1. **Backlog Management**

* Backlog contine: Epics, User stories, Non-functional requirements, Chores, Defects
* Exprimarea cerintelor: Product Requirements Document(PRD),Use cases, User stories
* INVEST: Independent, Negotiable, Valuable, Estimable, Small, Testable

1. **User story**

* Definitie

Describes functionality that will be valuable to either a user or purchaser of a system or software.

* Caracteristici
* Basic unit of work in an agile project
* Describes a desired piece of business functionality
* Small enough to be implemented in an iteration
* A good user story is the simplest statement about the system that: the customer cares about, test cases can be written to verify, can be reasonably estimated, can be reasonably prioritized
* Epic

A too large user story is called an epic and can be split into stories of smaller size

1. **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

Afirmaţii despre servicii pe care sistemul trebuie să le

conţină, cum trebuie el să răspundă la anumite intrări şi cum să reacţioneze în anumite situaţii. Ex: calculations, technical details, data manipulation and processing.

* Cerinte non-functionale

Constrângeri ale serviciilor și funcţiilor oferite de sistem cum ar fi: constrângeri de timp, constrângeri ale procesului de dezvoltare, standarde, etc.

1. Procese de dezvoltare software

* Procesul de dezvoltare cascada: 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.
* 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 incât 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.

1. **Metodologia agile**

Se concentreaza mai mult pe cod decât pe proiectare si se bazeaza pe o abordare iterativa de dezvoltare de software; produc rapid versiuni care functioneaza, acestea evoluând repede pentru a satisface cerinte in schimbare.

* Principii

4. Clientii si dezvoltatorii trebuie sa colaboreze zilnic pe parcursul proiectului.

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.

9. Atentia continua pentru excelenta tehnica si design bun imbunatateste agilitatea.

1. **Extreme programming (XP)**

* Definitie
* 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.
* Valori: simplitate, comunicare, reactie, curaj, respect.
* Principii
* “dezvoltarea incrementală” este susținută prin intermediul livrării de software în mod frecvent cu mici incremente.
* “implicarea clientului” înseamnă angajamentul “full-time” al clientului cu echipa de dezvoltare.
* “oameni, nu procese” prin programare în doi (pereche de programatori), proprietatea colectivă și un proces care să evite orele lungi de lucru.
* “receptivitate la schimbare” prin livrări frecvente.
* “menținerea simplității” prin refactoring constant de cod.
* Practici
* client disponibil pe tot parcursul proiectului
* implementare treptata limbaj comun
* integrare continua
* rescriere de cod pentru
* 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.
* Avantaje programare 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)

1. **Metoda 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 sfârsitului sprintului, rezultatul ar trebui sa fie livrabil (adica folosit de client sau vandabil). -dupa o analiza a sprintului, se reitereaza.
* There are 3 Elements of Scrum: Artifacts (The Product Backlog, the Sprint Backlog and the Burndown Chart), Timeboxes and Roles (ScrumMaster, Product Owner and Scrum Team).

1. **UML**

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

\*\*\*În UML, o **interfaţă** specifică o colecţie de operaţii și/sau

atribute, pe care trebuie să le furnizeze o clasă sau o

componentă.

* Avantaje
* 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
* 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
* 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 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.

-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 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

-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 ».

* **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.

Participarea unei clase la o asociere este caracterizată de o

anumită **multiplicitate**. De exemplu, o carte poate avea una

sau mai multe copii, în timp ce o copie de carte aparţine doar

unei singure cărți.

**Agregarea şi compunerea** reprezintă tipuri de asociere în care un

obiect dintr-o clasă face parte dintr-un obiect din altă clasă.

Agregarea este modul cel mai general de a indica în UML o relaţie

de tip parte-întreg.

Diferența dintre o simplă asociere și agregare este pur

conceptuală: folosirea agregării indică faptul că o clasă reprezintă

un lucru "mai mare" (întregul), care conține mai multe lucruri "mai

mici" (părțile).

Există însă un caz special de agregare, compunerea, în care

relația dintre întreg şi părţile sale este mai puternică.

Compunerea implică o apartenență puternică a părții la întreg

și o coincidență între durata de viață a părții și a întregului:

dacă întregul este creat, mutat sau distrus, atunci şi părţile

componente sunt create, mutate sau distruse.

* generalizare. Generalizare: relatie intre un lucru general (numit superclasa sau parinte, ex. Abonat) si un lucru specializat (numit subclasa sau copil, ex. AbonatPremium)

În plus, copilul poate chiar schimba comportamentul

părintelui. Acest lucru se întâmplă atunci când o operație a

subclasei care are aceeași semnătură ca și o operație a

superclasei suprascrie acea operație. Acest lucru poartă

numele de polimorfism.

Atunci când o clasă are un singur părinte, spunem că folosește moștenire simplă; în caz contrar, moștenirea se numește multiplă.

* dependenta

O clasă A depinde de o clasă B dacă o modificare în

specificația lui B poate produce modificarea lui A, dar nu

neapărat și invers.

O clasă depinde de clasele care apar ca parametrii în

funcții și e asociată cu clasele atributelor

* realizare
* **Diagrame de stari**

Diagramele de stare 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.

1. **Arhitecturi software**

* Clasificare
* Modele structurale:

- Client / server

- Arhitecturi bazate pe componente

- Arhitecturi structurate pe nivele

* Modele functionale:

- Mesagerie (message bus)

- N - nivele / 3 - nivele (segregarea functionalitatii pe nivele)

- Arhitecturi orientate obiect

- Arhitecturi orientate servicii

1. **MVC**

* Model - Componenta software ce reprezinta resursele conceptuale puse la dispozitie de catre aplicatie, de regula instantiate prin tehnici ORM din medii de stocare persistente.
* View - Construieste interfata prezentata utilizatorilor, pe baza cerintelor transmise de catre controller sau interogand direct modelele, obtinand ca rezultat un cod HTML, XML, JSON, text, …etc.
* Controller - Conponenta logica a aplicatiei, ce intercepteaza cererile clientilor, interogheaza baza de date prin intermediul modelelor, construieste un raspuns utilizand view-urile, pe care apoi il intoarce catre client.
* 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.

1. **IDE (Medii integrate de dezvoltare)**

* Definitie

= un set de programe care ajută programatorul în scrierea programelor,oferind toți pașii necesari creării unui program într-un singur soft, care, de regulă, oferă o interfață cu utilizatorul grafică, prietenoasă.

* Ce este integrat in IDE

-Tools linii de comanda: Compilator, Debugger, Build-tool, Profiler, Version contol tool, etc

-Navigatoare: File manager, project manager, class manager

-Unul sau mai multe editoare de text

-Unul sau mai multe console

1. Refactorizare

* Definitie

Refactorizarea codului sau „code refactoring” este procesul de modificare a unei secvențe de program fără a-i schimba funcționalitatea externă.

* Practici
* 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.

1. **VSC (version control systems)**

* Definitie

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.

* GIT
* Sistem open source, descentralizat, dezvoltat de Linus Torvalds (autorul Linux kernel).
* Particularităti:

-Salveaza patches (diferente) pentru fiecare commit

Local branches folosite intensiv

-Operatiuni de merge, rebase, fork mult mai facile decat in SVN

-Repository principal poate fi schimbat

-Fork & pull request

* Instrumente de build
* Apache Maven - Instrument de build, raportare și documentare a proiectului, utilizând plug-ins.
* Gradle - Instrument de build cu spectru larg, utilizat în general pentru proiecte Java & Scala, C/C++ și Android (default build tool).

1. **Inspectia codului (code reviews)**

* Echipa de inspecție/recenzie a codului are

patru membri:

1. Moderatorul – un programator competent
2. Programatorul – cel care a scris codul inspectat
3. Designer-ul, dacă este o persoană diferită de programator
4. Un specialist în testare

* Programatorul citeşte, instrucţiune cu instrucţiune, logica programului; ceilalţi participanţi pun întrebări

1. **Defecte directe**

Apar în interiorul aplicațiilor, ca efect al:

* Erorilor sau omisiunilor din etapele de specificare și de proiectare a aplicației software (defecte de design);
* Greșelilor sau erorilor ce apar în etapa de implementare a aplicației software (defecte de fabricație);

1. **Defecte indirecte**

Apar ca o consecință a funcționării aplicațiilor într-un anumit context. Astfel avem defecte datorate utilizării unor:

* Biblioteci extene (third party frameworks) instabile;
* Sisteme de operare care conțin la rândul lor defecte sau incompatibilități;
* Compilatoare care produc un cod executabil incorect sau incomplet;
* Dispozitive hardware instabile sau incompatibile;

1. **Mecanisme de limbaj (exemple)**

* Static type system (în majoritatea compilatoarelor);
* Namespaces și încapsularea din OOP care rezolvă o serie de ambiguități și conflicte
* de nume;
* Programarea modulară;
* Managementul automat al memoriei heap (garbage collector);
* Absența aritmeticii pointerilor;
* Implementarea de „bounds checking” pentru vectori (vezi limbajul Pascal și o serie de
* limbaje de scripting), etc.;

1. **Debugger**

Un debugger este o aplicație software folosită pentru a testa și a depana o altă aplicație

(deseori numită „target”). Un debugger poate porni target-ul sau se poate atașa la acesta în timp ce target-ul rulează.

* Atribute
* Permit executarea codului linie cu linie;
* Într-o linie de cod pot permite executarea operațiilor pas cu pas;
* Permit inspectarea și modificarea valorilor variabilelor;
* Permit inspectarea stivei de apeluri;
* Permit inspectarea diferitelor threaduri;
* Permit suspendarea condiționată sau necondiținată a execuției unei aplicații într-un
* anumit punct;
* Unele debuggere permit modificarea atât a stării target-ului cât și a funcționalității
* target-ului fără a-l reporni;
* Unele debuggere implementează operația de „reverse debugging”;

1. **Breakpoint**

Un breakpoint resprezintă un punct în cadrul unui program folosit pentru a opri execuția

acestuia în acel loc.

Atunci când execuția programului este suspendată (pause) programatorul poate inspecta:

* mediul/contextul intern al programului: variabile locale și globale, regiștrii, memoria, stiva de apelurii (call stack);
* mediul/contextul extern al programului: log-uri, fișiere (de configurare sau fisiere generate), ce a afișat până în acel punct, etc.

1. **Algoritm de localizarea sistematică a defectelor**

În I vom păstra un set de locații infectate (variabilă + contor de program)

În L păstrăm locația curentă într-o execuție care a eșuat

1. Fie L locația infectată raportată de eșec și I := {L}

2. Calculăm setul de instrucțiuni S care ar putea conține originea defectului:

un nivel de dependență “înapoi” din L pe calea de execuție

3. Inspectăm locațiile L1, … , Ln scrise în S și dintre ele alegem într-o

mulțime M ⊆ { L1, … , Ln } pe cele infectate

4. În cazul în care M ≠ ∅ (adică cel puțin un Li este infectat):

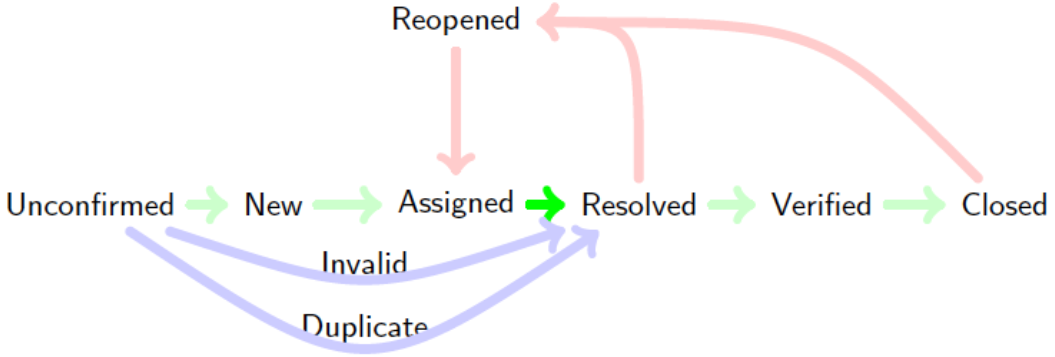
4.1 Fie I : = (I \ {L}) ∪ M (înlocuim L cu noii candidați din M)

4.2 Alegem ca nouă locație L, o locație aleatoare din I

4.3 Ne întoarcem la pasul 2.

5. L depinde doar de locații neinfectate, deci aici este locul de infectare!

1. **Stadiul de viata al defectelor**



1. **Testarea**

* 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
* Principii
* 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

1. **Testarea unitară (unit testing)**

O unitate (sau un modul) se referă de obicei la un element atomic (clasă sau funcţie), dar poate însemna şi un element de nivel mai înalt: bibliotecă, driver etc. Testarea unei unităţi se face în izolare.

* Un test conține
* inițializarea (clasei sau a argumentelor necesare)
* apelul metodei testate
* decizia (oracolul) dacă testul a reușit sau a eșuat

➡ acesta e foarte important pentru evaluarea automată a

testului

➡ compară valorile produse de metodă cu cele corecte

* Pasi

În test-driven development, se scrie mai întâi testul și apoi implementarea. Acum se implementează metoda, dar prima oară ne asigurăm ca testul eșuează (pentru a fi siguri ca nu cumva testul să aibă succces în orice condiții). După ce testul eșuează, implementăm metoda.

* Exemple

**@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));

}

**@Test** (**expected** = IndexOutOfBoundsException.class) public void outOfBounds() {

new ArrayList<Object>().get (1);

}

1. **Testarea de integrare (integration testing)**

Testează interacţiunea mai multor unităţi. Testarea este determinată de arhitectură.

1. **Testarea sistemului (system testing)**

Testarea sistemului testează aplicaţia ca întreg şi este

determinată de scenariile de analiză. Aplicaţia trebuie să execute cu succes toate scenariile pentru a putea fi pusă la dispoziţia clientului. Se face de obicei cu scripturi care rulează sistemul cu o serie de parametri şi colectează rezultatele. Testele se bazează pe specificaţiile sistemului.

1. **Testarea de acceptanță (acceptance testing)**

Testele de acceptanță determină dacă sunt îndeplinite cerințele unei specificații sau ale contractului cu clientul. Sunt de diferite tipuri:

* teste rulate de dezvoltator înainte de a livra produsul software
* teste rulate de utilizator
* teste de operaționalitate
* testare alfa și beta

1. **Testele 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 funcţiona 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.

1. **Testarea performanţei (performance testing)**

O parte din testare se concentrează pe evaluarea proprietăţilor non-funcționale ale sistemului, cum ar fi: siguranța, securitatea, utilizabilitatea, load & stress testing.

1. **Testarea la încărcare (load testing)**

Asigură faptul că sistemul poate gestiona un volum aşteptat de date, similar cu acela din locaţia-destinaţie (de exemplu la client). Verifică eficienţa sistemului şi modul în care scalează acesta pentru un mediu real de execuţie.

1. **Testarea la stres (stress testing)**

Solicită sistemul dincolo de încărcarea maximă proiectată. Supraîncărcarea testează modul în care „cade” sistemul sistemele nu trebuie să eşueze catastrofal testarea la stres verifică pierderile inacceptabile de date sau funcţionalităţi. Deseori apar aici conflicte între teste. Fiecare test funcţionează corect atunci când este făcut separat. Când două teste sunt rulate în paralel, unul sau ambele teste pot eşua

cauza este de obicei managementul incorect al accesului la resurse critice (de exemplu, memoria).

1. **Testarea interfeţei cu utilizatorul (GUI testing)**

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

testarea interfeţei cu utilizatorul presupune memorarea tuturor acestor informaţii şi elaborarea unei modalităţi prin care mesajele să fie trimise din nou aplicaţiei, la un moment ulterior. De obicei se folosesc scripturi pentru testare.

1. **Testarea utilizabilităţii (usability testing)**

Testează cât de uşor 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 (dacă își dau acordul pentru aceasta)
* recenzii ale unor experți (externi)
* A/B testing: în special pentru web design, modificarea unui singur element din UI (d.ex. culoarea sau poziția unui buton) și verificarea comportamentului unui grup de utilizatori

1. **Testarea de tip “cutie neagră” (metoda 1)**

Se iau în considerare numai intrările (într-un

modul, componentă sau sistem) şi ieşirile dorite, conform specificaţiilor; structura internă este ignorată (de unde și numele de “black box testing”).

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.

Cum toate valorile dintr-o clasă au același comportament, presupunem că toate valorile dintr-o clasă sunt procesate în același fel, fiind deci suficient să se aleagă câte o valoare din fiecare clasă.

1. **Testarea de tip “cutie albă” (metoda 2)**

Ia în calcul codul sursă al metodelor testate. Datele de test sunt generate pe baza implementării (programului), fără a lua în considerare specificația

(cerințele) programului. Pentru a utiliza structura programului, acesta poate fi reprezentat sub forma unui graf orientat. Datele de test sunt alese astfel încât să parcurgă toate elementele grafului (instrucțiune, ramură sau cale) măcar o

singură dată.

1. **Performanta**

Capacitatea unui sistem software de a executa sarcinile pentru care a fost proiectat, în timpul și condițiile de lucru prestabilite.

* Programe testare performanta
* Instrumente de stress testing: Apache JMeter, LoadRunner, Visual Studio
* Profilere: NetBeans Profiler, Chrome Inspector, JProfiler, JetBrains dotTrace.

1. **Continuous delivery**

Tehnica software de lucru in cicluri scurte, cu scopul de a avea in orice moment o aplicatie stabila, pregatita pentru deployment in productie. Ex: capistrano, chef, Docker.

* Caracteristici
* Se dezvolta aplicatia, se testeaza si se publica in
* productie in ritm crescut;
* Capacitate de a lucra pe mai multe fire de executie
* (functionalitati noi sau bug fixes), izolate si derivate din trunk;
* Optiunea de a publica in productie numai anumite
* modificari, intr-un timp scurt.

1. **Live monitoring software**

Sisteme software ce analizează condițiile de execuție ale aplicațiilor, atât în medii de dezvoltare sau testare, cât mai ales în producție. Ex: LiveAction, New Relic, SQL Live Monitor.

1. Caracteristici

* Instrumentează aplicațiile țintă, prin programe /plug-ins adiționale cu overhead minim.
* Colectează date de intrare, ieșire sau performață ale aplicațiilor, punându-le la dispoziția utilizatorilor într-o interfață prietenoasă, deregulă configurabilă.

1. **Sabloane**

= soluţii generale reutilizabile la probleme care apar frecvent în proiectare orientată pe obiecte).

1. Creaționale: Abstract Factory, Builder, Factory Method
2. Structurale: Adapter, Bridge, Composite
3. Comportamentale: Iterator, Mediator, Memento
4. **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.

1. **Abstract Factory**

Ofera o interfata pentru crearea de familii de obiecte inrudite sau dependente fara a specifica clasele lor concrete.Ex: 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 interschimbabil, cere consistenta intre produse.

1. **Builder**

Separa constructia unui obiect complex de reprezentarea sa, astfel incât acelasi proces de constructie poate crea reprezentari diferite. Ex: 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.

1. **Facade**

Ofera o interfata unificata pentru un set de interfete intr-un subsistem. Ex: un subsistem de tip compilator care contine scanner, parser, generator de cod etc. Șablonul Facade combina interfetele si ofera o noua operatie de tip compile().

1. **Observer**

Presupunem o dependenta de 1:n intre obiecte. Cand se schimba starea unui obiect, toate obiectele dependente sunt instiintate. Ex: poate mentine consistenta intre perspectiva interna si cea externa

1. **Visitor**

Reprezinta operatii pe o structura de obiect prin obiecte. Adauga noi operatii, fara a modifica insa clasele. Ex: procesarea arborelui sintactic intr-un compilator (type checking, generare de cod, pretty print)

1. **Code metrics**

Ex: Clang, CppCheck, CppDepend, Sonar, SourceMeter.