#Procese de dezvoltare software
##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: Sunt stabilite serviciile, constrângerile si scopurile sistemului prin consultare cu utilizatorul. (ce trebuie sa faca sistemul).
-design: Se stabileste o arhitectura de ansamblu si functiile sistemului software pornind de la cerinte. (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
Avantaj: 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
-sunt identificate cerintele sistemului la nivel inalt, dar, in loc de a dezvolta si livra un sistem dintr-o data, dezvoltarea si livrarea este realizata in parti (incremente), fiecare increment incorporând o parte de functionalitate.
-cerintele sunt ordonate dupa prioritati, astfel încât cele cu prioritatea cea mai mare fac parte din primul increment, etc.
-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.
-primele incremente pot fi prototipuri din care se pot stabili cerintele pentru urmatoarele incremente.
-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.
###Cele 12 principii
1. Prioritatea noastra este satisfactia clientului prin livrarea rapida si continua de software valoros.
2. Schimbarea cerintelor este binevenita chiar si intr-o faza avansata a dezvoltarii. Procesele agile valorifica
schimbarea in avantajul competitiv al clientului.
3. Livrarea de software functional se face frecvent, de preferinta la intervale de timp cât mai mici, de la câteva
saptamâni la câteva luni.
4. Clientii si dezvoltatorii trebuie sa colaboreze zilnic pe parcursul proiectului.
5. Construieste proiecte in jurul oamenilor motivati. Ofera-le mediul propice si suportul necesar si ai incredere ca obiectivele vor fi atinse.
6. Cea mai eficienta metoda de a transmite informatii inspre si in interiorul echipei de dezvoltare este comunicarea fata in fata.
7. Software functional este principala masura a progresului.
8. Procesele agile promoveaza dezvoltarea durabila. Sponsorii, dezvoltatorii si utilizatorii trebuie sa poata
mentine un ritm constant pe termen nedefinit.
9. Atentia continua pentru excelenta tehnica si design bun imbunatateste agilitatea.
10. Simplitatea – arta de a maximiza cantitatea de munca nerealizata – este esentiala.
11. Cele mai bune arhitecturi, cerinte si design se obtin de catre echipe care se auto-organizeaza.
12. La intervale regulate, echipa reflecteaza la cum sa devina mai eficienta, apoi isi adapteaza si ajusteaza omportamentul in consecinta.
###Aplicabilitate
-companii mici sau mijlocii
-software pentru uz intern
###Dezavantaje
-dificultatea de a pastra interesul clientilor implicati in acest procesul 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)
###Programrea 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 programrii 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 ("four eyes principle").
-ajuta la imbunatatirea codului.
-transfer de cunostinte si training implicit (important când membrii echipei se schimba)
-"more fun"?
###Avantaje
-solutie buna pentru proiecte mici
-programare organizata
-reducerea numarului de greseli
-clientul are control (de fapt, toata lumea are control, pentru ca toti sunt implicati in mod direct)
-dispozitie la schimbare chiar in cursul dezvoltarii
###Dezavantaje
-nu este scalabila
-necesita mai multe resurse umane "pe linie de cod"(d.ex. programare in 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
-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"). Ceremoniile sunt conduse de un "scrum master".
-la sfârsitul sprintului, rezultatul ar trebui sa fie livrabil (adica folosit de client sau vandabil).
-dupa o analiza a sprintului, se reitereaza.

#Z
Este bazat pe:
-logica de ordin 1, cu predicate
-teoria multimiilor
-notatii auxiliare
##Specificatii formale
O specificatie formala:
-foloseste notatii matematice pentru a descrie intr-un model precis ce proprietati trebuie sa aiba un sistem
-descrie ce trebuie sa faca sistemul si nu cum
-independent de cod
-poate fi folosita pentru intelegerea cerintelor si analiza lor (uneori se poate si genera cod dintr-o specificatie suficient de precisa)
În Z descompunerea unei specificatii se face in mai multe piese numite *scheme*.
##Operatori logici
-neg ¬ -negatie
-$\wedge$ ^ -conjunctie
-$\vee$ V -disjunctie
-$\implies$ -implicatie
-$\iff$ -echivalenta
##Egalitate
- = -egalitate
##Cuantificare
$Q x_1: S_1 ; \dots x_n : S_n | p \bullet q$ unde Q este $\forall, \exists, \exists_1$
Sensul este $\forall x_1: S_1 ; \dots x_n : S_n (p \implies q)$
##Multimi
###Multimi prin enumerare
$\{ e_1, \dotsc, e_n\}$ unde elementele au tipuri compatibile
###Multimi definite prin proprietati
$\{ x: T | pred(x) \bullet expr(x) \}$ reprezinta toate elementele care rezulta evaluand expr(x) pentru toti x de tip T care starisfac pred(x).
###Operatii pe multimi
-apartenenta
-submultime
-Operatorul de generare a submultimilor
-Produs cartezian
-reuniune
-Intersectie
-Diferenta multimilor
##Tipuri
Definitii de noi tipuri: `weekDay ::= mon | tue | wed | thu | fri | sat | sun`
##Variabile
Se decalara cu $x:\mathbb{Z}$.
###Relatii
####Relatii pe tipuri de multimi
$S \leftrightarrow T$ este multimea relatiilor intre multimiile S si T
$a \rightarrowtail b$ denota perechea (a, b), daca $(a, b) \in S \leftrightarrow T $
####Operatii pe relatii
-Domeniu $\text{domR} = \{ a : S; b : T | a \rightarrowtail b \in R \bullet a \}$
-Codomeniu $\text{ranR} = \{ a : S; b : T | a \rightarrowtail b \in R \bullet b \}$
-Relatia inversa
-Compunerea
-Inchideri
##Functii
###Notatii
In loc de $\leftrightarrow$ folosim notatiile:
-$\rightarrow$ pentru functii totale
-$\nrightarrow$ pentru functii partiale

#JML
E un limbaj de specificatie formala pentru Java. Comentarii in codul sursa descriu formal cum trebuie sa se comporte un modul, prevenind ambiguitatea. Se foloseste de e invarianti, pre-si postconditii.
```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @
  @ also
  @
  @ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter < 2
  @ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
  @*/
public void enterPIN (int pin) {...}
```
##Notatii
-~!a~ not a
-~a && b~ a si b
-~a | b~ a sau b
-~a ==> b~ a implica b
-~a <==> b~ a e echivalent cu b
-~(\forall T x; a)~ pentru toti x de tip T, a este adevarat
-~(\exists T x; a)~ exista x de tip T, astfel incât a este adevarat
-~(\forall T x; b; a)~ pentru toti x de tip T care satisfac b, a este adevarat
-~(\exists T x; b; a)~ exista x de tip T care satisfac b, astfel incât a este adevarat
###Exemple
Variabila *m* are valoarea elementului maxim din vectorul *vec*:

```
(\forall int i; 0 <= i && i < vec.length; m >= vec[i])
&& (vec.length > 0 ==>
    (\exists int i; 0 <= i && i < vec.length; m == vec[i]))
```

#Cerinte software
Cerintele sunt descrieri ale serviciilor oferite de sistem si a constrângerilor sub care acesta va fi dezvoltat si va opera. Cerintele pornesc de la afirmatii abstracte de nivel inalt pâna la specificatii matematice functionale detaliate (d.ex. Z)
##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, limbaj de programare sau metoda 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 sistem laolalta cu constrângerile operationale.
-scrise pentru clienti
-trebuie sa descrie cerinte functionale si non-functionale intr-o maniera in care sunt pe intelesul utilizatorilor sistemului care nu detin cunostinte tehnice detaliate.
Se adreseaza:
-utilizatorilor finali
-inginerilor clientului
-proiectantilor de sistem
-managerilor clientului
-managerilor 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 sistem
-programatorilor
##Structura documentului de specificare a cerintelor
-Prefata
-Introducere
-Glosar de termeni
-Definirea cerintelor utilizatorilor
-Arhitectura sistemului
-Specificarea cerintelor de sistem
-Modelarea sistemului
-Evolutia sistemului
-Anexe
-Index
##Posibilitati de reprezentare a cerintelor
-limbaj natural: Cerintele sunt organizate in paragrafe numerotate.
-limbaj natural structurat: utilizarea unui format standard sau a machetelor in conjunctie cu limbajul natural
-limbaj de proiectare: asemanator unui limbaj de programare dar mai abstract (nu prea se mai foloseste)
-limbaj grafic suplimentat cu adnotari textuale (mai ales pentru cerinte sistem), cum ar fi UML.
-specificatii matematice: concepte matematice lucrând cu masini cu stari finite sau relatii peste multimi -Z, B. Elimina ambiguitatile, dar pot fi dificil de inteles.

#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 "profiluri" 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
##Tipuri de folosire
-diagrame UML pentru a schita doar diverse aspecte ale sistemului
-diagrame UML care apar in documente (uneori dupa ce a fost facuta implementarea)
-diagrame UML foarte detaliate sunt descrise in tool-uri inainte de implementare si apoi cod este generat pe baza acestor modele
##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); de exemplu ClientCarte. De obicei, actorii principali initiaza cazul de utilizare.
-Actorii secundari sunt cei cu ajutorul carora se realizeaza cazul de utilizare; de exemplu un sistem pentru gestiunea unei biblioteci. 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 inglobeaza un set de cerinte ale sistemului care reies din specificatiile initiale si sunt rafinate pe parcurs.
-Cazurile de utilizare pot avea complexitati diferite; de exemplu "Împrumuta carte" si "Cauta carte".
####Frontiera sistemului
-este important de a defini frontiera sistemului astfel încât 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.
-daca se dezvolta un sistem software, frontiera se stabileste de obicei la frontiera dintre hardware si software.
-trasarea frontierei este optionala; trebuie insa indicata atunci când exista mai multe subsisteme, pentru a le delimita clar.
####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.

-Practic, relatia « include » arata ca secventa de evenimente descrisa in cazul de utilizare inclus se gaseste si in secventa de evenimente a cazului 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 întâmpla 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. "Generalizarea" este asemanatoare cu relatia « extend ».
De obicei, folosim « extend » daca descriem un comportament excepional care depinde de o conditie testata in timpul executiei si "generalizarea" pentru a evidentia o anumita versiune a unui task.
##Diagrame de secvente
Este tipul de diagrama UML care pune in evidenta transmiterea de mesaje (sau apeluri de metode) de-a lungul timpului. Seamana cu MSC.
###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 până 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.În UML, o clasa este prezentata printr-un dreptunghi in interiorul caruia se scrie numele acesteia. Fiecare clasa este caracterizata printr-o multime de atribute si operatii.
###Atribute de vizibilitate
-publice ``+``: pot fi accesate de orice alta clasa
-private ``-``: nu pot fi accesate de alte clase
-protejate ``#``: pot fi accesate doar de subclasele care descind din clasa respectiva
-package ``~``: pot fi accesate doar de clasele din acelasi "package"
###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.
-generalizare. Generalizare: relatie intre un lucru general (numit superclasa sau parinte, ex. Abonat) si un lucru specializat (numit subclasa sau copil, ex. AbonatPremium)
-dependenta
-realizare
##Diagrame de stari
Diagramele de stare (numite si masini de stare sau statecharts) 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.
-tranzitii intre stari, reprezentate prin sageti
-evenimente care declanseaza tranzitiile dintre stari
-cel mai des intâlnite 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.
#Testare
-testarea de validare intentioneaza sa arate ca produsul nu indeplineste cerintele testele incearca sa arate ca o cerinta nu a fost implementata adecvat
-testarea defectelor teste proiectate sa descopere prezenta defectelor in sistem testele incearca sa descopere defecte
-depanarea ("debugging") are ca scop localizarea si repararea erorilor corespunzatoare implica formularea unor ipoteze asupra comportamentului programului, corectarea defectelor si apoi re-testarea programului
##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 (saudepartamente) externe pentru testarea propriilor programe
-rezultatele testelor trebuie analizate amanuntit
-trebuie scrise cazuri de test atât pentru conditii de intrareinvalide si neasteptate, cât si pentru conditii de intrare valide siasteptate
-programul trebuie examinat pentru a vedea daca nu face cetrebuie; de asemenea, trebuie examinat pentru a vedea daca nucumva face ceva ce nu trebuie
-pe cât posibil, cazurile de test trebuie salvate si re-executatedupa efectuarea unor modificari
-probabilitatea ca mai multe erori sa existe intr-o sectiune aprogramului este proportionala cu numarul de erori dejadescoperite in acea sectiune
-efortul de testare nu trebuie subapreciat
-creativitatea necesara procesului de testare nu trebuiesubapreciat
##Tipuri de testari
###Testarea unitara(unit testing)
-o unitate (sau un modul) se refera de obicei la un element atomic (clasa sau functie), dar poate insemna si un element de nivel mai inalt: biblioteca, driver etc.
-testarea unei unitati se face in izolare
###Testarea de integrare(integration testing)
-Testeaza interactiunea mai multor unitati
-Testarea este determinata de arhitectura
###Testarea sistemului(system testing)
-testarea sistemului testeaza aplicatia ca intreg si este determinata de scenariile de analiza
-aplicatia trebuie sa execute cu succes toate scenariile pentru a putea fi pusa la dispozitia clientului
-spre deosebire de testarea interna si a componentelor, care se face prin program, testarea aplicatiei se face de obicei cu script-uri care ruleaza sistemul cu o serie de parametri si colecteaza rezultatele
-testarea aplicatiei trebuie sa fie realizata de o echipa independenta de echipa de implementare
-testele se bazeaza pe specificatiile sistemului
###Testarea de acceptanta(acceptance testing)
Ttestele de acceptanta determina daca sunt indeplinite cerintele unei specificatii sau ale contractului cu clientul.
Sunt de diferite tipuri:
-teste rulate de dezvoltator inainte de a livra produsul software
-teste rulate de utilizator (user acceptance testing)
-teste de operationalitate (operational testing)
-testare alfa si beta: alfa la dezvoltator, beta la client cu un grup ales de utilizatori
###Testarea de regresie(regression testing)
-un test valid genereaza un set de rezultate verificate, numit "standardul de aur"
-testele de regresie sunt utilizate la re-testare, dupa realizarea unor modificari, pentru a asigura faptul ca modificarile nu au introdus noi defecte in codul care functiona bine anterior
-pe masura ce dezvoltarea continua, sunt adaugate alte teste noi, iar testele vechi pot ramâne valide sau nu
-daca un test vechi nu mai este valid, rezultatele sale sunt modificate in standardul de aur
-acest mecanism previne regresia sistemului intr-o stare de eroare anterioara
###Testarea de performanta(performance testing)
O parte din testare se concentreaza pe evaluarea proprietatilor non-functionale ale sistemului, cum ar fi:
-siguranta ("reliability") -mentinerea unui nivel specificat de performanta
-securitatea -persoanele neautorizate sa nu aiba acces, iar celor autorizate sa nu le fie refuzat accesul
-utilizabilitatea -capacitatea de a fi inteles, invatat si utilizat etc. (v. urmatoarele doua slide-uri)
###Testarea la incarcare(load testing)
-asigura faptul ca sistemul poate gestiona un volum asteptat de date, similar cu acela din locatia destinatie (de exemplu la client)
-verifica eficienta sistemului si modul in care scaleaza acesta pentru un mediu real de executie
###Testarea la stres(stress testing)
-solicita sistemul dincolo de incarcarea maxima proiectata
-supraincarcarea testeaza modul in care „cade" sistemul sistemele nu trebuie sa esueze catastrofal testarea la stres verifica pierderile inacceptabile de date sau functionalitati
-deseori apar aici conflicte intre teste. Fiecare test functioneaza corect atunci când este facut separat. Când doua teste sunt rulate in paralel, unul sau ambele teste pot esua.
-o alta varianta, "soak testing", presupune rularea sistemului pentru o perioada lunga de timp (zile, saptamâni, luni); in acest caz, de exemplu scurgerile nesemnificative de memorie se pot acumula si pot provoca caderea sistemului
###Testarea interfetei cu utilizatorul(GUI testing)
Testarea interfetei grafice poate pune anumite probleme cele mai multe interfete, daca nu chiar toate, au bucle de evenimente, care contin cozi de mesaje de la mouse, tastatura, ferestre, touchscreen etc. asociate cu fiecare eveniment sunt coordonatele ecran. Testarea interfetei cu utilizatorul presupune memorarea tuturor acestor informatii si elaborarea unei modalitati prin care mesajele sa fie trimise din nou aplicatiei, la un moment ulterior. De obicei se folosesc scripturi pentru testare.
###Testarea utilizabilitatii(usability testing)
Testeaza cât de usor de folosit este sistemul. Se poate face in laboratoare sau „pe teren" cu utilizatori din lumea reala
Exemple de metode folosite:
-testare "pe hol" (hallway testing): cu câtiva utilizatori aleatori
-testare de la distanta: analizarea logurilor utilizatorilor (daca isi dau acordul pentru aceasta)
-recenzii ale unor experti (externi)
-A/B testing: in special pentru web design, modificarea unui singur element din UI (d.ex. culoarea sau pozitia unui buton) si verificarea comportamentului unui grup de utilizatori
##Metode de testare
##Testarea de tip cutie neagra(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")
###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 o alta metoda de tip cute neagra. Se concentreaza pe examinarea valorilor de frontiera ale claselor, care de regula sunt o sursa importanta de erori.
Avantaje si dezavantaje:
-pasii de inceput (identificarea parametrilor si a conditiilor de mediu precum si a categoriilor) nu sunt bine definiti, bazându-se pe experienta celui care face testarea. Pe de alta parte, odata ce acesti pasi au fost realizati, aplicarea metodei este clara.
-este mai clar definita decât metodele "cutie neagra" anterioare si poate produce date de testare mai cuprinzatoare, care testeaza functionalitati suplimentare; pe de alta parte, datorita exploziei combinatorice, pot rezulta date de test 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. În 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 încât 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 încât 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 independent 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
##Testarea unitara cu JUnit
```
public class Ex1Test {
        @Test public void test_find_min_1() {
                int[] a = {5, 1, 7};

                int res = Ex1.find_min(a);

                assertTrue(res == 1);
        }
        @Test public void test_insert_1() {
                int[] x = {2, 7};
                int n = 6;

                int[] res = Ex1.insert(x, n);
                int[] expected = {2, 6, 7}
        assertTrue(Arrays.equals(expected, res));
        }
}
```
#Depanarea codului
##Debugger
Functionalitatile de baza ale unui debugger (instrumentul care ne ajuta sa identificam problema sau defectul in cod) sunt:
-controlul executiei: poate opri executia la anumite locatii numite breakpoints
-interpretorul: poate executa instructiunile una câte una
-inspectia starii programului: poate observa valoarea variabilelor, obiectelor sau a stivei de executie
-schimbarea starii: poate schimba starea programului in timpul executiei
##Defecte
În multe organizatii se foloseste o clasificare a defectelor pe 4 niveluri:
-defecte critice: afecteaza multi utilizatori, pot întârzia proiectul
-defecte majore: au un impact major, necesita un volum mare de lucru pentru a le repara, dar nu afecteaza substantial graficul de lucru al proiectului
-defecte minore: izolate, care se manifesta rar si au un impact minor asupra proiectului
-defecte cosmetice: mici greseli care nu afecteaza functionarea corecta a produsului software urmarire
#Sabloane de proiectare -design patterns
sabloane de proiectare = solutii generale reutilizabile la probleme care apar frecvent in proiectare (orientata pe obiecte)
##Avantaje
-ca mod de a invata practici bune
-aplicarea consistenta a unor principii de generale de proiectare
-ca vocabular de calitate de nivel inalt (pentru comunicare)
-ca autoritate la care se poate face apel
-in cazul in care o echipa sau organizatie adopta propriile sabloane: un mod de a explicita cum se fac lucrurile acolo
##Dezavantaje
-sunt folositoare doar daca exista intr-adevar problema pe care ele o rezolva
-pot creste complexitatea si scadea performanta
##Sabloane arhitecturale
La nivel de arhitectura.
##Idiomuri
La nivel de limbaj.
##Sabloane de proiectare
Principii:
-programare folosind multe interfete: interfete si clase abstracte pe lânga clasele concrete, framework-uri generice in loc de solutii directe
-se prefera compozitia in loc de mostenire: delegarea catre obiecte "ajutatoare"
-se urmareste decuplarea: obiecte cât mai independente, folosirea "indirectiei", obiecte "ajutatoare"
Continutul unui sablon:
-numele
-problema: obiectivele
-contextul: pre-conditiile
-fortele: constrângerile care indica un compromis, de unde si apare nevoia de sablon
-solutia: cum se ating obiectivele
-contextul rezultat
-justificarea: cum functioneaza intern si de ce
-exemple
-moduri de utilizare cunoscute
-sabloanele inrudite
###Singleton
-implica numai o singura clasa
-ea este responsabila pentru a se instantia
-ea asigura ca se creeaza maxim o instanta
-in acelasi timp, ofera un punct global de acces la acea instanta
-in acest caz, aceeasi instanta poate fi utilizata de oriunde, fiind imposibil de a invoca direct constructorul de fiecare data.
Aplicabilitate:
-când doar un obiect al unei clase e necesar
-când instanta este accesibila global
-este folosit in alte sabloane (factories si builders)
Consecinte:
-accesul controlat la instanta
-spatiu de adresare structurat (comparativ cu o variabila globala)
###Abstract Factory
Ofera o interfata pentru crearea de familii de obiecte inrudite sau dependente fara a specifica clasele lor concrete.Exemplu: un set de instrumente GUI (Widgets) care ofera look-and-feel multiplu, sa zicem pentru pachetele Motif si Presentation Manager (PM).
Consecinte:
-numele de clase de produse nu apar in cod
-familiile de produse usor interschimbabile
-cere consistenta intre produse
###Builder
Separa constructia unui obiect complex de reprezentarea sa, astfel încât acelasi proces de constructie poate crea reprezentari diferite.
Exemplu: citeste RTF (Rich Text Format) si traduce in diferite formate interschimbabile.
Comparatie cu Abstract Factory: Builder creeaza un produs complex pas cu pas. Abstract Factory creeaza familii de produse, de fiecare data produsul fiind complet.
###Facade
Ofera o interfata unificata pentru un set de interfete intr-un subsistem. Exemplu: un subsistem de tip compilator care contine scanner, parser, generator de cod etc. Şablonul Facade combina interfetele si ofera o noua operatie de tip compile().
###Observer
Presupunem o dependenta de 1:n intre obiecte. Cand se schimba starea unui obiect, toate obiectele dependente sunt instiintate. Exemplu: poate mentine consistenta intre perspectiva interna si cea externa
###Visitor
Reprezinta operatii pe o structura de obiect prin obiecte. Adauga noi operatii, fara a modifica insa clasele. Exemplu: procesarea arborelui sintactic intr-un compilator (type checking, generare de cod, pretty print)
##Anti-sabloane
Exemple:
-abstraction inversion
-input kludge
-interface bloat
-magic pushbutton
-race hazard
-stovepipe system
#Licente
-licente comerciale (pe calculator sau utilizator)
-shareware (acces limitat temporal sau functional)
-freeware (gratuit)
-open source(BSD, X11 (MIT), Mozilla Public Licence (MPL), Gnu (GPL, LGPL))