----------BACKLOG (cerinte)----------

#Cerințele sunt descrieri ale serviciilor oferite de sistem și a constrângerilor sub care acesta va fi dezvoltat și va opera.

##Utilizatorii documentului cu cerinte: -*clienții*: impun cerințele și verifică apoi dacă acestea sunt conforme cu nevoile lor; -*managerii*: utilizează documentul pentru a stabili termenii contractului și a planifica procesul de producție; -*inginerii de sistem:* utilizează documentul pentru a înțelege ce trebuie dezvoltat; -*inginerii* de la testare: utilizează documentul pentru a proiecta testele de validare

#User story -> descrie o functionalitate care poate fi utila si valoroasa atat unui utilizator, cat si unui comparator al sistemului; unitate de baza intr-un proiect Agile; sufficient de mic pentru a putea fi implementat intr-o iteratie (Independent, Negotiable, Valuable, Estimable, Small, Testable)

##Cerinte functionale - Sunt afirmatii despre servicii pe care sistemul trebuie sa le contina, cum trebuie el sa raspunda la anumite intrari si cum sa reactioneze in anumite situatii.

-Descriu functionalitatea sistemului si serviciile oferite; -Depind de tipul softului, de utilizatorii avuti in vedere si de tipul sistemului pe care softul este utilizat; -Cerintele functionale ale utilizatorilor pot fi descrieri de ansamblu dar cerintele functionale ale sistemului trebuie sa descrie in detaliu serviciile oferite

##Cerinte non-functionale

Sunt constrângeri ale serviciilor si functiilor oferite de sistem cum ar fi: constrângeri de timp, constrângeri ale procesului de dezvoltare, standarde, etc. -Definesc proprietati si constrângeri ale sistemului, ca de exemplu: fiabilitatea, timpul de raspuns, cerintele pentru spatiul de stocare, cerinte ale sistemului de intrari-iesiri etc; -La intocmirea lor se va tine cont de un anumit mediu de dezvoltare; -Cerintele non-functionale pot fi mai critice decât cele functionale. Daca nu sunt indeplinite, sistemul nu va fi util scopului in care a fost dezvoltat.

####Tipuri -Cerinte ale produsului: Cerinte care specifica un anumit comportament al produsului, ca de exemplu: gradul de utilitate, eficienta (viteza de executie), fiabilitate, portabilitate etc. -**Cerinte legate de organizare**: Cerinte care sunt consecinte ale politicilor de organizare a productiei software, ca de exemplu: standarde utilizate, cerinte de implementare, cerinte de livrare etc. -**Cerinte externe**: Cerinte asociate unor factori externi, ca de exemplu: cerinte de interoperabilitate, cerinte legislative etc.

##Tipuri de cerinte

###Cerinte utilizator

-afirmatii in limbaj natural si diagrame a serviciilor oferite de system laolalta cu constrângerile operationale; -scrise pentru clienti

-trebuie sa descrie cerinte functionale si nonfunctionale intr-o maniera in care sunt pe intelesul utilizatorilor sistemului care nu detin cunostinte tehnice detaliate.

Se adreseaza: utilizatorilor finali, inginerilor clientului, proiectantilor de system, managerilor clientului, manager-ilor de contracte

###Cerintele sistemului

-un document structurat stabilind descrierea detaliata a functiilor sistemului, serviciile oferite si constrângerile operationale.

-poate fi parte a contractului cu clientul. Se adreseaza: utilizatorilor finali, inginerilor clientului-proiectantilor de system, programatorilor

##Structura documentului de specificare a cerintelor

-Prefata; -Introducere; -Glosar de termini; -Definirea cerintelor utilizatorilor; -Arhitectura sistemului; -Specificarea cerintelor de sistem; -Modelarea sistemului; -Evolutia sistemului; -Anexe; -Index

----------PROCESE DE DEZVOLTARE----------

#Procesul de dezvoltare cascada Modelul cascada trebuie folosit atunci cand cerintele sunt bine intelese si când este necesar un proces de dezvoltare clar si riguros.

##Etape

-analiza si definirea cerintelor: ce trebuie sa faca sistemul -design: cum trebuie sa se comporte sistemul

-implementare si testare unitara: designul sistemului este transformat intr-o multime de programe (unitati de program); testarea unitatilor de program verifica faptul ca fiecare unitate de program este conforma cu specificatia. -integrare si testare sistem: unitatile de program sunt integrate si testate ca un sistem complet; apoi acesta este livrat clientului.

-operare si mentenanta: sistemul este folosit in practica; mentenanta include: corectarea erorilor, imbunatatirea unor servicii, adaugarea de noi functionalitati.

##Avantaje -fiecare etapa nu trebuie sa inceapa inainte ca precedenta sa fie incheiata. -fiecare faza are ca rezultat unul sau mai multe documente care trebuie "aprobate" -bazat pe modele de proces folosite pentru productia de hardware (!! proces bine structurat, riguros, clar; produce sisteme robuste)

##Dezavantaje -dezvoltarea unui sistem software nu este de obicei un proces liniar; etapele se intrepatrund metoda ofera un punct de vedere static asupra cerintelor -schimbarile cerintelor nu pot fi luate in considerare dupa aprobarea specificatiei -nu permite implicarea utilizatorului dupa aprobarea specificatiei

----------Procesul de dezvoltare incremental----------

##Etape

-dezvoltarea si livrarea este realizata in parti (incremente), fiecare increment incorporând o parte de functionalitate

-cerintele sunt ordonate dupa prioritati (cele cu prioritatea cea mai mare fac parte din primul increment). -dupa ce dezvoltarea unui increment a inceput, cerintele pentru acel increment sunt inghetate, dar cerintele pentru noile incremente pot fi modificate.

##Avantaje -clientii nu trebuie sa astepte pâna ce intreg sistemul a fost livrat pentru a beneficia de el. Primul increment include cele mai importante cerinte, deci sistemul poate fi folosit imediat. -se micsoreaza riscul ca proiectul sa fie un esec deorece partile cele mai importante sunt livrate la inceput.

-deoarece cerintele cele mai importante fac parte din primele incremente, acestea vor fi testate cel mai mult.

##Dezavantaje -dificultati in transformarea cerintelor utilizatorului in incremente de marime potrivita. -procesul nu este foarte vizibil pentru utilizator

(nu e suficienta documentatie intre iteratii)

-codul se poate degrada in decursul ciclurilor

##Exemple: -Unified Process cu varianta Rational Unified Process;

-Peocese de dezvoltare in spurala introduse de Boehm;-Agile

#Metodologii agile

-se concentreaza mai mult pe cod decât pe proiectare

-se bazeaza pe o abordare iterativa de dezvoltare de software

-produc rapid versiuni care functioneaza, acestea evoluând repede pentru a satisface cerinte in schimbare.

##Aplicabilitate -companii mici sau mijlocii; -software pentru uz intern

##Dezavantaje -dificultatea de a pastra interesul clientilor implicati in acest proces de dezvoltare pentru perioade lungi; -membrii echipei nu sunt intotdeauna potriviti pentru implicarea intensa care caracterizeaza metodele agile; -prioritizarea modificarilor poate fi dificila atunci când exista mai multe parti interesate; -mentinerea simplitatii necesita o munca suplimentara; -contractele pot fi o problema ca si in alte metode de dezvoltare incrementala

##Exemple: -Extreme Programming (XP) -1996; -Adaptive Software Development (ASD); -Test-Driven Development (TDD); -Feature Driven Development (FDD); -Behavior Driven Developement (BDD); -Crystal Clear; -Scrum -1995

#Extreme programming -noile versiuni pot fi construite de mai multe ori pe zi; -acestea sunt livrate clientilor la fiecare 2 saptamâni;

-toate testele trebuie sa fie executate pentru fiecare versiune si o versiune e livrabila doar in cazul in care testele au rulat cu succes.

##Valorile XP: -Simplitate (Simplicity); -Comunicare (Communication); -Reactie (Feedback); -Curaj (Courage); -Respect (Respect)

##Practici: -procesul de planificare (The Planning Game); -client disponibil pe tot parcursul proiectului (On-Site Customer); -implementare treptata (Small Releases) -limbaj comun (Metaphor); -integrare continua (Continuous Integration); -proiectare simpla (Simple Design); -testare (Testing); -rescriere de cod pentru imbunatatire (Refactoring); -programare in pereche (Pair Programming); -drepturi colective (Collective Ownership) -40 ore/saptamâna (40-Hour Week); -standarde de scriere a codului (Coding Standard)

##Programarea in 2 -tot codul este scrisa de doua persoane folosind un singur calculator -sunt doua roluri in aceasta echipa: Unul scrie cod si celalalalt il ajuta gândindu-se la diverse posibilitati de imbunatatire.

###Avantajele programarii in 2 -sustine ideea de proprietate si responsabilitate in echipa pentru sistemul colectiv; -proces de revizuire imbunatatit, deoarece fiecare linie de cod este privita de catre cel putin doua persoane; -ajuta la imbunatatirea codului; -transfer de cunostinte si training implicit (important când membrii echipei se schimba)

##Avantaje XP: -solutie buna pentru proiecte mici; -programare organizata; -reducerea numarului de greseli -care pune in evidenta transmiterea de mesaje (sau apeluri de metode) de-a lungul timpului.

##Elemente

pentru ca toti sunt implicati in mod direct); -dispozitie la schimbare chiar in cursul dezvoltarii.

###Dezavantaje XP: -nu este scalabila; -necesita mai multe resurse umane "pe linie de cod"(d.ex. programare doi); -implicarea clientului in dezvoltare (costuri suplimentare si schimbari prea multe); -lipsa documentelor "oficiale" ;-necesita experienta in domeniu ("senior level" developers); -poate deveni uneori o metoda ineficienta (rescriere masiva de cod)

#SCRUM - o metodă "agilă", care se axează mai mult pe managementul dezvoltării incrementale, decât pe practici agile specifice.

-un proprietar de produs creeaza o lista de sarcini numita "**backlog**"; -apoi se planifica ce sarcini vor fi implementate in urmatoarea iteratie, numita "**sprint**"; -aceasta lista de sarcini se numeste "sprint backlog"; -sarcinile sunt rezolvate in decursul unui sprint care are rezervata o perioada relativ scurta de 2-4 saptamâni; -echipa se intruneste zilnic pentru a discuta progresul ("daily scrum"); -la sfârsitului sprintului, rezultatul ar trebui sa fie livrabil (adica folosit de client sau vandabil); -dupa o analiza a sprintului, se reitereaza.

----------UML----------

UML este un limbaj grafic pentru vizualizarea, specificarea, constructia si documentatia necesare pentru dezvoltarea de sisteme software (orientate pe obiecte) complexe.

#Motive pentru care UML nu e folosit: -Nu este cunoscuta notatia UML; -UML e prea complex (14 tipuri de diagrame); -Notatiile informale sunt suficiente; -Documentarea arhitecturii nu e considerata importanta

#Motive pentru care e folosit UML: -UML este standardizat; -existenta multor tool-uri; -flexibilitate: modelarea se poate adapta la diverse domenii folosind "profil-uri" si "stereotipuri"; -portabilitate: modelele pot fi exportate in format XMI (XML Metadata Interchange) si folosite de diverse tool-uri; -se poate folosi doar o submultime de diagrame

-arhitectura software e importanta

#Diagrama cazurilor de utilizare

##Elemente

-*Caz de utilizare* (componenta a sistemului): unitate coerenta de functionalitate sau task; reprezentata printr-un oval.

-*Actor* (utilizator al sistemului): element extern care interactioneaza cu sistemul; reprezentat printr-o figurina -Asociatii de comunicare**:** legaturi intre actori si cazuri de utilizare; reprezentate prin linii solide

-Descrierea cazurilor de utilizare: un document (narativ) care descrie secventa evenimentelor pe care le executa un actor pentru a efectua un caz de utilizare

###Actorii

-*Actorii primari* sunt cei pentru care folosirea sistemului are o anumita valoare (beneficiari);

-Actorii secundari sunt cei cu ajutorul carora se realizeaza cazul de utilizare; Actorii secundari nu initiaza cazul de utilizare, dar participa la realizarea acestuia.

###Cazuri de utilizare -Un caz de utilizare este o unitate coerenta de functionalitate. -Un caz de utilizare in-globeaza un set de cerinte ale sistemului care reies din specificatiile initiale si sunt rafinate pe parcurs. -Cazurile de utilizare pot avea complexitati diferite;

####Frontiera sistemului -este important de a defini frontiera sistemului astfel incât sa se poata face distinctie intre mediul extern si mediul intern (responsabilitatile sistemului) -ea poate avea un nume -cazurile de utilizare sunt inauntru, iar actorii in afara.

###Relatia << include >> -Daca doua sau mai multe cazuri de utilizare au o componenta comuna, aceasta poate fi reutilizata la definirea fiecaruia dintre ele. -În acest caz, componenta refolosita este reprezentata tot printr-un caz de utilizare legat prin relatia « include » de fiecare dintre cazurile de utilizare de baza.

###Relatia << extend >> Relatia « extend » se foloseste pentru separarea diferitelor comportamente ale cazurilor de utilizare. Daca un caz de utilizare contine doua sau mai multe scenarii semnificativ diferite (in sensul ca se pot intâmpla diferite lucruri in functie de anumite circumstante), acestea se pot reprezenta ca un caz de utilizare principal si unul sau mai multe cazuri de utilizare exceptionale.

###Relatia de generalizare Acest tip de relatie poate exista atât intre doua cazuri de utilizare cât si intre doi actori. -generalizarea intre cazuri de utilizare indica faptul ca un caz de utilizare poate mosteni comportamentul definit in alt caz de utilizare. generalizarea intre actori arata ca un actor mosteneste structura si comportamentul altui actor.

(!!!! NU EXISTA linii simple intre cazuri)

#Diagrame de secvente Este tipul de diagrama UML -care pune in evidenta transmiterea de mesaje (sau apeluri de metode) de-a lungul timpului.

##Elemente

-Obiectele si actorii sunt reprezentati la capatul de sus al unor linii punctate, care reprezinta linia de viata a obiectelor.

-Scurgerea timpului este reprezentata in cadrul diagramei de sus in jos.

-Un *mesaj* se reprezinta printr-o sageata de la linia de viata a obiectului care trimite mesajul la linia de viata a celui care-l primeste.

-Timpul cât un obiect este activat este reprezentat printr-un dreptunghi subtire care acopera linia sa de viata.

-Optional, pot fi reprezentate raspunsurile la mesaje printr-o linie punctata, dar acest lucru nu este necesar.

###Mesaje -*sincron* (sau apel de metoda). Obiectul pierde controlul pâna primeste raspuns -*de raspuns*: raspunsuri la mesajele sincrone; reprezentarea lor este optionala. -*asincron*: nu asteapta raspuns, cel care trimite mesajul ramânând activ (poate trimite alte mesaje).

#Diagrame de clase Diagramele de clase sunt folosite pentru a specifica structura statica a sistemului, adica: ce clase exista in sistem si care este legatura dintre ele.

##Elemente

###Atribute de vizibilitate

publice ``+``; *private* ``-``; protejate ``#``; package ``~`` -> exemplu array: nume: String[1..2];

###Operatii Semnatura unei operatii este formata din: numele operatiei, numele si tipurile parametrilor (daca e cazul) si tipul care trebuie returnat (daca este cazul).

###Relatii intre clase

-asociere. Asocierile sunt legaturi structurale intre clase. Între doua clase exista o asociere atunci când un obiect dintr-o clasa interactioneaza cu un obiect din cealalta clasa. Dupa cum clasele erau reprezentate prin substantive, asocierile sunt reprezentate prin verbe.

-multiplicitati (1..*/1/2..5 etc);

-navigabilitate(A->B => A stocheaza o referita la B)

-agregare si compunere

---agregare: o clasa este „intregul" care contine parti mai mici

---compunere(mai puternica): partile componente nu exista fara intreg

-generalizare: relatie intre un lucru general (numit superclasa) si un lucru specializat (numit subclasa)

-dependenta:A depinde de B daca o modificare din B schimba si A(ex: A are o metoda cu param de timp B)

-interfata(realizare)

#Diagrame de stari (numite si masini de stare sau state-charts) descriu dependenta dintre starea unui obiect si mesajele pe care le primeste sau alte evenimente receptionate.

##Elemente

-stari, reprezentate prin dreptunghiuri cu colturi rotunjite. O stare este o multime de configuratii ale obiectului care se comporta la fel la aparitia unui eveniment(evenimente asociate starilor: entry[cand ob intra in starea resp], exit[cand iese], do[in timp ce se afla in starea respectiva])

-tranzitii intre stari, reprezentate prin sageti

-evenimente care declanseaza tranzitiile dintre stari (cel mai des întâlnite evenimente sunt mesajele primite de catre obiect)

-semnul de inceput, reprezentat printr-un un disc negru dincare porneste o sageata (fara eticheta) spre starea initiala a sistemului.

-semne de sfârsit, reprezentate printr-un disc negru cu un cerc exterior, in care sosesc sageti din starile finale ale sistemului. Acestea corespund situatiilor in care obiectul ajunge la sfârsitul vietii sale si este distrus.

-garzi(în anumite situații un eveniment declanșează o tranziție numai dacă atributele obiectului îndeplinesc o anumită condiție suplimentară (gardă)): eveniment[garda]/actiune

----------ARHITECTURA ----------

-împărțirea optimă a unui sistem complex în diverse componente, evidențiind relațiile dintre acestea; - document ce se adreseaza atat clientului / beneficiarul aplicatiei, cat si echipei de dezvoltare; - se creaza o legatura intre cerintele logice ale aplicatiei si posibilitatile tehnologice; - arhitectura descrie structura sistemului, fara a expunde informatii detaliate despre implementare; - raspunde cerintelor de functionalitate

#Atribute de calitate: -*performanță*: trebuie paralelizat cat mai mult, descompunând sistemul în procese cooperante; -*precizie*: trebuie optimizată structura datelor și modul în care valorile sunt prelucrate; -*securitate*: trebuie gestionate bine restricțiile de comunicare și acces;

-*portabilitate și reutilizare*: trebuie minimizate dependențele puternice între componente

#MVC - software design pattern utilizat in implementarea aplicatiilor cu interfata care utilizatori, de regula applicatii web (Java, C#, Ruby, PHP)

##Model - Componenta software ce reprezinta resursele conceptuale puse la dispozitie de catre aplicatie

##View - Construieste interfata prezentata utilizatorilor, pe baza cerintelor transmise de catre controller sau intero-gand direct modelele

##Controller - Conponenta logica a aplicatiei, ce intercepteaza cererile clientilor, interogheaza baza de date prin intermediul modelelor, construieste un raspuns utilizand view-urile.

#Avantaje: -Decupleaza componentele aplicatiei, ce pot fi ulterior mai usor de realizat, modificat sau inlocuit; -Modularizeaza logic aplicatia, previne duplicarea codului sursa; -Permite crearea de interfete multiple pentru utilizatori, ale acelorasi date sau logica a aplicatiei (in general interfetele unei aplicatii se modifica mai des decat baza de date sau logica aplicatiei). -Datorita tehnologiilor diferite si utilizarea specifica a componentelor MVC, acestea pot fi dezvoltate si intretinute de o echipa specializata pe fiecare component.

#Dezavantaje: -Creste complexitatea aplicatiei, pe nivele; - Separa fiecare functionalitate pe mai multe nivele; - Necesita o perioada de adaptare mai mare a dezvoltatorilor, precum si aptitudini de baza pe toate cele trei nivele

----------SISTEME DE VERSIONARE----------

- Instrument de dezvoltare software utilizat în gestionarea multiplelor versiuni ale fișierelor și dependențelor unei aplicații, înregistrând toate stările acestora, inclusiv modificări, autori și comentarii privind fiecare modificare (necessitate: securitate-repository securizat, lucrul in echipa, istoria proiectului, integrare cu project trackers)

#Concepte

-Repository - server de fișiere (bază de date) unde sunt stocate datele proiectului software; *-Working copy* - versiunea curentă a proiectului; *-Commit* - modificări efectuate asupra unor fișiere, publicate în repository; *-Revision* - versiune a unui fișier, ca urmare a unui commit; *-Branch* - copie separată a proiectului, ce poate conține modificări individuale; *-Merge* - operație de combinare a modificărilor din branch-uri separate; *-Checkout / fetch / pull* - descărcare modificări de pe repository

#Particularitati Git: salveaza patches (diferente) pentru fiecare commit, local branches folosite intensive, operatiuni de merge, rebase, fork mult mai facile decat in SVN

#Instrumente de build: GNU Make(C/C++), Ant(Java), Maven, Gradle(Java, Scala, C/C++, Android)

-Build automation este procesul de creare automată a componentelor software pornind de la codul sursă și dependențe

-Fazele uzuale ale procesului de build:, Instalare dependențe, Compilare cod sursă în cod binar, Împachetare cod binar, Rulare aplicație, Rulare teste automate, Revenire la starea inițială

--------MEDII INTEGRATE DE DEZVOLTARE (IDE) --------

#I.D.E = Integrare, Dezvoltare, Environment

##Integrare

- O serie de instrumente de dezvoltare (tool-chains) – în marea lor majoritate la linia de comandă: Compilator, Debugger, Build-tool, Profiler, Version contol tool, Generatoare de cod/resurse, Dezasamboare/decompilatoare etc.; - Unul sau mai multe navigatoare cum ar fi: file manager, project manager, class manager, proprieties, widgets etc; - Unul sau mai multe editoare de text/resurse specializate în recunoașterea formatului fișierelor editate; - Unul sau mai multe console folosite pentru a afișa starea proiectului, sau a operațiilor care se execută supra acestuia

###Dezavantaj integrare - se pierde generalitatea unor componente. De exemplu un "file-manager" transformat într-un "project-navigator" nu mai permite vizualizarea tuturor fișierelor dintr-o structură de directoare, ci numai a acelor fișiere care aparțin logic de proiect;

###Avantaj integrare - se câștigă funcționalități noi. De exemplu, un editor de cod Java va avea: "syntax high-lighting", permite indentarea după un anumit code-style predefinit etc;

###Functionalitati noi aparute : Visual profiling, Visual diffs, Visual preview, Refactorizare.

##Dezvoltare (creare sau de programare a aplicațiilor)

###Deficiente: -Sunt foarte multe și au tendința de a crește ca număr în timp, de la o versiune la alta (a se vedea numărul de tool-uri în tool-chain-urile din JDK sau de la GNU); -Modul de utilizare al acestor instrumente variază în funcție de tipul de aplicație și/sau de limbajul de programare folosit; -Fiecare instrument de programare are foarte multe opțiuni de execuție, necesită multe operații manuale și un timp mare de învățare; -Nu permit refactorizarea și navigarea ușoară a codului în interiorul unui proiect; -Debugging-ul, profilingul sau compararea surselor la linia de -comandă este foarte complicată; -Unele sisteme de operare (Cum ar fi MAC OS sau Windows) descurajează folosirea liniei de comandă;

###Rezolvari: -Același look&feel pentru proiecte din aceeași clasă de aplicații. De exemplu: pentru Java,

indiferent dacă build-tool-ul folosit e Ant, Maven sau Gradle, proiectul va arăta la fel în IDE, iar operațiile de bază (clean, compile, run, debug) vor avea aceeași funcționalitate;
-Același look&feel pentru debugging (De exemplu: degugger-ul din NetBeans arată la fel indiferent de limbajul de programare folosit (C/C++, Java sau PHP) și de natura proiectului (aplicații stand-alone, mobile sau web).) In plus debugger-ul din IDE are o ergonomie ridicată datorită folosirii ferestrelor; -Operațiile de Search&Replace și mai ales Refactorizare funcționează atât la nivelul unui fișier sursă cât și la nivelul întregului proiect etc.
##Environment - un sistem hardware-software (format din unul sau mai multe calculatoare și din una sau mai multe aplicații auxiliare) în care o aplicație este deployată și apoi executată;

----------REFACTORIZARE----------
-Refactorizarea codului sau „code refactoring" este procesul de modificare a unei secvențe de program fără a-i schimba funcționalitatea externă
-Refactorizarea vizează în principal îmbunătățirea structurală a unui codul existent – cod care a fost testat și validat anterior - având ca obiective finale:
---Reducerea complexității;
---Creșterea lizibilității și implicit a mentenabilității codului;
---Modificarea internă a acestuia în vederea extinderii codului cu noi opțiuni (capacitatea codului de a fi extensibil);
-Refactorizarea poate produce un cod diferit de la caz la caz, în funcție de obiectivul urmărit. De exemplu dacă obiectivul urmărit este:
----Reducerea complexității => poate produce mai multe componente simple;
----Creșterea lizibilității și implicit a mentenabilității codului => o serie de componente simple pot fi grupate în expresii/funcții/clase mai complexe, dar mai elegante, care pot fi utilizate mai ușor în diferite părți ale aplicației;
----Modificarea internă a acestuia în vederea extinderii codului cu noi opțiuni => apar noi nivele de abstractizate situate deasupra nivelelor deja existente
#Tehnici de refactorizare a codului
1. Tehnici pentru îmbunătățirea denumirii și localizării codului:
--Mutarea definiției unei variabile, constante, funcție, etc. într-un fișier (sau într-o clasă) care ilustrează mai bine apartenența funcțională de aceasta; --Redenumirea unei variabile, constante, funcție, etc. într-un fișier (sau într-o clasă) care să reflecte mai bine scopul/utilitatea acesteia; -- Pull Up/Push Down – cazuri particulare de mutare folosite în OOP pentru a evidenția deplasarea unor membrii în cadrul ierarhiei de clase (mutare într-o super clasă respectiv mutare într-o sub clasă).
2. Tehnici pentru „spargerea" codului în secțiuni logice distincte:
--Componentizarea sau spargerea codului în unități semantice reutilizabile reprezentate de interfețe care sunt mai clare, mai bine definite și mai simplu de utilizat; --Extract class – mutarea unei părți de cod dintr-o clasă într-o clasă nouă; --Extract field/constant/method – extragerea unei valori (sau a unei secțiuni de cod) care se repetă într-o nouă variabilă sau funcție.
3. Tehnici care permit creșterea gradului de abstractizare a codului:
--Encapsulate Field – așa cum îi spune și numele permite încapsularea unui câmp și forțează utilizatorul să folosească accesori de tip getter și setter în locul accesării directe; --Generalize type – permite, în anumite condiții schimbarea tipului de date al unui câmp cu un tip de date cu un grad de generalizare mai ridicat (de exemplu: de la List la Collection);
#Bune practici pentru refactorizare: Restaurează codul inițial atunci când refactorizarea eșuează (folosind un version control system); Creează-ți un scenariu de testare (sau mai bine, o întreagă suită de teste) înainte de a realiza prima operație de refactorizare; Refactorizează în pași cât mai mici; Testează modificările după fiecare refactorizare; Refactorizează codul automat (folosind un IDE) și nu refactoriza manual decât în situații excepționale; Nu combina în același pas refactorizarea cu bug-fixing-ul și/sau cu extinderea funcționalității

----------INSPECTIA CODULUI----------
-O metodă importantă pentru asigurarea calității este citirea
codului cu scopul de a detecta erori
-O echipa de inspecție/recenzie a codului are patru membri:
----Moderatorul – un programator competent
----Programatorul – cel care a scris codul inspectat
----Designer-ul, dacă este o persoană diferită de programator
----Un specialist în testare
-Inspectorii detecteaza erorile, programatorul trebuile sa le corecteze; erorile determinate pot conduce la modificarea design-ului; de obicei o ispectie dureaza 90-120 de min, cu o rata de 150 de instructiuni/ore
-Aspecte care pot fi luate in considerare: formatul codului, stilul programarii, ce fel de nume sunt folosite, acoperirea cu teste a codului inspectat
#Avantaje: descoperire de bug-uri, cod scris mai bine de la inceput, transfer de cunostinte, solutii mai bune la o anumita problema
#Dezavantaje: creste timpul investit in dezvoltarea de cod, stres suplimentar asupra dezvoltatorilor care prefera sa lucreze singuri, uneori e suficient ca o singura persoana sa isi faca singura inspectia codului si sa gaseasca problemele

----------DEPANAREA PROGRAMELOR----------
#Defect software (BUG): o eroare, o omisiune, o neintelegere sau un esec etc. *-Consecinte*: producerea unui rezultat incorect sau neașteptat, producerea unui rezultat corect însă însoțit de o serie de comportamente neașteptate sau neintenționate
##Efectele defectelor software
-Efecte subtile-funcționalitatea software-ului pare a fi corectă. Totuși, dacă acestea sunt lăsate să ruleze pentru o perioadă mai lungă de timp, atunci aceste efecte subtile se vor acumula ducând în final la efecte vizibile; *-Efecte tranzitorii*-funcționalitatea software-ului este afectată pe termen scurt după care aceasta revine la normal. Aceste defecte poartă numele de glitch-uri și sunt uneori deosebit de greu de reprodus și mai ales de reparat. Exemple: defecte de timing, defecte de initializare, defecte datorate erorilor de comunicatie; *-Efecte vizibile*-funcționalitatea software-ului este afectată în general de o serie de factori externi cum ar fi: o combinatie a datelor de intrare, o secventa de comenzi si actiuni, o configuratie hardware particulara; se manifesta prin blocare(freeze) sau crash; in aceasta categorie intra majoritatea bug-urilor; *-Efecte secundare*-funcționalitatea software-ului nu este afectată. Ceea ce este afectat în acest caz este stabilitatea și/sau securitatea sistemului hardware/software pe care aplicația în cauză este executată (exemplu: acapararea in mod nejustificat a unor resurse software sau hardware)
##Modalitati de depanare
-folosind tiparirea-simplu de aplicat, nu necesita tool-uri; codul se complica, performanta scade uneori, sunt necesare recompilari repetate etc; *-folosind log-uri(istoria executiei progr)*-poate fi controlat prin progr sau proprietati(codul se complica); *-debugger (JAVA in ECLIPSE)*
```
public class BinarySearch {
// presupunem ca vectorul array e ordonat crescator!
public static int binarySearch(int array[], int target){
  int low = 0, high = array.length, mid;
  while (low <= high) {
    mid = (low + high)/2;
    if ( target < array[ mid ] )
      high = mid - 1;
    else if ( target > array[ mid ])
      low = mid + 1;
    else return mid; }
  return -1; }
```
Rulăm câteva teste pentru funcția de căutare:
-binarySearch({1,2,3}, 1) == 0 OK
-binarySearch({1,2,3}, 4) aruncă ArrayIndexOutOfBoundsException
TESTE IMPLEMENTATE IN JUNIT
```
public class BinarySearchTest {
@Test
public void test_1() {
int[] a = {1, 2, 3};
int x = 1;
int result = BinarySearch.binarySearch(a,x);
assertTrue(result == 0); }
@Test
public void test_2() {
int[] a = {1, 2, 3};
int x = 4;
int result = BinarySearch.binarySearch(a,x);
assertTrue(result == -1); }
```
#Debugging/Debuggers
Un debugger este o aplicație software folosită pentru a testa și a depana o altă aplicație
##Functionalitatile unui debugger:
-controlul execuției: poate opri execuția la anumite locații numite breakpoints
-interpretorul: poate executa instrucțiunile una câte una
-inspecția stării programului: poate observa valoarea variabilelor, obiectelor sau a stivei de execuție
-schimbarea stării: poate schimba starea programului în timpul execuției
##Breakpoint = punct in cadrul unui program folosit pentru a-i opri executia in acel loc
#Depanare sistematica (utila deoarece datele pot fi mari, programele pot avea mii de locatii de memorie etc)

----------TESTARE----------
#Verificare = se refera la dezvoltarea produsului
#Validare = se refera la respectarea specificatiilor, utilitatea produsului
#Evaluarea unui produs - depinde de functionalitate programului, asteptarile utilizatorilor, mediul de afaceri
#Terminologie IEEE ---*Eroare* (engl. "error") - o acțiune umană care are ca rezultat un defect în produsul software; ---*Defect* (engl. "fault") - consecința unei erori în produsul software- un defect poate fi latent: nu cauzează probleme cât timp nu apar condițiile care determină execuția anumitor linii de cod; ---*Defecțiune* (engl. "failure") - manifestarea unui defect: când execuția programului întâlnește un defect, acesta provoacă o defecțiune abaterea programului de la comportamentul așteptat
#Principii de testare -o parte necesara a unui caz de test este definirea iesirii saurezultatului asteptat; -programatorii nu ar trebui sa-si testeze propriile programe(exceptie face testarea de nivel foarte jos - testarea unitara); -organizatiile ar trebui sa foloseasca si companii (sau departamente) externe pentru testarea propriilor programe; -rezultatele testelor trebuie analizate amanuntit -trebuie scrise cazuri de test atât pentru conditii de intrare invalide si neasteptate, cât si pentru conditii de intrare valide si asteptate.
#Testare unitara (unit testing) -o unitate (sau un modul) se referă de obicei la un element atomic (clasă sau funcție), dar poate însemna și un element de nivel mai înalt; -testarea unei unități se face în izolare
--> Un test contine: initializarea(clasei sau arg necesare), apelul metodei testate, decizia daca testul a reusit sau a esuat(compara valorile produse de metoda cu cele corecte)
#Testarea de integrare (integration testing) -Testează interacțiunea mai multor unități; -Testarea este determinată de arhitectură
#Testarea sistemului (system testing) -testarea sistemului testează aplicația ca întreg și este determinată de scenariile de analiză; -aplicația trebuie să execute cu succes toate scenariile pentru a putea fi pusă la dispoziția clientului; -spre deosebire de testarea internă și a componentelor, care se face prin program, testarea aplicației se face de obicei cu scripturi care rulează sistemul cu o serie de parametri și colectează rezultatele; -testarea aplicației trebuie să fie realizată de o echipă independentă de echipa de implementare; -testele se bazează pe specificațiile sistemului
#Testarea de acceptanta (acceptance testing) -testele de acceptanță determină dacă sunt îndeplinite cerințele unei specificații sau ale contractului cu clientul.
#Testarea de regresie (regression testing) -un test valid generează un set de rezultate verificate, numit "standardul de aur"; -testele de regresie sunt utilizate la re-testare, după realizarea unor modificări, pentru a asigura faptul că modificările nu au introdus noi defecte în codul care funcționa bine anterior; -pe măsură ce dezvoltarea continuă, sunt adăugate alte 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; -acest mecanism previne regresia sistemului într-o stare de eroare anterioară.
#Testarea performantei (performance testing)
-O parte din testare se concentrează pe evaluarea proprietăților
non-funcționale ale sistemului, cum ar fi:
---siguranța ("reliability") - menținerea unui nivel specificat de performanță
---securitatea - persoanele neautorizate să nu
#Testarea la încărcare (load testing) -asigură faptul că sistemul poate gestiona un volum așteptat de date, similar cu acela din locația-destinație (de exemplu la client); -verifică eficiența sistemului și modul în care scalează acesta pentru un mediu real de execuție;
#Testarea la stres (stress testing)
-solicită sistemul dincolo de încărcarea maximă proiectată
- supraîncărcarea testează modul în care „cade" sistemul sistemele nu trebuie să eșueze catastrofal testarea la stres verifică pierderile inacceptabile de date sau funcționalități
-deseori apar aici conflicte între teste. Fiecare test funcționează
corect atunci când este făcut separat. Când două teste sunt rulate în paralel, unul sau ambele teste pot eșua
---cauza este de obicei managementul incorect al accesului la
resurse critice (de exemplu, memoria)
-o altă variantă, "soak testing", presupune rularea sistemului pentru o perioadă lungă de timp (zile, săptămâni, luni)
---în acest caz, de exemplu scurgerile nesemnificative de memorie se pot acumula și pot provoca căderea sistemului
#GUI testing -testarea interfeței cu utilizatorul presupune memorarea
tuturor acestor informații și elaborarea unei modalități prin care mesajele să fie trimise din nou aplicației, la un moment ulterior; -de obicei se folosesc scripturi pentru testare
#Testarea utilizabilității (usability testing) -testează cât de ușor de folosit este sistemul; -se poate face în laboratoare sau „pe teren" cu utilizatori din lumea reală; -
exemple de metode folosite: testare "pe hol" (hallway testing): cu câțiva utilizatori aleatori, testare de la distanță: analizarea logurilor utilizatorilor, recenzii ale unor experți (externi)
#Testarea de tip "cutie neagră" (functionala)
Se iau in considerare numai intrarile (intr-un modul, componenta sau sistem) si iesirile dorite, conform specificatiilor structura interna este ignorata (de unde si numele de "black box testing")
#Exemple de metode de testare de tip black box
###Partitionare in clase de echivalenta
-ideea de bază este de a partiționa domeniul problemei (datele de intrare) în partiții de echivalență sau clase de echivalență astfel încât, din punctul de vedere al specificației, datele dintr-o clasă să fie tratate în mod identic -după ce clasele au fost identificate, se alege o valoare din fiecare clasă. În plus, pot fi alese și date invalide (care sunt în
afara claselor și nu sunt procesate de nici o clasă)
####Avantaje -reduce drastic numarul de date de test doar pe baza specificatiei; -potrivita pentru aplicatii de tipul procesarii datelor, in care intrarile si iesirile sunt usor de identificat si iau valori distincte
####Dezavantaje -modul de definire a claselor nu este evident (nu exista nici o modalitate riguroasa sau macar niste indicatii clare pentru identificarea acestora). -in unele cazuri, desi specificatia ar putea sugera ca un grup de valori sunt procesate identic, acest lucru nu este adevarat. (Acest lucru intareste ideea ca metodele de tip "cutie neagra" trebuie combinate cu cele de tip "cutie alba".) -mai putin aplicabile pentru situatii când intrarile si iesirile sunt simple, dar procesarea este complexa.
##Analiza valorilor de frontiera -este folosită de obicei împreună cu partiționarea în clase de echivalență; -ea se concentrează pe examinarea valorilor de frontieră ale claselor, care de regulă sunt o sursă importantă de erori; -această metodă adaugă informații suplimentare pentru generarea setului de date de test
###Partiționarea în categorii (category-partition) -se bazează pe cele două anterioare; -ea încearcă să genereze date de test care "acoperă"
funcționalitatea sistemului și astfel, să crească posibilitatea de
găsire a erorilor.
####Avantaje si dezavantaje-pașii de început (identificarea parametrilor și a condițiilor de mediu precum și a categoriilor) nu sunt bine definiți, bazându-se pe experiența celui care face testarea. Pe de alta parte, odată ce acești pași au fost realizați, aplicarea metodei este clară; -este mai clar definită decât metodele "cutie neagră" anterioare și poate produce date de testare mai cuprinzătoare, care testează funcționalități suplimentare; pe de altă parte, datorită exploziei combinatorice, pot rezulta seturi de teste de foarte mari dimensiuni.
#Testarea de tip cutie alba (structurala)
Testarea de tip „cutie alba" ia in calcul codul sursa al metodelor testate. Vizeaza acoperirea diferitelor structuri ale programului.
Programul este modelat sub forma unui graf orientat.
##Acoperire la nivel de instructiune- Fiecare instructiune (sau nod al grafului) este parcursa macar o data.
###Avantaje -realizeaza executia macar o singura data a fiecarei
instructiuni; -in general, usor de realizat.
###Dezavantaje: -nu testeaza fiecare conditie in parte in cazul
conditiilor compuse (de exemplu, pentru a se atinge o acoperire la nivel de instructiune in programul anterior, nu este necesara introducerea unei valori mai mici ca 1 pentru n); -nu testeaza fiecare ramura; -probleme suplimentare apar in cazul instructiunilor if a caror clauza else lipseste. In acest caz, testarea la nivel de instructiune va forta executiei ramurii corespunzatoare valorii "adevarat", dar, deoarece nu exista clauza else, nu va fi necesara si executia celeilalte ramuri. Metoda poate fi extinsa pentru a rezolva aceasta problema.
##Acoperire la nivel de ramura -Fiecare ramura a grafului este parcursa macar o data. Dezavantaj: nu testeaza conditiile individuale ale fiecarei decizii.
##Acoperire la nivel de cale -Genereaza date pentru executarea fiecarei cai macar o singura data.
##Acoperire la nivel de conditie -Genereaza date de test astfel incât fiecare conditie individuala dintr-o decizie sa ia atât valoarea adevarat cât si valoarea fals (daca acest lucru este
posibil).
###Avantaje -se concentreaza asupra conditiilor individuale
###Dezavantaje -poate sa nu realizeze o acoperire la nivel de
ramura. Pentru a rezolva aceasta slabiciune se poate folosi testarea la nivel de conditie/decizie.
##Acoperire la nivel de conditie/decizie - Genereaza date de test astfel incât fiecare conditie individuala dintr-o decizie sa ia atât valoarea adevarat cât si valoarea fals (daca acest lucru este
posibil) si fiecare decizie sa ia atât valoarea adevarat cât si valoarea fals.
##Acoperirea MC/DC -fiecare conditie individuala dintr-o decizie ia atât valoarea true cât si valoarea false -fiecare decizie ia atât valoarea true cât si valoarea false -fiecare conditie individuala influenteaza in mod ndependent decizia din care face parte
###Avantaje: -acoperire mai puternica decât acoperirea conditie/decizie simpla, testând si influenta conditiilor individuale asupra deciziilor -produce teste mai putine – depinde liniar de numarul de conditii
#Metrici in testarea software: -no. of requirements, total no. of test cases written for all requirements, total no. of test cases executed/passed/failed/blocked, total no. of defects identified, critical/ high/ medium/ low defects count

----------PERFORMANTA ----------
#Software
-Capacitatea unui sistem software de a executa sarcinile pentru care a fost proiectat, în timpul și condițiile de lucru prestabilite. Sistemul software implementat întrunește un set de condiții de lucru optimale, ce includ capacitate de procesare, timpi de execuție sau medii de execuție, cunoscute de către utilizatori.
#Hardware
Sisteme distribuite ce inglobeaza puterea de calcul a mai multor calculatoare obisnuite aflate in aceeasi retea / Internet. Aceastea executa task-uri individuale simple al caror rezultat poate fi combinat
in scopul finalizarii unei sarcini complicate. Clustere de calculatoare / supercomputers - sisteme dedicate, instalate in aceeasi locatie, ce pot lucra impreuna pentru a executa task-uri in paralel
#Criterii de masurare a performatei aplicatiilor: -Capacitate de procesare a sarcinilor (workload); -Viteza de procesare a sarcinilor (throughput); -Resurse consumate (memorie, CPU, network, disk I/O, ...etc.); -Over-head elemente de securitate

----------SABLOANE(DESIGN PATTERNS)----------
-soluții generale reutilizabile la probleme care apar frecvent în roiectare (orientată pe obiecte)
#Folositoare in urmatoarele feluri: -ca mod de a învăța practici bune; -aplicarea consistentă a unor principii generale de proiectare; -ca vocabular de calitate de nivel înalt (pentru comunicare); -ca autoritate la care se poate face apel;-în cazul în care o echipă sau organizație adoptă propriile șabloane: un mod de a explicita cum se fac lucrurile acolo
#Trebuie folosite cu grija deoarece: -sunt folosite doar dacă există într-adevăr problema pe care ele o rezolvă; -pot crește complexitatea și scădea performanța
#Tipuri: arhitecturale, de proiectare, idiomuri
#Principii de baza: -programare folosind multe interfețe: interfețe și clase abstracte pe lângă clasele concrete, framework-uri generice în loc de soluții directe; se preferă compoziția în loc de moștenire(delegarea către obiecte -se urmărește decuplarea: obiecte cât mai independente, folosirea "indirecției", obiecte "ajutătoare".
#Sabloane creationale: singleton, abstract factory, builder
#Sablon structural: facade
#Sabloane comportamentale: observer, visitor
#Anti-sabloane: BaseBean, God object, yoyo-problem, race hazzard, input kludge etc.