucrul la specificaţie. Cu cât începem lucrul la specificaţie mai devreme, că să obţinem semnătura clientului pe ea, cu atât mai bine.

Spre finalul discuţiei o doamnă de la contabilitate îşi aminteşte că ea vrea neapărat un buton cu care sistemul să facă automat importul facturilor din Excel. I se explică rapid că această funcţionalitate necesită nişte dezvoltări care nu erau prevăzute dar asta o nemulţumeşte profund: se subînţelege că sistemul trebuie să facă asta pentru că altfel este inacceptabil. Ea nu are nevoie de acest sistem dacă nu face atâta lucru şi că ea nu este dispusă să bage de mână facturile în sistem. Abil, şeful nostru îi promite că va căuta o soluţie şi micul conflict se aplanează.

Întoarsă acasă, echipa de dezvoltare începe organizarea. Este numit un responsabil cu scrierea „specificaţiei” care trebuie terminată în două săptămâni că să scăpăm de treaba asta şi să ne apucăm de lucru. Scrierea specificaţiei va fi bineînţeles supervizată de şeful de proiect, cel mai în măsură să facă asta (chiar ar face el specificaţia dar nu are timp).

În fine, după alte trei şedinţe cu clientul şi după o lună de efort, „specificaţia” este gata, semnată şi parafată. Echipa poate începe să lucreze cu adevărat. Se împart task-urile pentru programatori: tu te ocupi de capitolul 1 din specificaţie, tu de capitolul 2 şi aşa mai departe. Dacă sunt neclarităţi programatorii sunt liberi să întrebe. Se fac şedinţe săptămânale în care se găsesc soluţii tehnice la diversele probleme. Din când în când şeful de proiect mai cere lămuriri la client în legătură cu o problemă sau alta. Dacă unele *"lămuriri"* se bat cap în cap cu specificaţia, se planifică o întâlnire la client şi se ia o decizie, de comun acord. Proiectul merge bine, în direcţia aşteptată chiar dacă apar din când în când bug-uri sau probleme de integrare (este interesant că aproape toată lumea poate face astfel de evaluări, chiar dacă *"direcţia aşteptată"* este definită vag şi, de obicei, se rezumă la evoluţia cantităţii de cod scris).

După opt luni de la începutul proiectului, echipa actuală începe să realizeze că există unele întârzieri. Pentru anumite module se pune problema că programatorul care a început treaba a plecat din echipă, iar cel care i-a luat locul nu înţelege prea bine ce are de făcut şi nici de ce s-a făcut într-un fel sau altul.

Şeful de proiect s-a săturat să tot pună întrebări la client aşa că răspunde la problemele ridicate de programatori cu nişte presupuneri logice, care, „simte” el că nu au cum să nu fie acceptate. Oricum, şi în echipa de proiect a clientului s-au făcut modificări şi nimeni nu mai ştie ce s-a cerut cu câteva luni în urmă.

Actualizarea specificaţiilor a fost abandonată demult pentru că nu mai era timp pentru aşa ceva şi oricum s-a dovedit o activitate inutilă.

După unsprezece luni şi jumătate este clar că proiectul e în întârziere. Deşi s-a dezvoltat aproape tot, mai sunt câteva probleme importante care nu au fost încă abordate din lipsă de timp, pentru care nu există o soluţie sau soluţia dată nu este acceptată de client şi trebuie făcute dezvoltări complexe. Testarea, care acum merge în plin, dă foarte mult de lucru echipei de dezvoltatori care nu mai fac faţă rezolvării bug-urilor. Şedinţele de analiză cu şeful de departament arată clar că principalul vinovat este clientul care nu ştie prea bine ce vrea şi care şi-a schimbat foarte mult cerinţele. Evident, şi echipa este puţin vinovată pentru că nu a reuşit să îl ţină în frâu, dar acum clientul trebuie să înţeleagă că proiectul nu se poate termina la termen.

După alte şase luni, şeful de proiect din partea clientului îl anunţă pe directorul său general că proiectul, deşi se află în mare întârziere, nu este deloc ceea ce se aştepta să fie (deşi, privind în urmă putem vedea că această aşteptare nu a fost niciodată definită riguros), că responsabilii din departamente nu sunt mulţumiţi de sistem, iar unii au declarat că ei nu îl pot folosi sub nici o formă. La departamentul de producţie este evident că utilizarea sistemului ar produce un blocaj.

Se decide o nouă întâlnire între părţi, se analizează situaţia, se iau măsuri...

Modele teoretice de abordare a problemelor. Decompoziţia

Dacă găseşti un drum fără obstacole, probabil că drumul acela nu duce nicăieri.
J. F. Kennedy

Acest capitol reprezintă o zonă de consideraţii teoretice, independente de domeniul Analizei, dar importante pentru înţelegerea lui. Modalităţile teoretice de abordare a problemelor sunt universale şi pot fi folosite oriunde însă pentru domeniul nostru, este important să le conştientizăm şi să le înţelegem deoarece activitatea de Analiză se bazează foarte puternic pe ele. De asemenea, vreau să precizez că nu am adus în discuţie toate modelele teoretice de rezolvare a problemelor şi nici nu am inclus detalii nerelevante pentru domeniul de care ne ocupăm.

Decompoziţia

Directorul companiei XYZ se plânge că nu îşi poate planifica corect aprovizionarea şi că pierde foarte mulţi bani din cauză că nu are întotdeauna suficientă marfă atunci când apar oportunităţi de vânzare sau, invers, pierde bani pentru că i se alterează cantităţi însemnate de marfă în stoc din cauză că s-a achiziţionat mai mult decât era necesar.

Pentru a rezolva o asemenea problemă, pe care foarte adesea oamenii o formulează, de exemplu sub forma "probleme cu aprovizionarea" sau "nu se aprovizionează conform necesarului real" primele soluţii care apar sunt pe măsura gradului de generalitate al descrierii (specificaţiei) problemei şi adesea sunt, după caz, chiar hilare: "să se aprovizioneze conform necesarului real" (soluţia este o inversare a problemei: "nu se face cutare lucru", devine "să se facă cutare lucru").

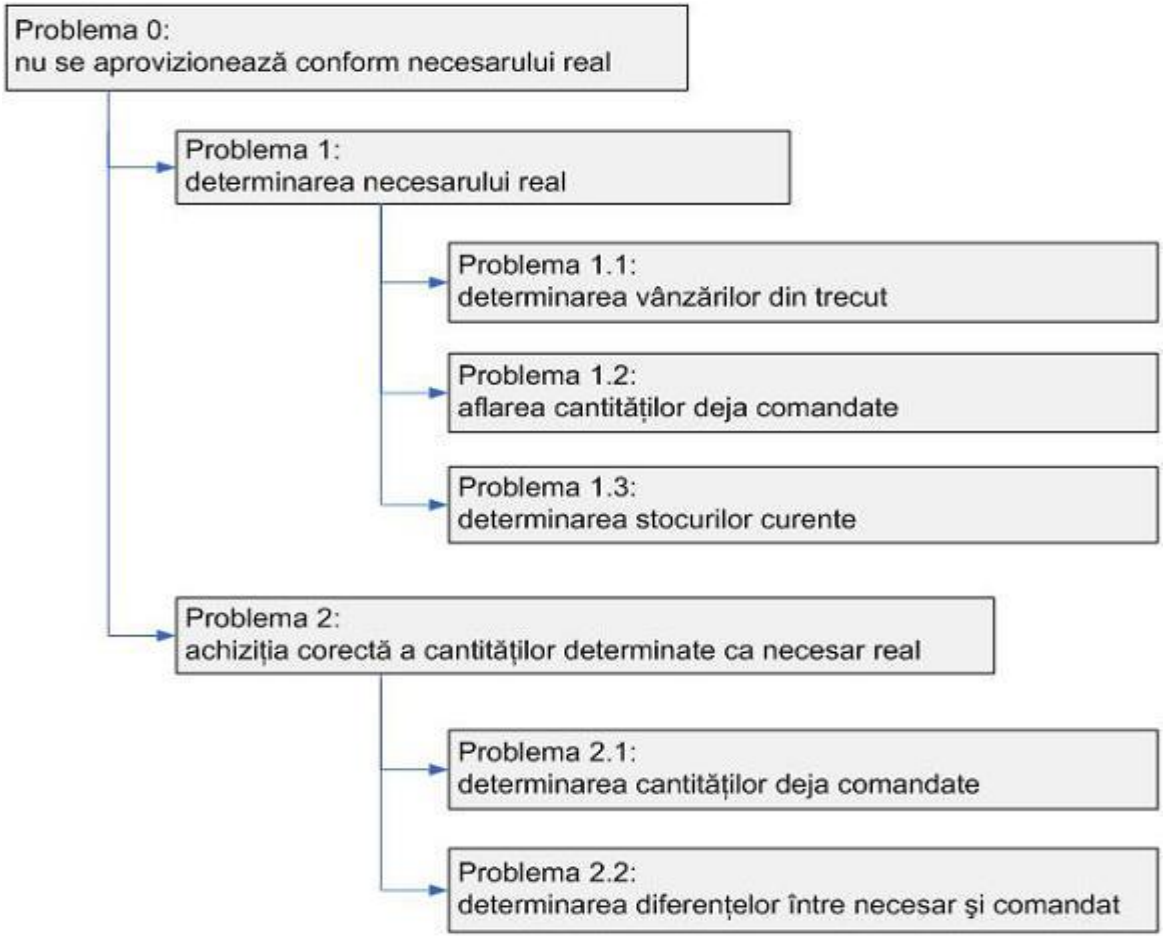
Deşi asemenea soluţii nu sunt greşite în sine, ele nu ne sunt utile. Problema iniţială, "nu se aprovizionează conform necesarului real" transformată în soluţia "să se aprovizioneze conform necesarului real" este în continuare o problemă. *Cum să se aprovizioneze conform necesarului real şi care este necesarul real?*

Pentru a răspunde acestei probleme este, evident, nevoie ca ea să fie spartă în bucăţi mai mici – adică, tradiţionalul *divide et impera* (pentru că noi românii avem o tradiţie în asta şi suntem foarte buni la aşa ceva, ştim foarte bine să fim divizaţi).

Pentru a se aproviziona conform necesarului real, domnul XYZ (dânsul este directorul companiei XYZ, desigur) va trebui *(1) să ştie care este necesarul real* şi *(2) să achiziţioneze exact atâta cât a decis că este necesarul real*.

Pentru a determina necesarul real trebuie, mai departe, să ştie *(1.1) cât a vândut în trecut*, *(1.2) ce comenzi curente* are şi *(1.3) ce cantitate de marfă are în stoc*.

Pentru a achiziţiona exact atâta cât a decis că este necesarul real trebuie să poată, în orice moment, *(2.1) să afle cantităţile deja comandate la furnizori* şi *(2.2) să poată determina diferenţele dintre necesarul calculat şi comenzile trimise la furnizori*:



Aşa cum probabil vă aşteptaţi deja, în Analiza Software, decompoziţia este utilizată foarte des. Este aproape o legătură organică, definitorie între Analiză şi descompunere. Pentru specialistul în software însă, trebuie să fie foarte clar, de la început, că nu toate sub-problemele de business determinate prin descompunere se vor rezolva prin soft. Mai multe amănunte veţi găsi în articolele următoare...

Modele teoretice de abordare a problemelor. Sinteza

Sinteza este o modalitate, prin care, din manifestări punctuale se determină problema reală. Cel mai simplu şi clar exemplu este acela al medicului, care pe baza simptomelor pacientului, stabileşte un diagnostic, stabileşte, de fapt, problema acelui pacient.

Pentru a limpezi utilitatea acestui procedeu, să ne închipuim că medicul ar proceda cam ca echipa proiectului ALPHA din exemplul din articolul Locul Analizei în proiectul software, tratând fiecare simptom în parte, dar fără să determine boala în sine: pentru febra ar administra antitermice iar pentru durerea de cap, antinevralgice. Cu siguranţă infecţia care produce simptomele ar rămâne neatinsă sau chiar s-ar agrava din cauza tratamentului neadecvat, iar simptomele ar reveni în forme şi mai dure.

Determinarea unei probleme mari, prin sinteza celor mici, indică o aparentă opoziţie cu modelul de mai sus. De ce ar fi utilă în Analiză o metodă exact opusă alteia? Aşa cum vom vedea şi în alte capitole ale cărţii, determinarea unei probleme majore, care determină apariţia unui proiect, este un pas esenţial către succesul proiectului.

Am întâlnit de multe ori exemple în care echipa de proiect nu avea aceeaşi imagine asupra problemei pe care încercau să o rezolve.

În Analiză, cele două modele nu sunt în opoziţie ci se completează unul pe celălalt. Ideea este că ele sunt utilizate în stadii diferite ale evoluţiei proiectului: în faza de culegere a cerinţelor (interviuri, documentare etc.) este utilizată cu precădere sinteza. Această fază duce la determinarea riguroasă a problemei clientului (problema pe care o soluţionăm aici este determinarea problemei clientului). Pentru a soluţiona problema clientului, se face *decompoziţia* acesteia.

Aici este important de menţionat că decompoziţia nu este o imagine în oglindă a sintezei: decompoziţia vizează elemente ale soluţiei, altele decât cele care au intrat în procesul de sinteză.

Pentru a păstra paralela de mai sus cu medicina, sinteza informaţiilor culese (adică a simptomelor bolii) duce la determinarea bolii iar stabilirea tratamentului se face prin descompunere (infecţia o tratăm cu medicamentul X, inflamaţia cu medicamentul Y şi aşa mai departe).

De exemplu, Ionel (aici vreau să dau un exemplu ca la şcoală iar Ionel este un nume cu o oarecare rezonanţă didactică) se plânge că spaţiul pe care îl are pentru a îşi depozita cărţile este împărţit în prea multe module, ceea ce îl face să se deplaseze mult pentru a căuta o carte şi în plus îi restricţionează accesul la dulapul cu dosare (Ionel lucrează des cu dosare).

De asemenea accesul la imprimantă este îngreunat din cauza cărţilor care stau peste tot în jurul acesteia.

Analizând aceste mici probleme şi desigur sintetizându-le, putem spune că Ionel are o problemă cu depozitarea cărţilor şi, cu un efort de gândire suplimentar, ne putem da seama că ar avea nevoie să îşi poată depozita cărţile pe verticală. Prin urmare o posibilă soluţie pentru el (aici exprimăm *viziunea* asupra viitoarei soluţii) ar fi o bibliotecă, sau un raft, dau un dulap care să ocupe mai mult spaţiu pe verticală şi foarte puţin pe orizontală.

De aici începe partea de soluţie la problema lui Ionel. Analizăm: pentru a rezolva problema depozitării cărţilor pe verticală avem nevoie, minimal, *(1) de nişte rafturi prinse între ele sau de perete* cu nişte *(2) elemente de prindere (cuie, şuruburi etc.)*. Iată aşadar cum am demonstrat ceea ce era de demonstrat: elementele rezultate din descompunere (rafturi, cuie, şuruburi etc.) sunt complet diferite de elementele care au intrat în procesul de sinteză (cărţi împrăştiate, efort irosit etc.).

Modele teoretice de abordare a problemelor.

Determinarea unui trunchi de bază

De foarte multe ori, în viaţa reală, o problemă nu poate fi punctată decât cu un efort substanţial sau chiar deloc. Pur şi simplu, obţinerea unui model mintal al unei realităţi foarte complexe este imposibil sau prea riscant: nu ştii de unde să începi, nu ştii care elemente sunt relevante şi care nu.

În aceste condiţii, modul natural de abordare este să construieşti o bază, un element sau un set de elemente pe care să te poţi baza şi cu ajutorul căruia să poţi construi mai departe. Altfel spus, să faci să existe un trunchi pe care, mai apoi să poată creşte ramurile şi frunzele.

De exemplu, geometria ca disciplină pe care o cunoaştem cu toţii, este construită pe câteva axiome. De la aceste axiome porneşte totul, ele sunt baza, fără de care restul teoriei nu ar fi posibilă.

Am cunoscut, în trecut, o companie în care necesitatea informatizării era simţită ca fiind importantă şi necesară dar *simptomele* de moment nu puteau conduce la aprobarea unui buget suficient de mare pentru o abordare globală. Pur şi simplu, nu se ştia decât că există probleme pe alocuri, că unele lucruri ar putea fi îmbunătăţite dar nu se puteau aduce argumente decisive în favoarea unei investiţii într-un sistem informatic.

Ideea avută de unul dintre responsabilii din companie, mi se pare grozavă: *pentru că elementele cele mai importante cu care lucrează compania sunt proiectele şi oamenii şi majoritatea problemelor noastre sunt legate de proiecte şi oameni, vom crea o bază de date în care vom gestiona proiectele şi oamenii. După ce vom şti în orice moment câte proiecte avem, care sunt ele, în ce stadiu sunt, câţi oameni sunt alocaţi, pentru cât timp şi aşa mai departe, vom şti mai bine ce vrem. La acestea ne va fi uşor să adăugăm, pas cu pas, şi alte lucruri, dar cel mai important, ne vom apropia pas cu pas de problemele noastre reale.*

Acesta a fost începutul şi s-a dovedit mai apoi că a fost un început bun. Sistemul dezvoltat a rezolvate multe probleme reale ale companiei.

Abordarea iterativă

Până acum oamenii n-au găsit alt drum spre adevăr decât greşeala.
Nicolae Iorga

Metoda iterativă presupune rezolvarea unei probleme cunoscute printr-o abordare iterativă, la fiecare iteraţie făcând un pas către rezolvarea problemei (ceea ce nu înseamnă neapărat rezolvarea integrală a unei sub-probleme, aşa cum vedem la descompunere).

La această abordare, fiecare iteraţie presupune îndeplinirea unui anumit obiectiv, obiectivul final fiind rezolvarea problemei iniţiale. Poate că această metodă poate să fie văzută ca o descompunere pe obiective, însă acest lucru este puţin important, importantă fiind ideea de abordarea iterativă.

De pildă, să analizăm problema unui voievod medieval român: trebuie să îi batem pe turci cu o armată mai puţin numeroasă. Prin descompunere aceasta înseamnă *(1) să le cunoaştem poziţia în teren*, *(2) să le cunoaştem numărul cât mai exact*, *(3) să îi împingem pe un teren mlăştinos care să îi defavorizeze*, apoi *(4) cât timp sunt blocaţi în mlaştină să îi atacăm necontenit* şi *(5) restul armatei să atace de pe dealul care mărgineşte mlaştina*.

Într-o *abordare iterativă*, în prima iteraţie voievodul îşi propune, trimiţând iscoade, să afle precis poziţia adversarilor în teren, să aibă o evaluare grosieră a numărul acestora.

În a doua iteraţie, lansarea primului atac, începe împingerea adversarilor către terenul mlăştinos (*pornind de la datele obţinute în prima iteraţie*) dar se păstrează obiectivul de a cunoaşte cu precizie poziţia lor în teren, precum şi obţinerea unor date mai precise privind numărul adversarilor şi împărţirea lor pe tipuri de arme.

La lansarea celui de-al doilea atac, adică la a treia iteraţie, se porneşte de la datele deja obţinute: poziţia actuală după primul atac, numărul adversarilor şi împărţirea lor pe tipuri de arme.

Atacul vizează poziţionarea mai precisă a adversarului astfel încât să se poată declanşa atacul de pe deal.

La a patra iteraţie se lansează atacul de pe deal dar se menţine obiectivul de a poziţiona adversarul în locul cel mai potrivit.

Dacă vreţi un exemplu şi mai clar (dar cam simplist), unul dintre tunarii voievodului trebuie să lovească o anumită ţintă. Neavând aparatură performanţă de ochire, el abordează problema iterativ: trage un foc, vede locul unde a lovit, repoziţionează tunul, tage din nou şi aşa mai departe până când îşi atinge ţinta. Ceea ce trebuie spus este că, în situaţia lui, această abordarea este cea mai realistă.

Probabil că metoda iterativă împrumută câte ceva de la fiecare dintre celelalte trei metode. Important, în cazul ei, este următorul aspect: *fiecare dintre iteraţii rezolvă atâta cât este posibil în acel stadiu şi creează baza pentru ca în iteraţia următoare să se poată face un nou pas*. Fiecare etapă trebuie să aducă echipei de proiect rezolvarea acelor probleme, precum şi suficient feed-back, încât etapa următoare să poată fi desfăşurată cu efectul scontat şi cu risc minim.

Evident, această metodă nu trebuie confundată cu ciclul de dezvoltare iterativ deoarece sunt lucruri diferite, chiar dacă ea poate fi considerată baza teoretică pentru acest ciclu.

Definiţia cerinţei software

Elementul central în Analiză este, cu siguranţă, *cerinţa software*. În jurul cerinţelor se desfăşoară totul: cerinţele trebuie culese de la clienţi, cerinţele trebuie documentate, trebuie gestionate, dezvoltate, testate. În fond, modelul creat prin specificaţiile software este un model compus din cerinţe care trebuie să se transforme în produsul final.
Din acest motiv, vom insista asupra definiţiilor date *cerinţelor software*, poate chiar mai mult decât asupra definiţiilor Analizei software în sine.

În literatura de specialitate există o mulţime de definiţii. Toate încearcă însă să cuprindă următoarele elemente esenţiale: cerinţele sunt descrieri *(specificaţii)*, într-un limbaj accesibil tuturor celor implicaţi a ceea ce un sistem informatic trebuie să poată acoperi, atât prin *comportamente* (behaviour) cât şi prin *atribute* ale sale.

Cea mai completă (şi, desigur, cea mai recunoscută) definiţie a cerinţei este dată de Institute of Electrical and Electronics Engineers (IEEE). Conform acestei organizaţii, prin standardul 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology, *cerinţa software* este definită astfel:

Cerinţa software este:

1. o ***condiţie*** sau ***capabilitate*** necesar a fi îndeplinită de către un sistem, pentru ca un *utilizator* să poată rezolva o anumită *problemă* sau să atingă un anumit *obiectiv*;

2. o ***condiţie*** sau ***capabilitate*** pe care un *sistem* trebuie să o realizeze sau să o posede pentru a satisface un contract, standard, specificaţie sau alt *document* formal impus;

3. un ***document*** – reprezentarea unei *condiţii* sau *capabilităţi*, aşa cum este descrisă la punctele 1 şi 2 de mai sus.

Mai înainte de a analiza această definiţie vom clarifica termenul "capabilitate" utilizat aici. Capabilităţile desemnează ceva ce un sistem software furnizează utilizatorilor, fie un anumit comportament fie un anumit atribut. Deşi termenul provine din englezescul "capability", limba română are aici privilegiul de a avea pentru capabilitate o descriere intuitivă: ea desemnează ceva ce un sistem este capabil să facă, ceva ce sistemul poate.

Printr-o capabilitate a unui sistem putem desemna fie un comportament fie un atribut. De pildă, o funcţionalitate de genul „sistemul validează formatul datei atunci când utilizatorul înregistrează factura în sistem” este un comportament al sistemului, în timp ce „poziţia unui buton pe ecran” este un atribut. (Mai pe româneşte, în final, un câmp dintr-o bază de date sau o proprietate a unui obiect poate corespunde unui atribut al unei entităţi, în timp ce o metodă a unui obiect înseamnă comportament.)

Revenind la definiţia cerinţei, vom detalia pe scurt cele trei părţi ale definiţiei.

Prima parte, reprezintă *punctul de vedere al utilizatorului*. Centrul atenţiei este, la acest punct, utilizatorul care are nevoie ce ceva pentru a rezolva o problemă sau pentru a atinge un obiectiv. Această parte a definiţiei ne spune clar că dacă problema/obiectivul utilizatorului nu poate fi specificat, atunci cerinţa nu există. O cerinţă există atâta timp cât ea reprezintă soluţia pentru o problemă a utilizatorului.

A doua parte a definiţiei reprezintă *punctul de vedere al dezvoltatorului*. Aici "o condiţie sau capabilitate pe care un sistem trebuie să o realizeze" este ceea ce dezvoltatorul trebuie să realizeze. Aşa cum se poate vedea din definiţie, pentru el referinţa este un "document formal impus". Vom vedea în capitolele următoare că aici nu este vorba despre o descriere a funcţiilor aşa cum se dezvoltă ele în limbaj de programare ci sunt capabilităţi care pot fi înţelese, au o logică şi pot fi validate şi de către utilizatorul sistemului, fără ca acesta să ştie programare.

Partea a treia a definiţiei exprimă un punct de vedere comun, sau un punct de vedere general, o regulă de bază: primele două puncte de vedere nu pot exista dacă nu există documentul pe care ambele părţi să îl poată folosi ca referinţă. Dacă documentul nu există, cele două puncte de vedere, precum şi evoluţia lor în timp nu au un element comun palpabil şi fără echivoc, o referinţă acceptabilă.

Aşadar, conform punctului 3 din definiţia cerinţei, o cerinţă care nu este documentată *nu există*.

Observăm, din definiţia de mai sus, că *cerinţa*, privită din partea utilizatorului sistemului este ceva util pentru rezolvarea unei probleme, în timp ce, pentru dezvoltator cerinţa este ceva ce el trebuie să dezvolte, conform specificaţiei. Ca urmare, cerinţa trebuie exprimată în aşa fel încât să poată fi interpretată fără dubii de ambele părţi, pentru că ea este destinată în egală măsură ambelor părţi.

Pe de o parte, pentru utilizator este importantă rezolvarea propriilor probleme, fără ca detaliile tehnice ce ţin de rezolvarea lor să fie relevante. De cealaltă parte, dezvoltatorul are nevoie de o referinţă pentru a şti ce să dezvolte, în timp ce problemele utilizatorului au relevanţă mai scăzută. Aici, la mijloc între cei doi se situează Analistul software şi produsul munci sale: cerinţa software – un document care arată utilizatorului soluţia problemei lui iar dezvoltatorului ce trebuie să dezvolte.

Aşa cum se vede în figura de mai sus, în cadrul evoluţiei de la Problemă la Soluţie, specificarea cerinţei este pasul intermediar: ea este, pentru utilizator, soluţia vizată a problemei lui, iar pentru dezvoltator este problema pe care o are de rezolvat.

După cum vom vedea şi în continuare, cerinţele exprimate în limbaj inteligibil, sub forma documentelor de specificaţii software, agreate de către client şi de către echipa de dezvoltare, constituie referinţa de bază pentru toţi cei implicaţi în proiectele software: pentru project manageri, în determinarea şi urmărirea task-urilor sau pentru inginerii de testare, în realizarea testelor.

Nivelurile cerinţelor

În capitolul Modele teoretice de abordare a problemelor prezentam un exemplu ipotetic al unei companii XYZ, al cărei director se plânge că nu îşi poate planifica corect aprovizionarea şi că pierde foarte mulţi bani din cauză că nu are întotdeauna suficientă marfă atunci când apar oportunităţi de vânzare sau, invers, pierde bani pentru că i se alterează cantităţi însemnate de marfă în stoc din cauză că s-a achiziţionat mai mult decât era necesar.

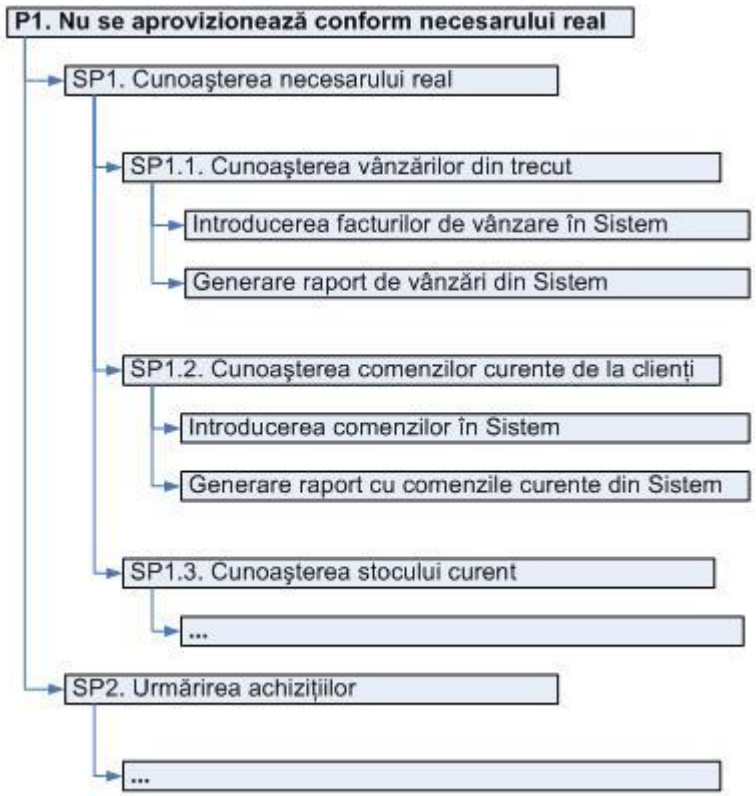
Prin descompunere, problema era spartă în sub-probleme astfel:

- sub-problema 1: cunoaşterea necesarului real:
- sub-problema 1.1: cunoaşterea vânzărilor din trecut;
- sub-problema 1.2: cunoaşterea comenzilor curente de la clienţi;
- sub-problema 1.3: cunoaşterea stocului curent.

- sub-problema 2: urmărirea achiziţiilor astfel încât să se achiziţioneze exact atâta cât a decis că este necesarul real:
- sub-problema 2.1: cunoaşterea cantităţilor deja comandate la furnizori;
- sub-problema 2.2: determinarea diferenţelor dintre necesarul calculat şi comenzile trimise la furnizori.

Mergând cu decompoziţia încă un nivel mai jos constatăm că suntem deja în situaţia de a da soluţii destul de concrete pentru unele dintre problemele din structură. Sigur că soluţia finală şi cât se poate de concretă este software-ul însuşi, dar fiecare pas în detaliere înseamnă un nou pas către soluţia finală.

Cu acest nou pas vom obţine o structură precum cea din figura de mai jos:



Astfel sub-problema *1.1 Cunoaşterea vânzărilor din trecut* se transformă în: *Introducerea facturilor de vânzare în Sistem* şi *Generare raport de vânzări din Sistem* (presupunem că rezolvând aceste două chestiuni se rezolvă complet problema cunoaşterii vânzărilor din trecut, chiar dacă în realitate lucrurile ar putea să fie mai complicate).

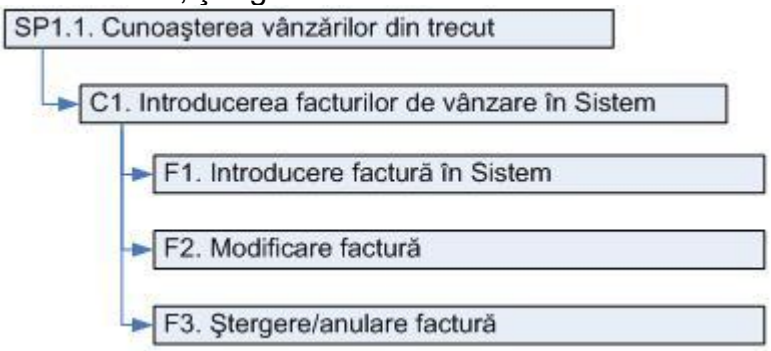
De la acest nivel de detaliere e clar că anumite probleme, şi anume cele de pe ultimul nivel de detaliere, nu mai sunt probleme pe care directorul companiei XYZ, sau chiar managerul de achiziţii, să le rezolve nemijlocit ci sunt task-uri adresate altui nivel de utilizatori.

Astfel, dacă task-ul determinării necesarului real şi al urmăririi achiziţiilor este destinat nivelului managerial, task-ul *Introducerea facturilor de vânzare* în Sistem este adresat unui alt utilizator, implicat direct în derularea business-ului.

Noul nivel adăugat însemnă nivelul la care este vizibilă interacţiunea dintre utilizatori şi Sistem. Pentru "Introducerea comenzilor în sistem" este evident că este necesară o funcţionalitate în Sistem, aferentă acestei probleme. În fond, pe acest nivel ajungem la cerinţele software pure, cele care se supun 100% definiţiei date la capitolul referitor la definiţia cerinţei: din punctul de vedere al

utilizatorului trebuie rezolvat task-ul introducerii facturii în Sistem iar din punctul de vedere al dezvoltatorului trebuie dezvoltată funcţionalitatea aferentă.

Dacă ar fi să se detalieze din nou chestiunea „Introducerii comenzilor în sistem”, dar propunându-ne să ne modelăm deja viitorul sistem informatic (să ne imaginăm un flux complet de lucru cu Sistemul!) ar rezulta următoarele funcţionalităţi pretinse Sistemului: adăugare factură, modificare factură, ştergere/anulare factură:



Acesta este nivelul la care granularitatea maximă a problemelor din business-ul real ating nivelul la care poate fi conceput un sistem informatic: adaugă, calculează, şterge, modifică, selectează, compară etc.

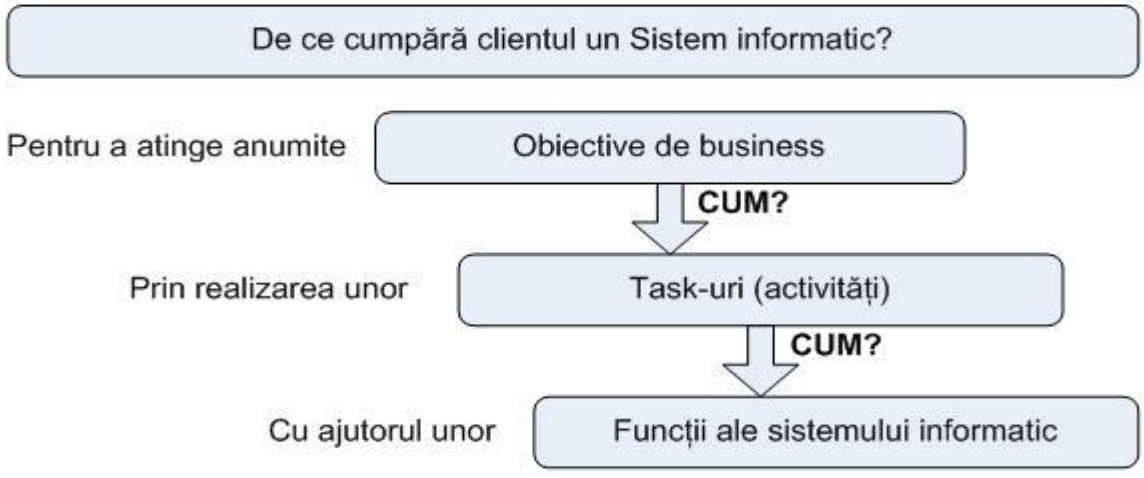
Privind în sens invers, de jos în sus, aceste funcţionalităţi sunt acelea pe care utilizatorul, într-un flux de lucru cu Sistemul, le utilizează pentru rezolvarea task-urilor sale. Apoi, pe un nivel mai sus, din task-urile utilizatorilor sunt rezolvate problemele de business.

Prin urmare, se pot stabili următoarele niveluri ale cerinţelor software [1]:

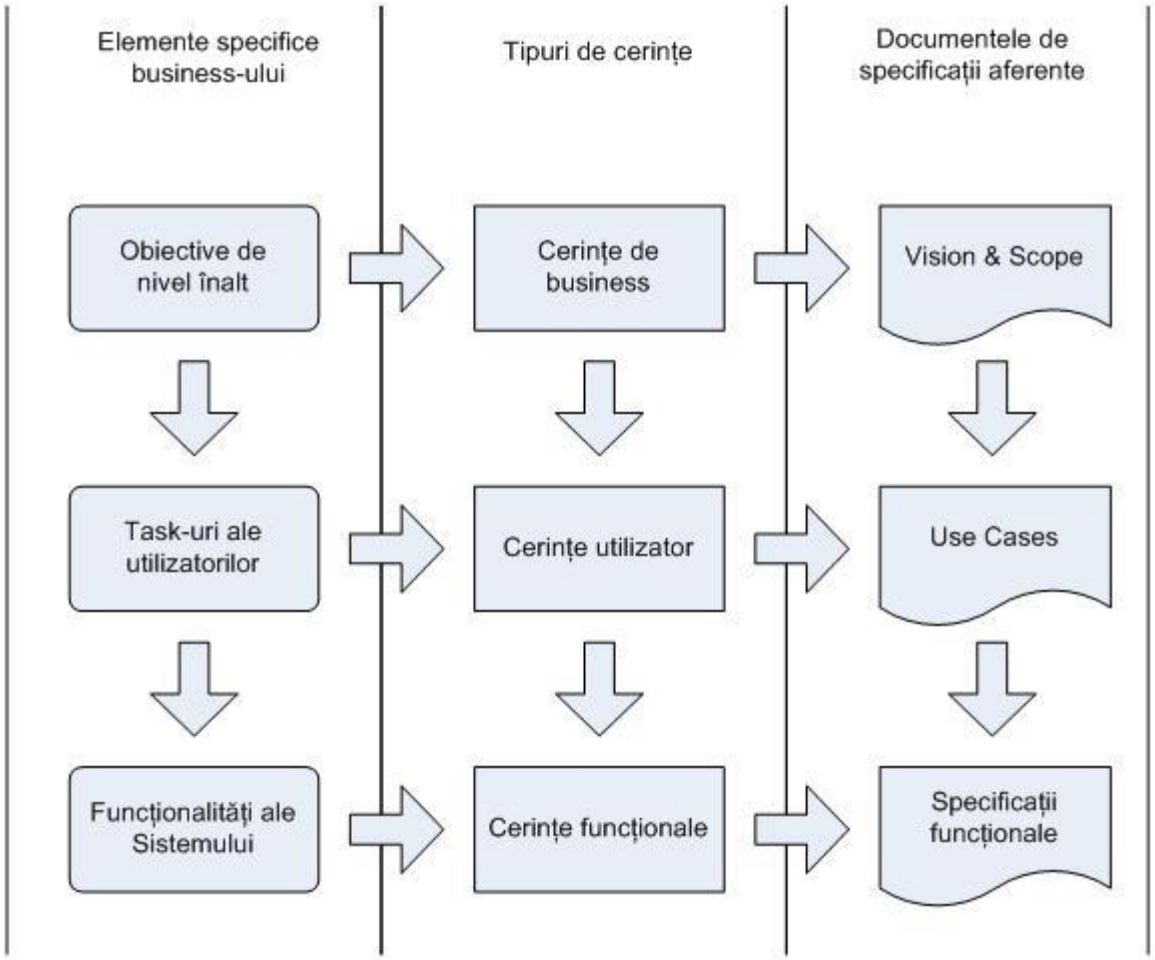
A. Cerinţe de business: acestea sunt cerinţele de pe cel mai înalt nivel şi sunt obiectivele (sau problemele) de nivel înalt ale clientului. Aşa cum vom vedea în continuare aceste cerinţe sunt de obicei descrise într-un document denumit Vision & Scope.

B. Cerinţe utilizator: pe acest nivel sunt task-urile pe care utilizatorul le va putea îndeplini utilizând produsul software. Aceste cerinţe sunt specificate de obicei sub formă de use case-uri.

C. Cerinţe funcţionale: sunt cerinţele adresate direct viitorului Sistem, funcţionalităţile care trebuie dezvoltate pentru ca utilizatorii să îşi poată îndeplini task-urile. Acest nivel este nivelul cel mai apropiat de designul Sistemului.



În figura de mai jos se pot vedea atât nivelurile cerinţelor, cât şi nivelurile din business care le generează, precum şi documentele de specificaţii utilizate în general pentru acel nivel de cerinţe (săgeţile cu vârful în jos înseamnă de obicei descompunere):



Trebuie precizat că documentele de specificaţii au denumiri diferite, în funcţie de metodologie. De exemplu, în anumite metodologii documentul Vision & Scope este denumit Vision sau Specificaţie de business.

De asemenea, este important de precizat că decompoziţia prezentată în exemplul de mai sus are rol teoretic şi nu este întotdeauna un mod de lucru recomandat. Deşi este nu doar inevitabilă, ci şi normală, utilizarea descompunerii în Analiză, ea trebuie completată cu (sau derulată împreună cu) analiza pe fluxuri, concretizată prin utilizarea use case-urilor. În exemplul de mai sus, cerinţele corespunzătoare unor task-uri concrete trebuie detaliate şi analizate sub formă de use case-uri.

Pentru mai multe detalii vă recomand capitolul dedicat use case-urilor.

Unde se termină cerinţele clientului şi unde începe designul?

Urmărind capitolul anterior, probabil că fiecare dintre noi şi-a ridicat problema "unde se termină această descompunere?" sau "când se poate spune că detalierea este suficientă?". Acest lucru încerc să îl clarifica în capitolul de faţă.

Urmărind descompunerea, de sus în jos, putem observa că la orice nivel ne-am situa, pe nivelul imediat inferior avem o propunere de soluţie. Astfel, *Cunoaşterea necesarului real (SP1)* se poate rezolva prin *Cunoaşterea vânzărilor din trecut (SP1.1), Cunoaşterea comenzilor curente de la clienţi (SP1.2)* şi *Cunoaşterea stocului actual(SP1.3)*. Prin urmare, nivelul inferior este răspunsul la întrebarea **"CUM?"** privitoare la nivelul curent. Dacă ne poziţionăm pe nivelul *Cunoaşterea vânzărilor din trecut (SP1.1)* şi ne punem întrebarea "cum rezolvăm această problemă?", răspunsul se găseşte

pe nivelul inferior: prin *Introducerea facturilor de vânzare (C1)* şi *Generare raport de vânzări (C2)*. Şi aşa mai departe.

Dacă ne situăm pe un nivel oarecare şi privim de jos în sus putem vedea problema pentru care nivelul respectiv reprezintă o soluţie. Nivelul superior este răspunsul la întrebarea **"CE?"** privitoare la nivelul curent (sau *"ce se rezolvă prin..."*). De pildă, *Introducerea facturilor de vânzare (C1)* este un element care ajută la *Cunoaşterea vânzărilor din trecut (SP1.1)*.

Prin urmare, pornind de la problema iniţială, fiecare nivel de descompunere reprezintă o **soluţie**, dar în acelaşi timp o **problemă**. Totuşi, la un anumit nivel trebuie să se găsească soluţia finală.

Judecând lucrurile practic, ar fi o eroare să ne închipuim că putem să lungim lucrurile la nesfârşit. Undeva trebuie să se termine (aşa cum bugetul alocat proiectului se termină sigur undeva).

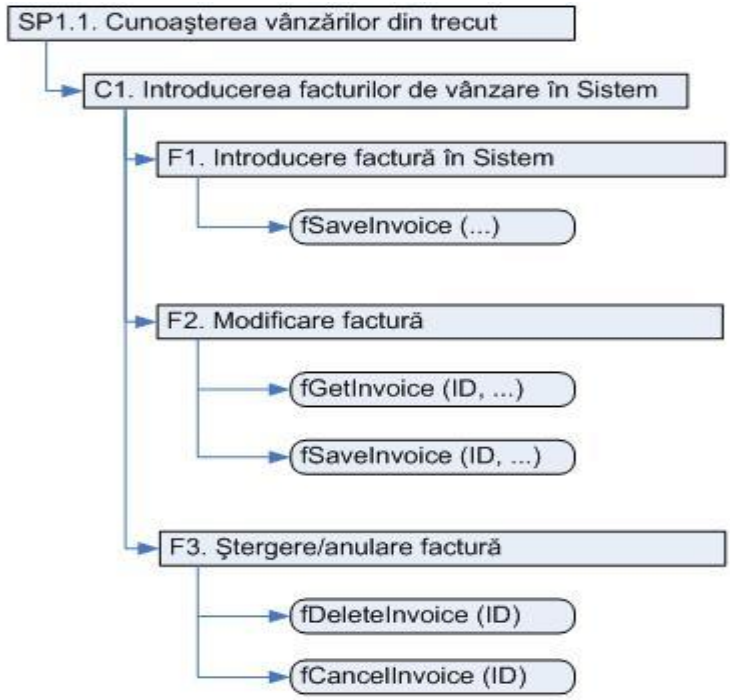
PROBLEMĂ vs. SOLUŢIE

Privind lucrurile din punctul de vedere al întregului proiect, soluţia ultimă a problemei clientului este desigur aplicaţia software în sine, programul executabil livrat clientului. Din punctul de vedere al analistului însă, programul executabil final este punerea în practică, întocmai, a unui model (a unei *specificaţii*), ceea ce conduce la ideea că, pentru analist, dar nu numai pentru el, soluţia ultimă a problemei clientului se găseşte într-o specificaţie, ce urmează a fi modelul pentru dezvoltarea executabilului – altfel spus, proiectul viitorului executabil.

Aşadar, putem împărţi spaţiul dintre problema iniţială a clientului şi soluţia, constând în modelul viitoarei aplicaţii software în două părţi distincte: **PROBLEMA** şi **SOLUŢIA**. În zona PROBLEMEI vom situa acele specificaţii care descriu problema clientului iar în zona SOLUŢIEI acele specificaţii care descriu modul de rezolvare a PROBLEMEI. Trebuie spus, din start că nu toate specificaţiile din zona SOLUŢIEI sunt de competenţa analistului. Dimpotrivă, cea mai mare parte a soluţiei este concepută şi descrisă de către arhitecţi software sau designeri de soluţii software.

De la cerinţele clientului la cod

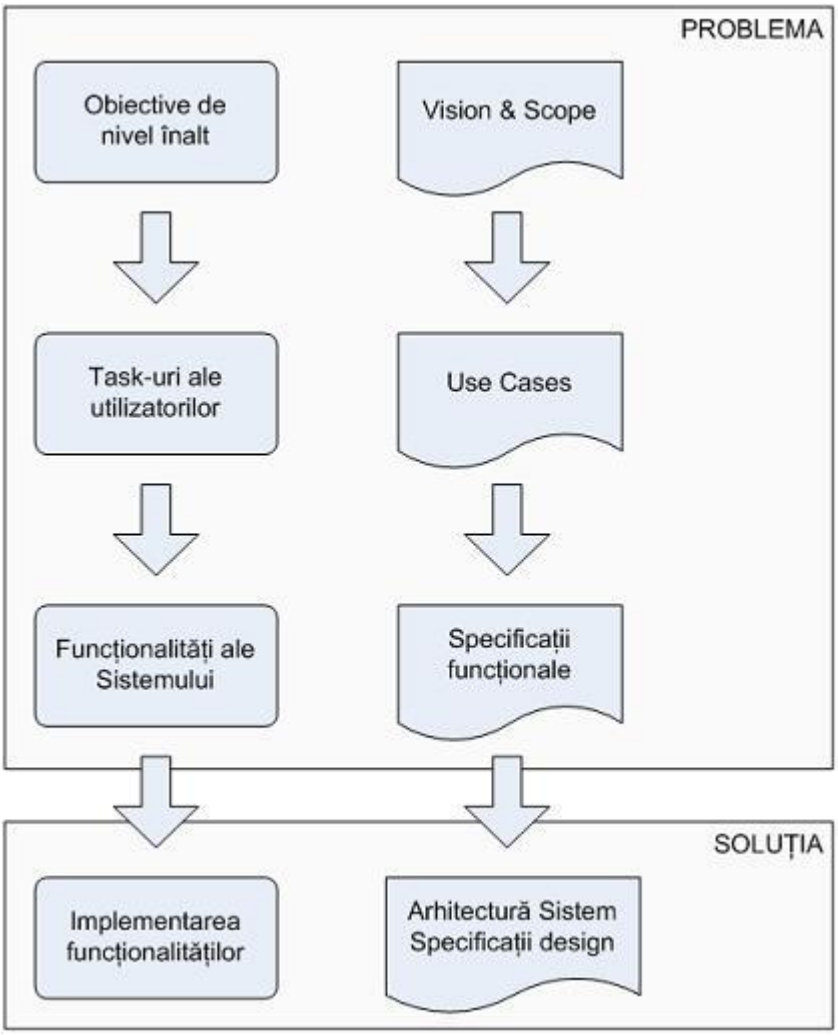
Probabil că, în continuarea exemplului de la capitolul anterior, detalierea pe încă un nivel ar însemna descrierea modului de rezolvare a problemelor prin soft, detalierea funcţiilor pe care viitorul sistem le va avea:



Acest nou nivel de detaliere începe deja descrierea propriu-zisă a aplicaţiei software. La acest nivel, după descrierea pe sub-nivele a PROBLEMEI, începe descrierea SOLUŢIEI.

Nivelurile cerinţelor vs. documente de specificaţii

Pentru diversele etape, atât în privinţa evoluţiei în timp a cerinţelor cât şi în privinţa detalierii informaţiile şi documentele implicate sunt următoarele:



Pentru a elimina orice confuzie în legătură cu specificaţiile funcţionale, trebuie spus că acestea nu conţin detalierea a cum se dezvoltă acestea, nu conţin elemente de design ci sunt văzute ca funcţionalităţi pe care Sistemul software trebuie să le posede pentru a permite realizarea task-urilor utilizatorilor. Sistemul este văzut aici ca o cutie neagră – ştim ce funcţii trebuie să îndeplinească dar nu ştim cum.

Mai trebuie spus că de la o metodologie de lucru la alta tipurile, conţinutul documentelor sau chiar limitele dintre PROBLEMĂ şi SOLUŢIE pot să difere. În unele cazuri PROBLEMA se încheie cu specificarea use case-urilor (ceea ce înseamnă acoperirea task-urilor utilizatorilor), în altele odată cu Specificaţiile funcţionale.

Personal, deşi susţin cu tărie că analistul trebuie să se limiteze la zona de business vă recomand cu căldură să vă inspiraţi din metodologiile "clasice": RUP, MSF etc. sau din standardele existente (de pildă IEEE Std 830-1993: IEEE Recommended Practice for Software Requirements Specifications). Aceasta vă va ajuta să vă formaţi propria părere şi să înţelegeţi cum este mai bine să împărţiţi sarcinile în organizaţia dumneavoastră.

Problema clientului este legată întotdeauna de business

În principiu, treaba analistului se termină acolo unde încep să fie vizibile interacţiunile dintre utilizatori şi viitorul sistem informatic. Din acest punct începe treaba arhitectului software.

În fond, specificaţiile cerinţelor se termină, logic, acolo unde clientul nu poate sau nu trebuie să impună propriul punct de vedere.

De exemplu, structura bazei de date, variabile utilizate, componente software reutilizabile, împărţirea pe module nu sunt lucruri asupra cărora clientul trebuie să se pronunţe (cu rare şi nedorite excepţii).

Întotdeauna se va avea în vedere că documentul de specificaţii trebuie asumat şi semnat de către client în cunoştinţă de cauză, în deplină înţelegere a conţinutului său, motiv pentru care nu i se poate pretinde asumarea soluţiilor tehnice sau validarea unor detalii care depăşesc capacitatea tehnică a acestuia.

În concluzie, documentaţia care precede dezvoltarea propriu-zisă a codului unei anumite aplicaţii software descrie, pe de o parte PROBLEMA, pe de altă parte SOLUŢIA. Rolul Analizei este să descrie cât mai corect şi complet PROBLEMA, restrângând doar din punct de vedere al business-ului clientului spaţiul SOLUŢIILOR posibile, fără să introducă restricţii tehnologice.

În general, PROBLEMA clientului este o *problemă de business* nu o *problemă de tehnologie*.

Riscuri legate de cerinţe

Daca vrei sa afli o cale spre mai bine, e nevoie sa privesti îndelung tot ce este mai rau.
Thomas Hardy

Într-un articol anterior am descris un mic exemplu, ipotetic (exemplul cu proiectul ALPHA) al unui proiect ratat. Spuneam acolo ca, printre altele, echipa de proiect nu a stiut sa abordeze chestiunile de Project Mangement sau Analiza. Cu siguranta ca asa este, dar mai mult decât atât, echipa proiectului ALPHA nu a pornit la drum cu o viziune realista asupra a ceea ce este în lumea reala, a afacerilor, un proiect software si a constrângerilor impuse unui proiect software.

Probabil ca într-o lume ideala, în care programatorii ar fi avut la dispozitie un buget nelimitat, termene nelimitate, o echipa stabila si un set stabil de functionalitati, proiectul s-ar fi sfârsit cu bine. Asta însemnând ca functionalitatile cerute ar fi fost cândva gata, la parametrii de calitate ceruţi.

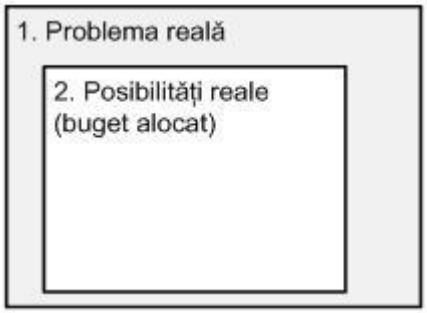
În viata reala însa, lucrurile nu sunt la fel de usoare (vedeti si capitolul Axiome ale dezvoltarii de software!). Proiectul real porneste întotdeauna cu un buget limitat si evolueaza întotdeauna în directia adaugarii de cerinte cât pentru trei bugete (axioma întâia), dar se termina de cele mai multe ori la un nivel de calitate mult sub cel dorit.

Aici este, de fapt, întreaga maiestrie: sa controlezi un proiect real, afectat de multimea de constrângeri, sa controlezi schimbarile permanente, sa negociezi în permanenta succesul.

Acesta este si motivul pentru care adevaratii Project manageri, analisti, arhitecti software sau programatori sunt atât de bine platiti, ceea ce fac ei nu poate fi facut oricum si de catre oricine.

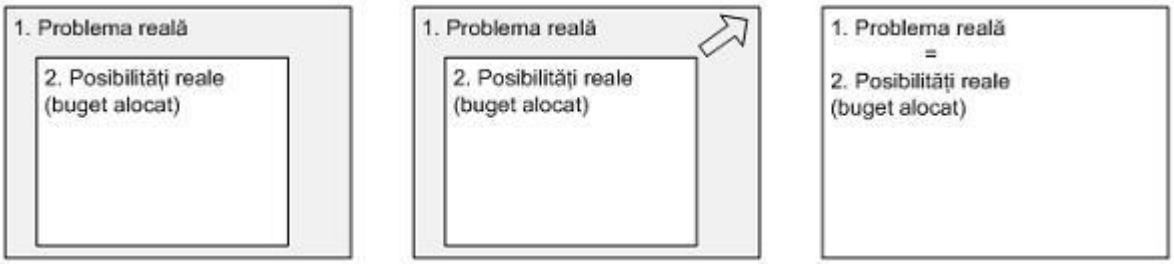
Succesul unui proiect software, pândit din toate partile de o sumedenie de constrângeri, probleme si riscuri se masoara pe trei coordonate: **functionalitati**, **calitate** si **buget**. Încadrarea sau neîncadrarea în limitele stabilite de la început ne dau masura succesului proiectului, iar pastrarea acestei încadrari nu se poate face, în contextul *schimbarilor* permanente, decât pastrând controlul asupra a ceea ce este important si prin negociere. Ori, doua dintre aceste coordonate (*functionalitati* si *calitate*) precum si informatiile pentru a controla si decide asupra schimbarilor provin din analiza. De aici deriva si importanta riscurilor aferente analizei, care atunci când se transforma în probleme concrete au efecte devastatoare asupra întregului proiect.

Pentru a analiza riscurile potentiale ale activitatii de analiza vom porni de la ceea ce ar trebui sa fie într-un proiect. Lucrurile de la care pornim sunt *problema* clientului si *resursele* posibil de alocat pentru rezolvarea acesteia:

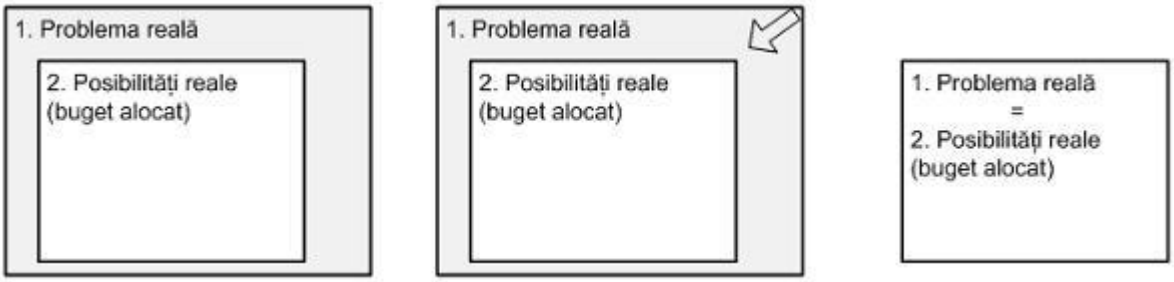


Problema reala a clientului este, de obicei, foarte vasta, mai ales în raport cu bugetul alocat. De aceea, ceea ce trebuie sa fie domeniul problemei pe care își propune, în mod realist, sa o rezolve un proiect trebuie negociat astfel încât dezvoltatorul sa nu fie obligat sa lucreze în pierdere (lucru care de obicei degenereaza într-o pierdere de ambele parti) iar clientul sa rezolve cu adevarat problema de baza.

Într-o prima varianta clientul întelege ca solutionarea problemei lui solicita o investitie mai mare si accepta modificarea bugetului pâna la acoperirea întregului necesar:



În cealaltă varianta clientul, constrâns de probleme bugetare, nu poate mari suma alocata si împreuna cu consultantul sau decide sa reduca domeniul problemei la ceea ce își permite sa cheltiue (în urma unei analize de impact, fie gaseste acele 20% dintre cerinte care rezolva 80% din problema, fie decide ce este mai important si mai profitabil sa implementeze în acest proiect si ce se poate amâna pentru un alt proiect viitor).



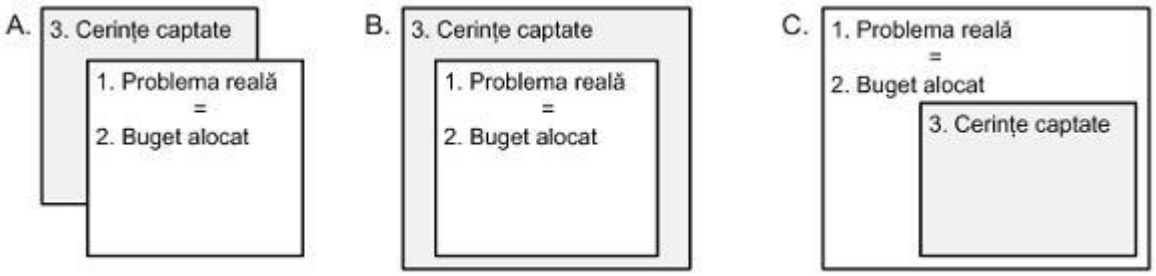
În orice caz, la sfârsitul acestei etape presupunem ca avem declarata, chiar în termeni vagi, o anumita problema si avem alocat bugetul adecvat. Desi în realitate este putin probabil sa se ajunga aici, pentru noi aceasta simplificare a modelului este utila pentru întelegerea situatiilor care pot sa apăra.

Prin urmare noi vom presupune ca la primul pas am definit (chiar daca destul de vag) *domeniul problemei*.

Urmatorul pas, culegerea cerintelor este o detaliere a problemei, si rezultatul ei final ar trebui sa fie tot o suprapunere perfecta peste acel domeniu, numit de noi problema reala sau domeniul

problemei. În realitate însa nu ne putem astepta la o suprapunere perfecta dar ne propunem, în mod realist, pastrarea unei diferente minimale, controlata, care sa nu afecteze obiectivele de baza ale proiectului.

Situatiile nefavorabile care pot sa apara sunt urmatoarele:



A. În prima situatie, cerintele captate nu acopera problema în întregime (anumite lucruri nu se vor rezolva) însa produce costuri considerabile prin includerea unor cerinte inutile sau a unor cerinte care ies din domeniul initial al problemei (probleme reale dar care nu contribuie la rezolvarea problemei).

B. În a doua situatie, cerintele captate si acceptate a fi incluse în proiect depasesc posibilitatile bugetare ale proiectului si va conduce la pierderi financiare, litigii în justitie sau la abateri grave de la calitate.

C. A treia situatie neplacuta, este rezolvarea incompleta a problemei prin omiterea unor cerinte sau întelegerea vaga si incompleta altora sau chiar a problemei.

La ce efecte negative ne putem aştepta?

Prin urmare, oricare dintre variante poate conduce la următoarele efecte negative:
- dezvoltarea unor funcţionalităţi inutile;
- dezvoltarea unor funcţionalităţi utile dar în afara domeniului problemei, care ridică probleme de buget;
- omiterea unor funcţionalităţi necesare sau chiar esenţiale.

Am numit aceste rezultate negative „efecte negative” pentru că lucrul către care trebuie să ne îndreptăm sunt de fapt cauzele de la baza întregului fenomen negativ.

Recapitulând, efectul ultim este nereuşita proiectului în ceea ce priveşte atingerea obiectivelor legate de funcţionalităţi, calitate şi buget (simultan), iar din punct de vederea al analizei cerinţelor aceasta porneşte de la dezvoltarea unor funcţionalităţi inutile, dezvoltarea unor funcţionalităţi necesare dar din afara domeniului problemei şi omiterea unor funcţionalităţi necesare. La rândul lor aceste fenomene pornesc de la un set de cauze, aşa cum sunt descrise în tabelul de mai jos:

Cauze	Efect negativ
- înţelegerea slabă a problemei clientului, a raţiunii proiectului; - insuficienta implicarea a clientului; - cerinţe ambigue; - adăugarea necontrolată a unor cerinţe a căror raţiune este neclară;	- dezvoltarea unor funcţionalităţi inutile;
- înţelegerea slabă a problemei clientului, a raţiunii proiectului; - supraîncărcarea proiectului cu cerinţe, din cauza pierderii controlului asupra schimbărilor;	- dezvoltarea unor funcţionalităţi necesare dar din afara domeniului problemei;

- înţelegerea slabă a problemei clientului, a raţiunii proiectului; - insuficienta implicarea a clientului; - cerinţe ambigue, specificaţii incomplete;	- omiterea unor funcţionalităţi necesare;

Între cauzele prezentate mai sus *"înţelegerea slabă a problemei clientului"*, deşi poate să pară puţin surprinzătoare este un fenomen foarte des întâlnit.

Pur şi simplu, întrebaţi membrii unei echipe de dezvoltare ***"de ce se dezvoltă acest proiect?"*** sau ***"de ce clientul dă bani pe acest soft?"***.

Veţi fi surprinşi să aflaţi că acel proiect se dezvoltă pentru că *"aşa avem contract cu clientul" (păi nu...!?)* sau *"nu ştiu, dar dacă clientul plăteşte... e treaba lui"*, ori "pentru că aşa trebuie, nu poţi să rămâi în urmă cu tehnologia".

Cel mai dramatic este atunci când asemenea răspunsuri vin de la persoane care sunt analişti sau project manageri, încât te întrebi, firesc, cum se stabilesc acolo priorităţile şi pe ce bază se negociază schimbările?

Mai multe despre rolul clientului

Încă de la primele articole spuneam că *"insuficienta implicarea a clientului"* este prima cauză pentru problemelor privind calitatea şi livrarea la timp, în proiectele software.

Altfel spus, situaţia normală ar fi aceea în care clientul este actorul principal în definirea cerinţelor – proiectul de dezvoltare fiind proiectul în primul rând al lui, nu al echipei de dezvoltare. Nimic din ceea ce este cerinţă pentru viitorul sistem nu trebuie presupus şi de asemenea, nici-o schimbare a cerinţelor nu se presupune a fi acceptată fără consultarea clientului şi explicarea impactului schimbării.

În permanenţă analistul trebuie să fie primul care îşi doreşte să obţină feed-back de la client, chiar dacă acesta este negativ (*doar feed-back-ul negativ îţi garantează corectarea traiectoriei atunci când ai pornit pe un drum greşit*). Acesta este modul de lucru principal al analistului, mecanismul de corectare şi de finisare a specificaţiei.

Dacă aţi lucra cu metal, lemn sau piatră aţi avea, cu siguranţă instrumente specifice pentru corectarea erorilor şi pentru şlefuirea materialului, atunci când lucraţi cu informaţie, instrumentul de care aveţi nevoie, se cheamă *feed-back*.

Cerinţele ambigue sunt acelea care pot fi interpretate în mai multe feluri fără faultarea logicii.

Tuturor ne scapă asemenea lucruri, pentru că noi ştim ce vrem să spunem şi "înţelegem" chiar dintr-o frază pe care o construim prost. Mai mult, chiar în comunicarea între două persoane se pot transmite lucruri ambigue pentru că cel care recepţionează un mesaj crede că a înţeles ceea ce trebuie. Atâta timp cât el crede sincer că a înţeles ce trebuie, e chiar dificil să depistezi ambiguitatea.

Totuşi, posibilităţi de depistare a ambiguităţilor există. Folosiţi *use case*-uri, organizaţi *review*-uri formale ale specificaţiilor, trimiteţi responsabililor de la *testare* specificaţia pentru crearea planului de teste, înainte de a se dezvolta aplicaţia şi insistaţi să se scrie prima versiune a *manualului de utilizare* pe baza specificaţiei.

Mai ales cerinţele neclare, dificile, acelea de care îţi vine, mai degrabă să scapi cât mai repede decât să insişti pentru clarificarea lor, trebuie avute în vedere aici. Nici-o ambiguitate nu va scăpa până la sfârşit (acceptanţa!) – orice datorie neplătită va fi plătită la un moment dat, dar cu o penalizare mult mai mare.

În privinţa chestiunii "adăugării necontrolate a unor cerinţe a căror raţiune este neclară" putem spune că orice cerinţă a cărei raţiune nu este clară, nu ar trebui, de fapt, să fie considerată cerinţă. La fel ca la raţiunea proiectului, dacă răspunsul la întrebarea "de ce?" este ceva de genul "pentru că aşa vrea clientul ...", acea cerinţă este incomplet înţeleasă.

În acest grup intră cu succes unele funcţionalităţi de tip *nice to have* propuse de dezvoltatori sau de utilizatori, precum "îmbunătăţiri" ale interfeţei utilizator, sau "îmbunătăţiri" mânate de dorinţa utilizării unei tehnologii noi, proaspăt descoperită de programator. Am participat, aici în România, un proiect la care utilizarea XML-ului a fost motivaţia complicării inutile a proiectului, a adăugării unor funcţionalităţi doar pentru că "XML-ul permite", lucruri care au condus la un eşec aproape total al proiectului. După părerea mea, cel puţin jumătate din funcţionalităţile dezvoltate în acel proiect au fost inutile.

Ultima chestiune pusă în discuţie, "supraîncărcarea proiectului cu cerinţe, din cauza pierderii controlului asupra schimbărilor" nu este deloc cea din urmă.

Dimpotrivă, prin faptul că este o permanenţă în proiectele software (*axioma A1: întotdeauna cerinţele se schimbă pe parcursul derulării proiectului*...), prin efectele devastatoare (apropo de axioma *A6: nici un proiect software nu dispune de un buget nelimitat*...) această chestiune este una vitală. Dacă lucraţi într-un proiect mare, ignoraţi-o şi veţi pierde!

Vestea proastă este că factorul care generează această problemă, este schimbarea cerinţelor, *lucru care nu poate fi evitat*, şi că impunerea controlului asupra schimbării cerinţelor este un lucru greu de realizat şi nu se poate face decât prin respectarea riguroasă a unei proceduri de includere a schimbărilor în proiect. Pentru fiecare schimbare trebuie evaluat impactul asupra întregului (lucru care, în sine, costă) şi trebuie decis dacă cerinţa poate fi inclusă în proiect, dacă este necesară modificarea sau eliminarea altor cerinţe, renegocierea bugetului sau a termenelor.

De asemenea, procedura de control al schimbării cerinţelor trebuie respectată de ambele părţi, atât dezvoltator cât şi client. Nimeni nu poate schimba cerinţele fără respectarea procedurii (decât, poate, în măsura în care schimbă automat şi bugetul şi termenele la niveluri acoperitoare).

Cerinţele nefuncţionale

Una dintre greşelile cele mai frecvente în specificaţiile software este tratarea superficială a cerinţelor nefuncţionale.

Acestea pot fi cerinţe legate de atributele de calitate a produsului, cerinţe privind performanţa, respectarea unor standarde, regulamente, contracte, interfeţe externe sau alte constrângeri de design.

Am întâlnit cu câţiva ani în urmă un caz, zic eu, relevant. Aplicaţia era o aplicaţie pe web, destul de obişnuită de altfel, cu cerinţe privind performanţa destul de normale, chiar modeste.

Din păcate, specificaţia dădea drept cerinţă încărcarea a sute de pagini simultan, lucru greu de realizat cu un server obişnuit (la asemenea cerinţe, de altfel inutile, nu răspund nici site-urile mari de pe Internet). Nu e cazul să mai spun că deşi cerinţa fusese propusă de către echipa de dezvoltare nu de către client, în încercarea de a o rezolva s-a pierdut timp, pentru ca în final, dificultăţile ridicate de o asemenea ţintă să conducă la rediscutarea şi "renegocierea" cu clientul a cerinţei respective. Tratarea cu superficialitate a cerinţelor privind performanţa a condus, iată, la imposibilitatea respectării specificaţiei.

În afara faptului că situaţia a fost destul de jenantă, cerându-i-se clientului să renegocieze ceva ce echipa de dezvoltate propusese şi scrisese, de bună voie şi nesilită de nimeni, în specificaţie, au fost implicate şi costuri total nejustificate.

Determinarea *cerinţelor nefuncţionale* trebuie să urmeze un proces sănătos, de la determinarea lor şi până la specificare.

În primul rând, determinarea acestor cerinţe este un lucru dificil, pentru că, în general utilizatorii nu sunt interesaţi, şi nici puşi în temă, în legătură cu costurile cerinţelor lor şi au tendinţa să exagereze: "sistemul trebuie să lucreze 24 de ore din 24", "timpul de răspuns?... păi sistemul trebuie să răspundă instantaneu la orice comandă", "nu se admite nici un bug în funcţionarea sistemului".

În realitate nu este deloc important, nici măcar util, să se arunce banii pe obţinerea unor caracteristici care, de fapt, sunt excepţionale, ale produselor software. Şi asta pentru că nu toate aplicaţiile software sunt folosite pentru ghidarea navetelor spaţiale sau pentru controlul centralelor nucleare.

Dimpotrivă, am văzut cu toţii site-uri de succes care nu răspund chiar instantaneu la solicitări, sau aplicaţii de calcul tabelar, extrem de utile şi practice dealtfel, care nu pot gestiona fără pierderi de performanţă cantităţi foarte mari de date.

Prin urmare, ceea ce trebuie neapărat identificat sunt cerinţele cu adevărat importante, specifice business-ului respectiv, care pot produce pierderi sau dificultăţi reale. De pildă, în unele cazuri, ar putea fi necesară disponibilitatea sistemului 24 de ore din 24 pentru facturare, chiar dacă sunt acceptabile unele deprecieri ale timpului de răspuns în anumite intervale orare. În fiecare caz în parte există un raport optim între performanţe şi costuri.

Cerinţele nefuncţionale includ cerinţele de calitate precum:

- cerinţele de performanţă (viteza de răspuns, disponibilitatea sistemului, timpul de recuperare în caz de indisponibilitate temporară a sistemului, utilizarea resurselor);
- siguranţa în funcţionare (frecvenţa avariilor, uşurinţa recuperării);
- suportabilitatea (posibilităţile de adaptare, posibilităţile de extindere, configurabilitatea, compatibilitatea cu alte sisteme, localizare);
- cerinţe de implementare (standarde, legislaţie aplicabilă, politici de securitate, limitări de resurse);
- uşurinţa în utilizare (consistenţa interfeţei utilizator, standarde de ergonomie aplicabile, documentaţie, help);
- constrângeri de design;
- cerinţe de interfaţare cu alte sisteme.

Caracteristicile cerinţelor software

Atunci când sunteţi în situaţia de a culege, analiza şi specifica cerinţe (să zicem, de exemplu atunci când sunteţi analist software) trebuie să aveţi în vedere că, indiferent de forma în care o specificaţi, în limbaj natural, în UML sau în orice alt limbaj, sub formă de use case-uri sau full text, indiferent de tipul lor, cerinţele trebuie să aibă anumite caracteristici care le fac să fie cerinţe adevărate, corect specificate şi posibil de realizat, în parametrii bugetari şi de calitate determinaţi.

Înainte de a trece efectiv la enunţarea acestor caracteristici, vă invit să vă gândiţi la cerinţă aşa cum este definită în capitolul Definiţia cerinţei software:

1. o *condiţie* sau *capabilitate* necesar a fi îndeplinită de către un sistem, pentru ca *un utilizator să poată rezolva o anumită problemă* sau să atingă un anumit obiectiv; 2. o *condiţie* sau *capabilitate* pe care un sistem trebuie să o realizeze sau să o posede *pentru a satisface un contract, standard, specificaţie* sau alt document formal impus.

Aşadar, privită dintr-o parte cerinţa se vede ca o *problemă a unui client*, privită din cealaltă parte este o *soluţie furnizată de un anumit sistem*. De aceste două faţete ale ei depind caracteristicile de care vorbim în continuare.

Aceste caracteristici sunt, în principal, următoarele (depinzând de autor lista poate fi mai mare sau mai mică dar cele esenţiale sunt acestea):

1. Necesară

O cerinţă există *dacă şi numai dacă este necesară*. Amintind de prima parte a definiţiei de mai sus, putem spune că cerinţa există dacă există o problemă reală de la care porneşte. *În caz contrar cerinţa nu există*.

Această caracteristică este extrem de utilă pentru managementul cerinţelor, problema din spatele cerinţei fiind, în mod necesar, o frunză din decompoziţia problemei mari a proiectului (mai multe detalii în capitolul Nivelurile cerinţelor).

Doar pe baza problemei acesteia vom putea şti dacă o cerinţă se încadrează sau nu în proiect.

Pentru a demonstra că o cerinţă este necesară este nevoie de existenţa unei legături către cerinţele de pe nivelul superior (mai multe detalii în capitolul Nivelurile cerinţelor).

2. Corectă

Atunci când spunem că o cerinţă trebuie să fie corectă, ne apropiem deja de a doua parte a definiţiei de mai sus (sau mai degrabă le cuprindem pe amândouă). O cerinţă este corectă dacă faţeta denumită *soluţie* este, nimic altceva decât soluţia corectă la problema dată.

De exemplu, un *use case* care este gândit pentru realizarea unui anumit task este corect dacă înşiruirea paşilor descrişi conduce la realizarea, fără dubii, a task-ului.În caz contrar cerinţa descrisă astfel nu este corectă.

În general pentru a determina corectitudinea unei cerinţe este necesar să se facă referire la cerinţele de pe nivelul superior (mai multe detalii în capitolul Nivelurile cerinţelor).

3. **Completă**

O cerinţă este completă dacă reprezintă o soluţie completă pentru rezolvarea completă a problemei. Deşi cazurile de incompletitudine sunt greu de descoperit, organizarea de review-uri formale cu participarea oamenilor tehnici din echipa de dezvoltare, de obicei dă rezultate.

4. Consistentă

O cerinţă este considerată consistentă dacă nu intră în contradicţie cu altă cerinţă. De exemplu, următoarele cerinţe sunt inconsistente:

- autovehiculul se va deplasa cu viteza maximă de 100 km/h;
- autovehiculul va parcurge 200 de km în maximum o oră.

5. Verificabilă

O cerinţă este verificabilă (testabilă) dacă permite realizarea validarea fără echivoc a soluţionării ei prin măsurare sau testare. De exemplu, „sistemul va permite derularea optimă a activităţii”, „timpul de răspuns va fi cât mai mic cu putinţă” sau „sistemul va putea fi accesat de un număr mare de utilizatori simultan” sunt cerinţe care nu pot fi verificate în mod cert, fără dubii.

Oricând se poate pune întrebarea ce înseamnă derularea optimă a activităţii, când se poate spune că ţinta a fost atinsă?

6. Clară (fără ambiguităţi)

O cerinţă poate fi considerată lipsită de ambiguităţi atunci când poate fi interpretată într-un singur fel. Dacă mai mulţi cititori înţeleg lucruri diferite atunci cerinţa este ambiguă.

Pentru a ţine sub control fenomenul existenţei ambiguităţilor (axioma A4, care spune că niciodată oamenii implicaţi în proiect nu sunt perfecţi, ne spune că ele sunt inerente) trebuie organizate review-uri ale specificaţiei. De asemenea, specificaţiile vor fi folosite ca sursă primară pentru crearea planurilor de teste şi a manualului de utilizare.

7. Trasabilă

Trasabilitatea se referă la posibilitatea de a reface traseul pe care o cerinţă a luat naştere, pornind de la solicitarea iniţială a unui reprezentant al clientului. Acest mod de abordare asigură informaţia care justifică existenţa sau nu a cerinţei, precum şi posibilitatea de a reface drumul pe care a apărut o cerinţă, atunci când apar dubii asupra acesteia, asupra sursei sau asupra raţiunii ei.