

**1.Criză software din anii 60 a fost cauzată de hardul insuficient de puternic fata de complexitatea excesiva a softwareului existent la acel moment.**

Afirmatia este evident falsa. Criza software din anii 60 a fost cauzata de cresterea rapida a puterii computerelor( de unde deducem ca hardul nu era insuficient) si a complexitatii problemelor ce trebuiau rezolvate. Masinariile au devenit cu cateva ordine de marime mai puternice.(au devenit mult mai puternice decat erau oamenii capabili sa le programeze).

**2.Cunoasterea perfectă a limbajului de programare în care se va implementa un proiect de mari dimensiuni e o condiție necesară și suficientă pentru terminarea cu succes a proiectului.**

In mod evident, este necesara cunoasterea limbajului de implementare pentru a termina cu succes un proiect. Insa afirmatia este falsa, deoarece indiferent cat de bine stiu ar fi limbajul de programare, acesta nu asigura terminarea proiectului cu succes, incat si alti factori precum: intelegera cerintelor, realizarea designului si testarea corespunzatoare sunt extrem de importanti. Prin cunoasterea limbajului de programare la perfectie se salveaza timp, insa doar 1/6 din timpul necesar realizarii unui proiect ii este alocat programarii.

**3.Intr-un sistem cu o modularitate foarte buna legea lui Brooks nu se aplica deoarece construirea sistemului e o activitate perfect partitionabila.**

Aceasta afirmatie este fundamental falsa. Legea lui Brooks ne spune ca nu e eficient sa adaugam mai multi oameni la un program aflat in intarziere, deoarece acest fapt il va face sa intarzie si mai mult. Acest lucru se datoreaza faptului ca construirea sistemului NU este o activitate perfect partitionabila, incat este absolut necesara comunicarea intre membrii echipei. Evident, in cazul unui sistem cu o modularitate foarte buna, taskurile sunt mai usor partitionabile. Totusi, chiar si in acest caz:

- 1) indiferent de cat de buna este modularitatea, exista un grad mare de comunicare necesar
- 2) indiferent cat de buna e modularitatea, construirea sistemului nu e o activitate perfect partitionabila

**4.Procesul de dezvoltare in cascada e recomandat in majoritatea proiectului, deoarece datorita structurii sale rigide poate constraingea clientul sa descopere inca de la inceput toate posibilele cauze de schimbare din sistem.**

Afirmatia este falsa. Procesul de dezvoltare in cascada nu este recomandat in majoritatea proiectelor ,ci doar in cazul proiectelor care se desfasoara in mai multe locatii. Structura rigida nu poate sa constrainga clientul sa descopere inca de la inceput toate posibilele cauze de schimbare din sistem, fiind indicat doar cand:

- a)cerintele sunt extrem de clare de la bun inceput
- b)schimbarile cerintelor sunt extrem de limitate

Desi se poate anticipa costul si eventualele erori se elimina in faza initiala, dureaza mult pana clientul primeste versiunea finala a proiectului.

**5. Procesele de dezvoltare “agile” se numesc astăzi pentru că atunci când le folosim, dezvoltarea întregului sistem durează garantat mai puțin decât dacă folosim alte procese.**

Afirmatia de mai sus este falsa. Desi agile este un proces cu o abordare iterativa, care produce satisfactia clientilor livrand codul frecvent(saptamanal) este gresit sa afirmam ca numele acestuia este datorita faptului ca procesul dureaza mai putin ca alte procese. Nu se poate stii cu certitudine daca acest proces dureaza garantat mai putin, incat pricepiile agile nu doresc sa obtina acest lucru,

ci mai de graba: interacțiunea f2f cu clientii, primirea cu bratele deschise a schimbarilor tarzii, atenția continua acordata perfectiunii tehnice si a designului, etc.

**6.In procesele de dezvoltare iterative se stabileste pentru fiecare iteratie un set de funcitonalitati care trebuie adaugate iar daca se constata ca timpul prevazut pentru iteratie nu este suficient pentru implementarea intregului set de funcitonalitati, se prelungeste corespunzator durata iteratiei si se creste preventiv si durata urmatoarei iteratii.**

Afirmatia precedenta este falsa. Este adevarat ca in cazul proceselor iterative pentru fiecare iteratie se stabileste un set de functionalitati ce trebuesc adaugate, insa NICIODATA durata iteratiei nu se mareste, chiar daca timpul prevazut a fost insuficient. Daca se constata ca timpul este insuficient se poate prelungi durata catorva functionalitati din iteratie.

**6.Ar trebui modelate in sistem doar clasele care transmit mesaje, nu cele care primesc mesaje**

Afirmatia de mai sus este falsa, incat conform problemei proliferarii claselor, clasele sunt aceleia care primesc mesaje, nu aceleia care primesc mesaje.

**7.Relatia de mai jos se interpreteaza astfel "clasa Car contine clasa Engine", mai precis cand se distruse clasa car se distrug si clasa engine.**

Fals. Intr-adevar in imagine reprezentata relatia de compositie, relatie de tip "parte-intreg", conform careia obiectele parte sunt create si folosite de un singur un singur obiect de tip intreg. Afirmatia este totusi falsa, deoarece relatia de compositie nu se stabileste intre clase, ci intre obiectele claselor. Nu clasele sunt cele distruse, ci obiectele claselor.

**8.Scenariile aferente use caseurilor si diagramele de secventa UML sunt doua modalitati alternative de modelare a interacțiunii intre clase. Cu alte cuvinte, daca dorim sa observam cum colaboreaza clasele, putem folosie fie usecase uri, fie diagrame de secventa.**

Afirmatia precedenta este falsa. Diagramale use case suprind interacțiunile dintre actori si sistem, insa pe baza lor nu se poate observa cum colaboreaza clasele. Diagramale UML, in schimb descriu clasele de obiecte: inclusiv atributile si operatiile lor, precum si relatiile dintre ele, fiind potrivite daca dorim sa observam cum colaboreaza acestea.

**9.Precizati si argumentati succint daca relatia descrisa(telefon dual sim) cartelele pot fi oricand reutilizate si in alt telefon.**

Relatia descrisa de diagrama UML NU modeleaza in totalitatea ceea ce sustine enuntul. In mod cert, realtia din figura, este o relatie de compositie, si indica faptul ca un obiect telefon are in compositie sa un numar de cartele ce poate varia intre niciuna si doua. Partea din enunt care ma face sa remarc ca este fals e aceea ca "cartele pot fi oricand reutilizate si in alt telefon", complet fals in cazul relatiei de compositie, conform careia obiectele parte sunt create si folosite exclusiv de catre un singur obiect intreg. Astfel, obiectele "cartela" sunt construite pentru un singur obiect "telefon", putand fi folosite doar de catre acele obiecte.

**10. Un actor nu poate introduce informatii in sistemul modelat ci poate doar extrage informatii din acesta.**

Afirmatia este falsa. Actorul specifica rolul jucat de user in cadrul interacțiunii cu sistemul. Aceasta poate atat oferi cat si primii informatii din acestea, cat timp este exterior sistemului.

**11. Use caseul “Retrage bani de la ATM” e intr o relatie de tip “include” cu use case ul “Retrage bani de la ATM ramas fara cash” pentru ca primul e mai cuprinzator.**

Afirmatia este in mod evident falsa. Relatia de tip “<<include>>” este intradevar folosita pentru a factoriza functionalitatile comune ale un use case, dar este evident din enunt ca este vorba de o relatie de tip “<<extends>>” prin intermediul careia marcam o situatie exceptionala, un caz rar intalnit cum este cazul in care bancomatul ramane fara bani.

**12. In tehnica FAST de analiza a cerintelor, fiecare participant trebuie sa construiasca cate 4 liste independent de ceilalți și fara presiunea ca listele sa fie complete.**

Afirmatia de mai sus este adevarata. In cadrul tehnicii FAST dupa ce s-au facut 1-2 pagini cu cerinte, fiecare participant trebuie sa vina cu 4 liste(care contin: obiect, servicii, constrangeri, criterii de performanta), fara presiunea ca listele sa fie complete, incat se doreste doar observarea a cat mai multe puncte de vedere, motiv pentru care este important ca listele sa fie facute in mod independent. In cadrul tehnicii FAST nimic nu se “arunca”, ci din toate listele se face in final una singura eliminand evident redundantele.

**13. Testarea ocupa 25% din costul total al dezvoltarii unui sistem soft**

Afirmatia este falsa. Conform regulilor lui Brooks de impartire a programului, testarii ii este asociat jumata din costul total al dezvoltarii(1/4 se duce pe testarea de componente si ¼ pe testarea sistemului).

**14. Pentru a evita proliferarea claselor, clasa Family tr implementata ca in C4 sl4, var dreapta**

Afirmaria este falsa. Comportamentul este cel care decide: daca este diferit avem clase, iar daca nu doar roluri ale aceleiasi clase. Evident, depinde de DOMENIUL MODELAT DE APlicatie(de ceea ce cere sistemul). De exemplu, daca o functionalitate ceruta de sistem ar fi “schimbaScutece()” atunci am opta pentru a doua varianta, deoarece aceasta operatie poate fi realizata de orice membru al familiei. In schimb, daca functionalitatea ceruta ar fi ”naste()” este evident ca doar mama ar putea face asta, varianta potrivita fiind cea din stanga.

**15. Folosirea constantei globale incalca criteriul de modularitate al continuitatii.**

Conform criteriului de modularitate al continuitatii schimbarile mici au ca efect producerea de schimbare in doar cateva module(nu afecteaza intreaga arhitectura). Astfel, este evident ca constantele globale incalca acest criteriu, incat modificarea lor intr-un loc determina modificarea lor in toate functiile in care apar. Acest fapt e in antiteza cu criteriul de modularitate al continuitatii, deci este FALS.

**16. 50% din timpul si costurile totale pentru un proiect ar trebui alocat testarii, dar in multe cazuri in realitate, procentajul este mai mic.**

Intr-adevar, conform lui Brooks modul in care ar trebui construit programul este: 1/6 codare, 1/3 planificare, ½ pentru testare, deci testarii I-ar trebui alocare jumata din timpul si costurile totale pentru un proiect. Afirmatia este insa falsa, incat, in realitate, procentajul nu este mai mic, deoarece oamenii sunt constienti de efectele dezastroase produse de testarea insuficienta.

**17. Forma reala a curbei ratei defectarii sw este datorata faptului ca cerintele nu sunt intelese clar de la bun inceput, ceea ce duce la necesitatea de schimbari continue asupra sistemului**

Forma reala a curbei defectarii sw este datorata schimbarilor permanente ce se produc in procesul de dezvoltare software( fie ca ne referim la noi cerinte, ori la procesul de mentenanta). Afirmatia este totusi falsa, deoarece indiferent cat de bine sunt intelese cerintele la inceput, schimbarile sunt inevitabile.

**18. Conceptul de time boxing spune ca sarcinile alocate pentru o anumita iteratie trebuie terminate in cadrul iterataiei in cauza.**

Conceptul de time boxing spune ca fiecarei iteratii ii este atribuita o perioada fixa de timp( numita time box), timp in care pot fi indeplinite anumite sarcini. In schimb, nu spune nicaieri ca o anumita iteratie trebuie terminata strict in cadrul iteratiei respective. In cazul in care timpul este insuficient, timpul alocat iteratiei NU se maresteste, insa se pot asigna anumite sarcini urmatoarei iteratii. Astfel, afirmatia este FALSA.

**19. In procesele de dezvoltare agile nu este necesar sa intelegem de la bun inceput cerintele sistemului.**

Afirmatia este cu certitudine falsa. Intr-adevar, procesele de dezvoltare agile se remarca prin flexibilitate, incat chiar si schimbarile tarzii sunt bine-venite, insa este gresit sa credem ca asta inseamna ca cerintele nu trebuie clar intelese de la bun inceput. In orice proces de dezvoltare este esential ca cerintele sa fie intelese, insa comparativ cu alte procese(cum ar fi Waterfall), in cadrul carora cerintele trebuie sa aiba un numar limitat de schimbari si sa fie perfect clare de la bun inceput, metodologia agile se bazeaza mai mult pe comunicare f2f cu oamenii si este deschisa spre schimbari.

**20. Daca avem la dispozitie diagrame UML foarte explice pentru un sistem, putem deduce diagramele de secventa pentru acel sistem, adica toate interactiunile posibile dintre obiecte.**

Diagramele UML descriu clasele din punct de vedere structural, impreuna cu atributele si operatiile lor, prezantand de asemenea relatiile ce exista intre acestea, fara a prezenta vreun amanunt legat de implementarea acestora. In schimb, diagramele de secventa descriu sistemul din punct de vedere al interactiunilor, fiind extrem de folositoare pentru a vedea daca un obiect lipseste( desi consuma mult timp, merita investit) iar construitea lor necesita accesul la cod. Evident, afirmatia este falsa deoarece indiferent cat de explice ar fi diagramele UML pentru un sistem, acestea nu ajuta la construirea diagramelor de secventa(si reciproc).

**21. Un sistem nu poate fi actor pentru un alt sistem pentru ca acest lucru ar implica accesul la informatiile din sistemul mare, ceea ce incalca incapsularea.**

Actorul este cel care specifica rolul jucat de un user in interactiunea cu sistemul. Este important de stiut ca actorul interactioneaza cu sistemul din exterior, fara ca acesta sa aiba acces la informatiile din sistem. Astfel, in mod evident, afirmatia este falsa incat un sistem poate fi un actor pentru un alt sistem, iar acest fapt nu incalca incapsularea.(De ex daca un om (actor) vrea sa retraga o suma de bani de la bancomat(sistem), acesta nu trebuie sa stie cum a fost implementat bancomatul).

**22. Relatia din figura(compozitie, rombul colorat) modeleaza cazul unui telefon dual-sim, cand cartela poate fi oricat reutilizata si in alt telefon.**

Relatia din figura alaturata este evident o relatia de tip "parte-intreg", iar rombul colorat ne indica faptul ca este o relatia de compozitie. Intr-adevar, relatia modeleaza cazul unui telefon dual-sim, incat din figura deducem ca putem avea de la 0 la 2 cartele. Partea din enunt care indica ca este FALSE ca o cartela poate fi reutilizata si in alt telefon. Daca relatia ar fi fost de agregare, atunci enuntul ar fi fost adevarat, dar cum in cazul relatiei de compozitie ,obiectului intreg ii este asociat un singur obiect de tip parte(iar odata cu distrugerea intregului este distrus si obiectul/obiectele parte), este fals.

COMPOZITIE → obiectele parte sunt create si folosite exclusiv de catre un singur obiect intreg  
AGREGARE → obiectele parte pot fi reutilizate de diferite obiecte intreg

**23. Intr-o diagrama de secventa obiectele care sunt instante ale aceleiasi clase trebuie sa comasate intr-un singur dreptunghi, adica nu este admis sau recomandat sa reprezentam fiecare obiect ca un dreptunghi separat pentru ca astfel s-ar incarca prea mult diagrama**

Intr-o diagrama de secventa, obiectele care sunt instante ale aceleiasi clase NU trebuie sa comasate intr-un singur dreptunghi, incat fiecare obiect are propriul sau rol. Daca ele ar fi comasate intr-un singur dreptunghi nu am mai veda interactiunile asa cum ar fi firesc, deci afirmatia este fundamental falsa.

**24. Scenariile aferente use caseurilor si diagramele de secventa UML sunt doua modalitati de modelare a interactiunilor intre clase.**

Scenariile use case sunt folosite pentru a arata interactiunea dintre actori si sistem, pe cand diagramele UML de secventa prezinta comportamentul intre 2 sau > entitati in termeni de interactiune si ordinea in care sunt mesajele schimbate. Astfel, enuntul este fals, incat use caseurile nu pot fi folosite pentru a arata interactiunea claselor.

**25. In modelarea cerintelor se poate aplica urmatoarea regula: aproape orice substantiv intalnit intr-un document de cerinte e un actor si aproape orice verb e un USE CASE.**

Afirmatia este cu certitudine falsa. Actorul este acela care arata rolul jucat de un user in interactiunea cu sistemul, insa nu orice substantiv intalnit in documentul de cerinte indeplineste un rol. De asemenea, use caseurile, actiunile ce definesc interactiunea dintre sistem si actori nu sunt reprezentate de aproximativ orice verbe. Doar actiunile care au un anumit scop trebuie extrase, nu si pasii intermediari. De regula use caseul este descris printr-un substantiv si un verb.

**26. Un actor nu poate introduce informatii intr-un sistem modelat ci poate fi doar beneficiarul responsabil de sistem(nu poate extrage informatii din sistem)**

Actorul, cel care specifica rolul jucat de un user in interactiunea cu sistemul, poate atat sa trimita cat si sa primeasca informatii de la sistem, cat timp interactioneaza cu acesta din exterior. Astfel, afirmatia este falsa.

**27. In procesul de dezvoltare Scrum Burndown Chart ne arata evolutia efortului de-a lungul intregii istorii a proiectului, mai exact ne ofera informatii despre sprinturile deja incheiate precum si numarul de taskuri finalizate pana in prezent in intregul proiect.**

Afirmatia este falsa. In procesul de dezvoltare Scrum, Burndown Chart-ul este o reprezentare grafica a muncii care mai trebuie finalizata si timpul ramas pentru terminarea ei, fiind utilizat pentru a anticipa cand se va finaliza proiectul, in niciun caz pentru masurarea evolutiei efortului de-a lungul intregului proiect.

**28. O problema importanta a proceselor de dezvoltare iterative e aceea ca adesea trebuie sa se modifice codul care a fost scris intr-o iteratie anterioara si uneori anumite portiuni de cod trebuie sa se sterse, ceea ce reprezinta o pierdere/risipa.**

Afirmatia este adevarata deoarece intr-adevar in cadrul proceselor de dezvoltare iterative, modificarea codului scris intr-o iteratie anterioara ori chiar sergerea anumitor portiuni de cod reprezinta o risipa de timp, deci implicit de cost. In schimb, pentru a reduce aceasta pierdere, codul ar trebui rescris, incat se pierde mult mai mult timp in incercarea de a "repara" un cod prost decat se pierde pentru rescriere.

**29. In procesul de dezvoltare Scrum, daca functionalitatile prevazute pentru un Sprint nu pot fi terminate intr-un timp prevazut, Sprintul poate fi prelungit, dar nu mai mult de cateva zile.**

In cadrul procesului de dezvoltare Scrum, se aloca o perioada fixa de timp (numita Sprint) in care trebuie finalizate anumite functionalitati. Chiar daca aceste nu sunt finalizate la timp, Sprintul NU se prelungeste, incat acest fapt poate duce la cresterea complexitatii, a riscului si evident, a costului.

Astfel, afirmatia este falsa, iar sprintul nu poate fi prelungit nici macar cateva zile, insa pot fi eliminate din functionalitati.

**30. Legile evolutiei software ale lui Lehman nu se aplica la sisteme ale caror cerinte sunt perfect intelese de la bun inceput, pentru ca astfel de sisteme nu au niciun motiv sa evolueze.**

Afirmatia este in totalitate falsa, incat nu exista niciun sistem care sa nu trebuiasca sa evolueze (exceptand poate sistemele care nu sunt folosite niciodata). De asemenea, faptul ca cerintele sunt perfect intelese de la bun inceput nu indica faptul ca sistemul nu ar trebui sa evolueze, ci poate ajuta la finalizarea mai alerta a sistemului Software. Legile lui Lehman se aplică tuturor sistemelor, incat toate sistemele trebuie sa evolueze, caci in caz contrar devin din ce in ce mai nesatisfacatoare pentru clienti.

**31. Daca avem diagrame de clasa UML explicite pentru un sistem, atunci putem deduce diagramele de secventa pentru el, adica toate interactiunile posibile din sistem.**

Diagramele UML descriu clasele din punct de vedere structural, incluzand operatiile si atributele lor, reprezentand de asemenea si relatiile dintre ele. Diagramele de secventa, pe de alta parte, descriu comportamentul din punct de vedere al interactiunilor si ordinea in care mesajele sunt schimbate. Astfel, este evident gresit sa credem ca diagramele UML, indiferent de cat de explicite ar fi ele, ajuta la deducerea diagramelor de secventa(cu atat mai putin cu cat pentru a realiza o diagrama de secventa trebuie sa avem acces la cod, iar diagramele UML nu ofera acest lucru).

**32. Stilul arhitectural Pipes and Filters este avantajos din punct de vedere al timpului.**

Afirmatia este in mod evident falsa. Filtrele, independente unele fata de celalalte, se occupa cu prelucrarea de date, iar conductele sunt cele care transporta datele de la un filtru la altul. Este insa cunoscut faptul ca pipes and Filters nu este un stil arhitectural avantajos din punct de vedere al timpului, incat au nevoie de un format comun de date, iar fiecare filtru trebuie sa isi faca "parse" si "unparse" datelor, ceea ce duce la OVERHEAD, deci, implicit, la pierderea timpului.

**33. Diagramele UML de secventa sunt folosite ca punct de plecare pentru sesiunile de CRC cards, in urma carora se construieaza diagramele uml de clase.**

Afirmatia este in mod cert falsa. CRC cards sunt folosite pentru identificarea claselor ce indica responsabilitatile si colaboratorii lor, pe cand diagramele UML de secventa descriu comportamentul din punct de vedere al interactiunilor si surprind ordinea in care se activeaza mesajele. In mod evident, diagramele de secventa nu pot fi folosite ca punct de plecare pentru realizarea CRC cards, punctul de plecare al acestora fiind de fapt use case-urile. Este insa adevarat ca pe baza crc cards se realizeaza diagramele de clasa.

**34. Un sistem care respecta criteriul de modularitate al compozibilitatii il va respecta aproape sigur si pe cel al decompozibilitatii incat primul se refera la posibilitatea de a crea un sistem din module.**

Criteriul de modularitate al decompozibilitatii presupune descompunerea problemelor in probleme mai mici care pot fi rezolvate separat, pe cand criteriul de modularitate al compozibilitatii presupune compunerea libera a modulelor pentru a produce sisteme. In mod cert, afirmatia este falsa, incat cele doua criterii sunt independente si complet opuse. De exemplu Ikea,ne exemplifica un caz in care este respectat criteriul decompozibilitatii, dar nu si cel al compozibilitatii/

**35. Un avantaj principal al stilului arhitectural Stratificat este asigurarea criteriului de modularitate al protectiei.**

Afirmatia este adevarata. Criteriul de modularitate al protectiei spune ca efectele unei conditii de executie anormale sunt limitate la cateva module. Stilul arhitectural Stratificat indeplineste cu siguranta acest criteriu, incat, in cazul in care se modifica interfata unui Layer sau se adauga facilitati noi, doar layerele adiacente vor fi afectate, nu tot sistemul.

**36. Procesul de dezvoltare Extreme Programming spune ca majoritatea timpului ar trebui investit in programare si mai putin in alte activitati conexe precum pactarea cerintelor/testare, incat produsul software este creat prin programare.**

Extreme programming este un proces de dezvoltare ce abordeaza extrem dezvoltarea iterativa: o noua versiune in fiecare zi/noapte, incrementii sunt livrati la fiecare 2-3 saptamani, iar toate teste trebuie esc rulate pentru fiecare constructie. Afirmatia este insa falsa, incat EP nu spune ca majoritatea timpului ar trebui investit in programare, incat testarea, designul si alte activitati sunt la fel de importante.

**40. Use Case-urile de tip Fish Level se folosesc atunci cand vrem sa detaliem fiecare actiune principala din cadrul unui use case de tip sea Level.**

Use Case-urile de tip Sea Level sunt acele care descriu interacțiunea dintre user și sistem, interacțiunea fiind majoră, iar scopul bine precizat. Use case-urile de tip Fish Level le folosim când vrem să factorizăm funcționalități comune folosind <<include>>. În mod cert, afirmația este falsă.

**41. Stim ca am descoperit toate Use-Case-urile dintr-un sistem atunci cand toate UC sunt acoperite de use case-uri Sea Level, fiecare dintre acestea trebuind să fie conectate cu cel puțin un actor.**

Use case-urile de tip Sea Level sunt folosite pentru a descrie interacțiunea dintre sistem și actori, interacțiunile fiind majore și cu un scop bine precizat. Astfel, putem într-adevăr spune că am descoperit toate use-case-urile din sistem când toate UC sunt acoperite de UC Sea level. De asemenea, este adevarat că fiecare dintre acestea trebuie conectat cu cel puțin un actor, caci altfel nu ar avea sens, un use case presupun existența a minim un user care să îl folosească.

**42. În tehnica CRC clasele sunt identificate pornind de la substantivale din descrierea UC, iar responsabilitatea de identificare dintre verbele folosite în cadrul descrierii.**

Adevărat. În tehnica CRC clasele sunt identificate pornind de la substantivale, incat POO CERE CA SI CLASELE SA POATA DEFINI ENTITATI ALE SISTEMULUI, deci ele nu pot fi decat substantivale, iar responsabilitatile claselor reprezinta functionalitatea lor REDATA PRIN METODE, , iar aceasta functionalitate este redată de verbele din scenariul unei diagrame use case/

**43. În stilul architectural Repository diferențele componente care procesează date sunt total independente unele de altele, cu excepția faptului că folosesc același model de date.**

Stilul repository este indicat de folosit pentru distribuirea unor cantități mari de date, consumatorii și producatorii fiind independenți între ei. Într-adevăr un dezavantaj este faptul că folosesc același model de date. Astfel, afirmația este adevarată.

**44. Un program care nu are cicluri are numarul ciclomatic = 1**

Complexitatea ciclometrică este influențată de numărul de decizii simple din program. Astfel, este absurd să credem că dacă un program nu are cicluri, acesta are complexitate = 1, incat și instrucțiunile de tip if/else contin astfel de decizii. Astfel, numărul ciclomatic depinde de cum este structurat programul.

**45. Principalul dezavantaj al testarii top down integration este crearea de stuburi și driveri.**

In cazul integrării de tip top down se integrează modulele de sus în jos, testând întâi modulul principal. Într-adevăr un dezavantaj al testării top-down este necesitatea de a crea stub-uri, care înlocuiesc modulele care sunt subordonate componentei testate, însă în cazul acestei testări nu se creează driveri, acestea din urmă fiind caracteristice testării bottom-up, în cadrul căreia integrarea și testarea începe de jos în sus, testând întâi cele mai mici module. Driverele se folosesc în cadrul testării bottom up drept un program principal în care se apelează funcția testată.

**46. Ideea de baza in testarea whitebox e aceea de a ne asigura ca cel putin privita individual functia nu are niciun bug in nicio instructiune, incat toate instructiunile sunt verificate cu cate cel putin un test.**

Afirmatia este adevarata in totalita. In cazul testarii de tip white box, dupa cum spunea Pressman scopul e sa ne asiguram ca fiecare conditie a fost executata macar o data, deci implicit, ca functia nu are niciun bun in nicio instructiune cel putin privita individual. In cazul acestui tip de testare se face un minim de teste astfel: se parcurge bucla de 0,1,2 ori, de max-1 si de max ori.

**47. Partitiile echivalente se folosesc la sistemele cu domeniul datelor de intrare foarte mare si reprezinta clase de input-uri asemanatoare din punct de vedere al testarii. Astfel, pentru fiecare partitie se aleg una sau mai multe functii pentru care se vor scrie suite black box.**

Intr-adevar partitiile echivalente se folosesc in cazul sistemelor cu un domeniu al datelor de intrare foarte mare, in cadrul careia clasele au inputuri asemanatoare din punct de vedere al testarii. Ceea ce face enuntul sa fie fals, este a doua propozitie, incat in cazul partitiilor echivalente testelete se fac pentru valorile corespunzatoare marginilor si mijlocului.

**48. Daca dorim sa verificam cat mai timpuriu principalele puncte de control si de decizie din sistem vom alege testarea de integrare de timp Bottom up.**

Testarea de integrare de tip Bottom up presupune integrarea si testarea modulelor de jos in sus, intai fiind testate cele mai mici module, astfel ca prin aceasta metoda nu se verifica timpuriu principalele puncte de control si de decizie din sistem, acest avantaj fiind caracteristic testarii Top Down, in cadrul careia sunt integrate modulele de sus in jos, deci prima data se testeaza modulul principal.

**49. Valoarea complexitatii ciclomatrice a unei functii nu e influentata de numarul de instructiuni de ciclare incat singurele instructiuni care incrementeaza caloare complexitatii sunt cele de decizie de tip if-else.**

Afirmatia este cu siguranta falsa. Complexitatea ciclomatica ne spune cat de complex este codul, iar aceasta poate fi calculata ca fiind numarul de decizii simple + 1. Este drept ca instructiunile de decizie de tip if-else contin astfel de decizii simple, insa acest fapt nu inseamna ca instructiunile de ciclare nu contin astfel de decizii, incat complexitatea ciclomatica a unui program este influentata si de de buclele for, while etc.

**50. Principalul motiv pentru care nu este recomandata testarea de tip Big Bang e acela ca aceasta depisteaza mai putine erori.**

Afirmatia este falsa. In cazul testarii de tip Big Bang tot programul este testat o singura data, motiv pentru care se creeaza un haos si este extrem de dificila depistarea erorilor, iar corectarea lor la fel de dificila incat cand aceste sunt corectate apar altele, iar testarea pare ca intra intr-o bucla infinita.

**51. Testarea ciclurilor pentru o functie cu doua cicluri imbriicate implica scrierea a doua cazuri de test: unul care sa parcurga instructiunile din ciclul interior si un al doilea care sa testeze instructiunile din ciclul exterior.**

Afirmatia este adevarata. In cazul functiei cu doua cicluri imbriicate intai se testeaza bucla din interior( care se face ca testarea unei bucle simple) pastrand iteratioul buclei exterioare la valoare minima. Apoi, se aplica acelasi procedeu si in cazul buclei exterioare, iar acelasi procedeu s-ar fi aplicat indiferent de cate cicluri imbriicate ar fi fost.

**52. Precizati ce stil arhitectural s-ar asocia cel mai bine cu fiecare dintre imagini:**

**a. O ceapa**→In mod evident, aceasta imagine ne duce cu gandul la stilul arhitectural Layered, tocmai datorita aspectului stratificat, fiecare strat comunicand doar cu straturile adiacente.

**b. O linie de productie**→Aceasta imagine ne duce cu gandul la stilul arhitectural pipes and filters. Filtrele din imagine le asociem cu diferite masinarii care prelucreaza datele, respectiv transforma

materiile prime in produs finit, iar conductele le asociem cu scarile rulante care transporta produsele prin fabrica.

c.IDE→ Ne duce cu gandul la Repository, in cadrul caruia se gasesc mai multe unelte, independente intre ele care folosesc un model comun de date

**Numarul ciclomatic ne arata numarul de “cai” de executie posibile pentru un anumit program.**

1. Forma reala a curbei ratei defectarii software-ului este datorata faptului ca cerintele nu sunt intelese clar de la bun inceput, ceea ce duce la necesitatea de schimbari continue asupra sistemului.

Fals. Curba reala difera de cea ideală din cauza apariției schimbărilor repetitive în soft. Schimbările apar atunci când se efectuează lucrări de menținere asupra softului sau modificări asupra softului.

2. 50% din timp și costuri ar trebui alocate testării, dar în multe cazuri procentajul alocat testării este mai mic.

Fals. Testarea reprezintă într-adevar jumătate din costuri și din timp, dar aceste estimări nu sunt fictive, deoarece chiar atât reprezintă, trebuie să facem multă testare pe lângă partea de cod pentru a ne asigura că produsul software este bun și face ce trebuie.

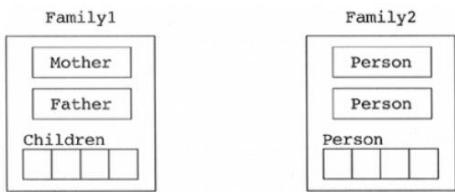
3. Conceptul de time boxing spune că sarcinile alocate pentru o anumită iterată trebuie terminate în cadrul iterării în cauză.

Adevarat. Conceptul de time boxing aloca o perioadă fixă fiecărei iterării, numita time box. În cadrul acestei perioade de timp sarcinile alocate în cadrul iterării trebuie terminate. În cazul în care nu se termină task-urile alocate deadline-ul nu se modifică, mai degrabă se amână niste task-uri pentru time box-ul viitor.

4. Dacă lucram cu procese de dezvoltare agile, asta înseamnă că nu trebuie să intelegem clar cerintele sistemului.

Fals, cerintele sistemului trebuie intelese indiferent de procesele de dezvoltare, dar în cazul proceselor de dezvoltare agile acestea se pot modifica de la o iterată la alta sau se mai pot adăuga alte cerinte de la o iterată la alta, important este că acestea să nu se schimbe în timpul unei iterării.

5. Pentru a evita problema proliferării claselor, ar trebui să implementăm clasa Family ca mai jos.



Adevarat. Pentru a evita proliferarea claselor trebuie să fim atenți la comportamentul acestora. El este cel care decide: dacă comportamentul difera avem clase diferite, dacă nu difera avem roluri diferite ale acelasi clase.

6. Dacă avem diagrame de clasa UML explicite pentru un sistem, putem deduce diagramele de secvență pentru acel sistem, adică toate interacțiunile posibile dintre obiecte.

Fals. Diagramalele de clasa ne arată structura sistemului, atributele și metodele fiecărei clase, pe când diagramalele de secvență modelează scenariile dintr-un sistem. Acestea nu se pot deduce unele din altele.

**7. Un sistem nu poate fi actor pentru un alt sistem pentru ca acest lucru ar implica accesul la informatiile din sistemul mare, ceea ce incalca incapsularea.**

Fals, un sistem poate fi actor pentru un alt sistem, intrucat acesta interactioneaza cu interfata acestuia si nu are acces la codul acestuia, astfel nu se incalca principiul incapsularii.

**8. Stilul architectural *Pipes and Filters* este avantajos din punct de vedere al timpului.**

Fals. Acesta este format din filtre care sunt independente unele fata de celelalte si care prelucreaza datele si din conducte care sunt cai prin care datele circula de la filtru la filtru. Filtrele nu au un format comun de date, iar din aceasta cauza fiecare filtru trebuie sa faca "parse" si "un-parse" datelor, lucru care duce la overhead si, implicit, la pierderea timpului

**9. Un program care nu are cicluri (for, while) are numarul ciclomatic=1.**

Fals. Numarul ciclomatic este influentat de numarul deciziilor simple din cadrul unui program. Instructiunile de ciclare de tip for/while contin si ele o astfel de instructiune, asadar numarul ciclomatic in cazul in care nu avem instructiuni de ciclare in program este influentat doar de numarul deciziilor simple din cadrul acestuia.

**10. Principalul dezavantaj al testarii top down integration este crearea de drivere si stub-uri .**

Fals. In cazul testarii de tip top-down se creeaza doar stub-uri, care imita comportamentul metodelor apelate din functia testata. Driver-ele sunt create la testarea de tip bottom-up si reprezinta un program principal simplu din care se apeleaza functia testata.

**11. Folosirea constantelor globale incalca criteriul de modularitate al continuitatii.**

Fals. Folosirea constantelor globale nu incalca principiul continuitatii.

**12.Curba reală a evoluției în timp a ratei de defecte (Failure Rate) diferă față de cea ideală în principal din cauza lipsei de experiență a programatorilor.**

Fals, deoarece curba reala difera de cea ideală din cauza aparitiei schimbarilor repeatate in soft. Schimbarile apar atunci cand se efectueaza lucrari de mentenanta asupra softului. Lipsa de experienta a programatorilor nu este un motiv principal de aparitie a erorilor in soft.

**13. Legile evoluției software-ului ale lui Lehman nu se aplică la sisteme ale căror cerințe sunt perfect înțelese de la bun început, pentru că altfel de sisteme nu au nici un motiv să evolueze.**

Fals, deoarece legile evolutiei software se aplica oricarui sistem indiferent daca cerintele sunt intelese sau nu de la bun inceput. Sistemele software oricum evolueaza datorita schimbarilor inconjuratoare.

**14.** În procesul de dezvoltare Scrum, Burndown Char ne arată evoluția efortului de-a lungul întregii istorii a proiectului, mai exact oferă informații despre Sprint-urile deja încheiate, precum și numărul de Task-uri finalizează până în prezent în întregul proiect.

Fals. Burndown Chart ne arata evolutia efortului si numarul de task-uri finalizeate si de finalizat de-a lungul unui Sprint, perioada in care se realizeaza un Increment dintr-un proiect. Burndown Chart-ul nu arata evolutia efortului pentru intregul proiect.

**15.** O problemă importantă a proceselor de dezvoltare iterative este aceea că adesea trebuie modificat codul care a fost scris într-o iteracție anterioară, și uneori anumite porțiuni de cod trebuie chiar șterse, ceea ce reprezintă o pierdere/risipă.

Corect. Aceasta este o problema importantă a proceselor de dezvoltare iterative și este o risipa, în schimb, putem să refacem codul decât să pierdem timpul modificand același cod. Singurul cod care se pastrează este codul funcțional care face ceva util în cadrul softului.

**16.** În diagramele de UML de Use-Case relațiile <<extends>> și <<include>> sunt foarte asemănătoare și pot fi folosite interschimbabil însăcum ambele sunt folosite pentru a da “factor comun” descrierea unei anumite funcționalități.

Fals. Relațiile <<extends>> și <<include>> sunt ceea ce poate să difere. Prima reprezintă un caz exceptionál al unui use case, pe când cea de-a două factorizează un comportament comun al mai multor use case-uri.

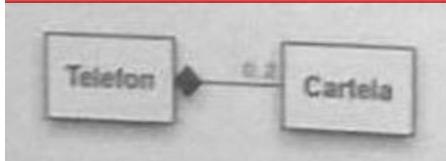
**17.** Diagramele UML de secvență (sequence diagrams) sunt folosite ca punct de plecare pentru sesiunile de CRC Cards în urma cărora se construiesc diagramele UML de clase (class diagrams).

Fals. Punctul de plecare pentru a crea un CRC card sunt diagramele use case care redau un scenariu identificând actorii și interacțiunile lor cu sistemul. În urma creării cardurilor CRC se realizează diagramele de clasa și apoi diagramele de secvență.

18. Într-o diagramă de secvență UML, toate obiectele care sunt instanțe ale aceleiași clase trebuie "comasate" într-un singur dreptunghi; adică nu este admis sau recomandat să reprezentăm fiecare obiect ca un dreptunghi separat, pentru că altfel s-ar încărca prea mult diagrama.

Fals. Toate instantele unei clase trebuie desenate în dreptunghiuri separate, însă fiecare obiect poate avea un comportament diferit chiar dacă sunt instante ale aceleiași clase.

19. Relația din figura de mai jos modelează cazul unui telefon dual-sim (care poate să îndeplinească simultan două cartele) cu cartele care pot fi oricând reutilizate și în alt telefon.



Adevărat. Avem clasa Telefon și clasa Cartela cu o relație de compunere între ele. Într-adevar, un obiect Cartela nu poate exista fără un obiect Telefon, dar putem avea instante ale clasei Cartela și în alte obiecte Telefon.

20. Un sistem care respectă criteriul de modularitate al Decompozabilității îl va respecta aproape sigur și pe cel al Compozabilității întrucât acesta din urmă se referă la posibilitatea compunerii unui sistem din module.

Fals. Dacă un sistem respectă criteriul de modularizare al decompozabilității nu înseamnă că îl va respecta și pe cel al compozabilității. Exemplu cu IKEA și LEGO.

21. Un avantaj principal al stilului arhitectural Stratificat (Layered Architecture) este asigurarea criteriului de modularitate al Protecției.

Adevărat. În cazul stilului arhitectural stratificat când se modifică interfața unui layer sau se adaugă noi facilități este afectat doar layer-ul adjacente, nu întregul sistem.

22. Ideea de bază în testarea whitebox este aceea de a ne asigura că cel puțin privită individual o funcție nu are nici un bug și nici o instrucțiune, întrucât toate instrucțiunile sunt verificate cu câte cel puțin un test.

Adevărat. Testarea de tip whitebox verifică toate instrucțiunile dintr-o funcție cu cel puțin un test. Astfel, funcția cel puțin privată individual nu are niciun bug.

23. Partițiile echivalente se folosesc la sisteme cu foarte multe funcții lungi și complexe, și reprezintă grupuri de funcții care sunt asemănătoare din punctul de vedere al testării. Astfel, din fiecare grup (partiție) se vor alege una sau mai multe funcții pentru care se vor scrie suite de teste blackbox.

Fals. Partițiile echivalente se folosesc la sistemele cu domeniul datelor de intrare foarte mare și reprezintă clase de input-uri asemănătoare din punct de vedere al testării. Astfel, pentru fiecare partiție se aleg cazuri de teste pentru valorile corespunzătoare marginilor și mijlocului.

24. Legile evoluției software-ului enumărate de către Lehman ne spun că nu există nici o modalitate de a încetini declinul calității software-ului.

Fals. Legile scaderii calitatii a lui Lehman spun ca putem incetini procesul de deteriorare al software-ului daca il adaptam in mod continuu la schimbarile inconjuratoare.

25. Printr-o proiectare riguroasă, implementarea software-ului poate fi transformată într-o activitate aproape perfect partaționabilă.

Fals, o proiectare riguroasa ajuta intr-adevar la partitionarea implementarii softului, dar nu garanteaza o partitionare perfecta. Asta depinde de natura produsul, daca prin natura lui raportat la tehnologiile existente, el poate fi sau nu partitionat cat mai bine.

26. Procesul de dezvoltare Extreme Programming spune că majoritatea efortului trebuie investit în programare și mai puțin în alte activități conexe cum ar fi captarea cerințelor sau testare, întrucât până la urmă produsul software propriu-zis este creat prin programare.

Fals, Extreme Programming nu se refera la programarea in sensul de a scrie cod si atat. Este o metodologie bazata pe Agile care abordeaza extrem dezvoltarea iterativa. Este scrisa o noua versiune foarte des, incrementii sunt livrati clientului la fiecare doua trei saptamani, iar constructia e aprobată doar daca testele sunt trecute cu succes.

27. Deși în procesele de dezvoltare iterative și incrementate trebuie uneori schimbate sau chiar rescrise complet fragmente semnificative de cod ce au fost scrise în iterațiile anterioare, acestea nu reprezintă o pierdere întrucât schimbările sunt inevitabile.

Fals. Aceasta este o problema importantă a proceselor de dezvoltare iterative și este o risipa, in schimb, putem sa refacem codul decat sa pierdem timpul modificand acelasi cod. Singurul cod care se pastreaza este codul functional care face ceva util in cadrul softului.

28. Tehnica de analiza cerințelor folosind Use-Case-uri este specifică procesului de dezvoltare Waterfall pentru că aici cerințele trebuie analizate riguros la începutul proiectului.

Fals. Intr-adevar, procesul de tip Waterfall se foloseste atunci cand cerintele sunt foarte bine cunoscute de la inceput, dar asta nu inseamna ca folosirea use case-urilor este specifica doar acestui tip de proces.

29. Doi sau mai mulți actori nu pot fi asociați (adică nu pot interacționa) cu același use-case pentru că aceasta ar însemna că sunt redundanți.

Fals. Doi actori pot avea acelasi use case printre altele atata timp cat interactionarea lor cu sistemul este diferita in ansamblu. (Nu interactioneaza cu sistemul pentru aceeasi functionalitate a sistemului).

**30.** Într-o diagramă de secvență se recomandă ca dacă apar mai multe instanțe ale aceleiași clase acestea să fie reprezentate într-un singur element pentru a nu se încărca excesiv diagrama.

Fals. Toate instantele unei clase trebuie desenate în dreptunghiuri separate, intrucât fiecare obiect poate avea un comportament diferit chiar dacă sunt instante ale aceleiasi clase.

**31.** Pentru a evita problema proliferării claselor prin modelare ca și clase a entităților externe sistemului trebuie să ținem cont că sunt clase doar acele entități care apelează alte entități (clase) din sistem.

Fals. Clasele sunt acele care sunt apelate (primesc mesaje), nu cele care apelează (transmit mesaje).

**32.** Un corp de mobilă modular care vine împachetat pe bucăți și care trebuie să îl asamblăm/compunem (gen IKEA) a fost proiectat urmărind în special criteriul de modularitate al compozabilității.

Fals. Un corp de mobila modular de la Ikea a fost proiectat în astă fel încât să se poată realiza din acele piese numai un singur corp și numai într-un anumit fel. Acest lucru este în contradicție cu principiul modular al compozabilității care spune că putem compune un sistem modular în mod liber și cum ne dorim.

**33.** Importanța criteriilor și regulilor de modularitate este cu atât mai mare cu cât anvergura sistemului software proiectat este mai mare, cu alte cuvinte: importanța modularizării este proporțională cu dimensiunea sistemului software.

Adevarat. Într-un soft mic aproape că nu avem nevoie de modularizare pentru că nu avem în ce bucati să spargem. Pe când într-un soft mare e important să spargem pe bucati pentru structura frumoasă și pentru a realizare a o comunicare între componente ușor de urmarit și componente ușor de reutilizat.

**34.** Dacă dorim să verificăm cât mai timpuriu principalele puncte de control și decizie din sistem vom alege testarea de integrare de tip Bottom-Up.

Fals. Testarea de integrare Bottom-Up verifică timpuriu procesarea de date de nivel scazut. Pentru a verifica că mai timpuriu principalele puncte de control și decizie din sistem vom alege testarea de integrare de tip Top-Down.

**35.** Valoarea complexității ciclomatische a unei funcții nu este influențată de numărul de instrucțiuni de ciclare (for/while) întrucât singurele instrucțiuni care incrementează valoarea complexității ciclomatische sunt cele de decizie de tip if-else.

Fals. Valoarea complexității ciclomatische este influențată atât de deciziile de tip if/else, cât și de numărul de instrucțiuni de ciclare (for/while), deoarece și aceste instrucțiuni contin câte o decizie simplă.

36. Curba defectelor poate fi făcută să tindă în timp spre zero dacă cerințele sunt bine înțelese de la început, și dacă se aplică sistematic metode de testare eficiente.

Fals. Chiar dacă sunt bine înțelese cerințele de la început și se aplică metode de testare eficiente, în timp, din cauza schimbărilor repetitive tot vor apărea erori în sistem, deci curba defectelor nu va tinde în timp spre 0.

37. Utilizarea eficientă a instrumentelor CASE dedicate programării poate reduce semnificativ costurile totale ale proiectului având în vedere că într-un proiect software o parte semnificativă a costurilor sunt legate de activitatea de codare.

Fals. Utilizarea eficientă CASE reduce costurile, dar ele se folosesc nu doar pentru partea de codare, ci și pentru alte etape ale procesului precum specificații, design, testare, debugging. De-asemenea, o mare parte din costuri nu se duc pe partea de codare, ci pe partea de testare și planificare.

38. În procesul de dezvoltare Waterfall se face doar un singur tip de activitate la un moment dat, spre deosebire de cele iterative unde într-o iterată trebuie efectuate toate tipurile de activitate.

Adevarat. Procesul de dezvoltare Waterfall împarte proiectul în mai multe activități ce sunt rezolvate pe rand. De exemplu, se realizează analiza și definirea cerințelor, după ce se încheie aceasta fază se realizează design-ul de sistem și software și astăzi departe.

39. În procesul Rational Unified Process (RUP), deși este permisă efectuarea mai multor tipuri de activități în interiorul unei faze, se recomandă ca numărul de activități desfășurate în paralel să fie limitat.

Fals. Într-adevar, în cazul procesului Rational Unified Process putem efectua mai multe tipuri de activități simultan în interiorul unei faze, dar nu există nicio limitare cu privire la numărul de procese desfășurate în paralel. De verificat

40. Use-Case-urile de tip Fish Level se folosesc atunci când vrem să detaliem fiecare acțiune principală din cadrul unui use-case de tip Sea Level.

Fals. Use Case-urile de tip Sea Level ne arată cum userul interacționează cu sistemul, interacțiunile sunt majore, iar scopul este bine precizat. Use Case-urile de tip Fish Level sunt acele use case-uri care se dau factor comun din use case-urile Sea-Level (cu <<include>>).

41. Știm că am descoperit toate Use-Case-urile dintr-un sistem atunci când toate cerințele funcționale sunt acoperite de use-case-uri Sea Level, fiecare dintre acestea trebuind să fie conectate cu cel puțin un actor.

Adevarat. Use case-urile de tip Sea Level prin definiție reprezintă interacțiuni majore ale user-ilor cu sistemul cu un scop precis. Astfel dacă toate cerințele funcționale sunt acoperite de Use Case-uri de tip Sea Level am descoperit toate Use Case-urile din

sistem. Si da, trebuie ca fiecare use case sea level sa fie legat la un user pentru ca altfel el nu are sens, un use case presupune existenta cel putin a unui user care sa-l foloseasca.

42. În tehnica CRC clasele sunt identificate pornind de la substantivele din descrierea use-case-urilor, iar responsabilitatea se identifică dintre verbele folosite în cadrul descrierii scenariilor.

Adevarat. In tehnica CRC clasele sunt identificate pornind de la substantive, intrucat programarea orientata pe obiecte cere ca si clasele sa poata defini entitati ale sistemului, deci ele nu pot fi decat substantive, iar responsabilitatile claselor reprezinta functionalitatea lor redată prin metode, iar aceasta functionalitate este data de verbele din scenariul unei diagrame use case.

43. Relația de compoziție este un caz special de agregare (relație parte-întreg) în care indicăm ca distrugerea clasei "întreg", atrage după sine și distrugerea obiectelor "parte" conținute de către clasa "întreg".

Adevarat. Relatia de compositie este o forma de agregare in care componentele nu pot exista una fara cealalta.

44. Între specificatorul de acces "protected" și criteriul de modularitate al protecției există o legătură strânsă.

Fals. Specificatorul de acces „protected” face ca atributele si metodele unei clase sa fie vazute doar de clasele care mostenesc clasa respectiva. Criteriul de modularitate al protectiei ne spune ca daca apare o eroare de executie, aceasta se limiteaza la doar cateva module. Criteriul se poate asigura si cu specificatorul private, cu pachete etc.

45. În stilul arhitectural Repository diferentele componente care procesează date sunt total independente unele de altele, cu excepția faptului că folosesc același model de date.

Adevarat. Stilul arhitectural Repository este un mod eficient de a distribui mari cantitati de date in care producatorii si consumatorii de date sunt independenti, dar marele compromis este faptul ca folosesc acelasi model de date.

46. Principalul motiv pentru care testarea de integrare de tip Big Bang nu este recomandată este acela ca spre deosebire alte tehnici de testare de integrare aceasta identifică mai puține defecte.

Fals. Testarea de integrare de tip Big Bang combina toate componente in avans si testeaza intreg programul. In acest fel se creeaza un haos cu multe erori care la prima vedere nu se leaga. Corectarea acestora se realizeaza greu, dupa aceasta rezulta alte erori si astfel testarea pare sa intre intr-o bucla infinita.

47. Testarea ciclurilor pentru o funcție cu două cicluri imbicate(nested) implică scrierea a două cazuri de test; unul să parcurgă instrucțiunile din ciclul interior, și un al doilea care să testeze instrucțiunile din ciclul exterior.

Adevarat. Testarea functiilor cu 2 cicluri imbicate incepe cu testarea bulei din interior care se face ca testarea unei bucle simple pastrand iteratorul bulei exterioare la valoarea minima. Dupa aceasta se trece la testarea celei de a doua bucle in maniera testarii unei bucle simple.

**49. Precizati ce stil arhitectural s-ar asocia cel mai bine urmatoarele imagini:**

- a. O ceapa - Layered Architecture intrucat aceasta are o structura stratificata, iar in mijloc se afla lastarul care este cea mai importanta parte a acesteia.
- b. O linie de productie - conducte si filtre. Conductele reprezinta benzile rulante care ajuta produsele sa se deplaseze prin fabrica, iarfiltrele sunt reprezentate de diferite masinarii care transforma simplele materii prime in produsul finit, pas cu pas.
- c. Un mediu de dezvoltare integrat (IDE), gen Eclipse, Visual Studio, JBuilder sau un instrument CASE complex - Repository deoarece un mediu de dezvoltare are mai multe unele: compilator, debugger etc, care sunt independente intre ele si care creeaza si folosesc acelasi tip de date.

1. **Forma reala a curbei ratei defectarii software-ului este datorata faptului ca cerintele nu sunt intelese clar de la bun inceput, ceea ce duce la necesitatea de schimbari continue asupra sistemului.**

Fals. Curba reala difera de cea ideală din cauza apariției schimbărilor repetitive în soft. Schimbările apar atunci când se efectuează lucrări de menținere asupra softului sau modificări asupra softului.

2. **50% din timp și costuri ar trebui alocate testării, dar în multe cazuri procentajul alocat testării este mai mic.**

Fals. Testarea reprezintă într-adevar jumătate din costuri și din timp, dar aceste estimări nu sunt fictive, deoarece chiar atât reprezintă, trebuie să facem multă testare pe lângă partea de cod pentru a ne asigura că produsul software este bun și face ce trebuie.

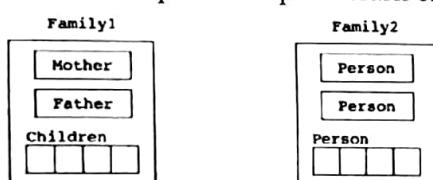
3. **Conceptul de time boxing spune că sarcinile alocate pentru o anumita iteratie trebuie terminate în cadrul iteratiei în cauză.**

Adevarat. Conceptul de time boxing aloca o perioadă fixă fiecărei iterării, numită time box. În cadrul acestei perioade de timp sarcinile alocate în cadrul iterării trebuie terminate. În cazul în care nu se termină task-urile alocate deadline-ul nu se modifică, mai degrabă se amână niste task-uri pentru time box-ul viitor.

4. **Dacă lucram cu procese de dezvoltare agile, asta înseamnă că nu trebuie să intelegem clar cerințele sistemului.**

Fals, cerințele sistemului trebuie intelese indiferent de procesele de dezvoltare, dar în cazul proceselor de dezvoltare agile acestea se pot modifica de la o iteratie la alta sau se mai pot adăuga alte cerinte de la o iteratie la alta, important este că acestea să nu se schimbe în timpul unei iterării.

5. **Pentru a evita problema proliferării claselor, ar trebui să implementăm clasa Family ca mai jos.**



Adevarat. Pentru a evita proliferarea claselor trebuie să fim atenți la comportamentul acestora. El este cel care decide: dacă comportamentul difera avem clase diferite, dacă nu difera avem roluri diferite ale acelasi clase.

6. **Dacă avem diagrame de clasa UML explicite pentru un sistem, putem deduce diagramele de secvență pentru acel sistem, adică toate interacțiunile posibile dintre obiecte.**

Fals. Diagramalele de clasa ne arată structura sistemului, atributele și metodele fiecărei clase, pe când diagramalele de secvență modelează scenariile dintr-un sistem. Acestea nu se pot deduce unele din altele.

**7. Un sistem nu poate fi actor pentru un alt sistem pentru ca acest lucru ar implica accesul la informatiile din sistemul mare, ceea ce incalca incapsularea.**

Fals, un sistem poate fi actor pentru un alt sistem, intrucat acesta interacționează cu interfața acestuia și nu are acces la codul acestuia, astfel nu se incalca principiul incapsularii.

**8. Stilul architectural *Pipes and Filters* este avantajos din punct de vedere al timpului.**

Fals. Aceasta este format din filtre care sunt independente unele fata de celelalte și care prelucră datele și din conducte care sunt cai prin care datele circula de la filtru la filtru. Filtrele nu au un format comun de date, iar din aceasta cauza fiecare filtru trebuie să facă "parse" și "un-parse" datelor, lucru care duce la overhead și, implicit, la pierderea timpului.

**9. Un program care nu are cicluri (for, while) are numarul ciclomatic=1.**

Fals. Numarul ciclomatic este influențat de numărul deciziilor simple din cadrul unui program. Instrucțiunile de ciclare de tip for/while contin și ele o astfel de instrucțiune, asadar numărul ciclomatic în cazul în care nu avem instrucțiuni de ciclare în program este influențat doar de numărul deciziilor simple din cadrul acestuia.

**10. Principalul dezavantaj al testării top down integration este crearea de drivere și stub-uri .**

Fals. În cazul testării de tip top-down se creează doar stub-uri, care imită comportamentul metodelor apelate din funcția testată. Driver-ele sunt create la testarea de tip bottom-up și reprezintă un program principal simplu din care se apelează funcția testată.

**11. Folosirea constantei globale incalca criteriul de modularitate și continuitate.**

~~Fals. Folosirea constantei globale nu incalca criteriul de modularitate și continuitate.~~

**12.Curba reală a evoluției în timp a ratei de defecte (Failure Rate) diferă față de cea ideală în principal din cauza lipsei de experiență a programatorilor.**

Fals, deoarece curba reală diferește de cea ideală din cauza apariției schimbărilor repetitive în soft. Schimbările apar atunci când se efectuează lucrări de menenanță asupra softului. Lipsa de experiență a programatorilor nu este un motiv principal de apariție a erorilor în soft.

**13. Legile evoluției software-ului ale lui Lehman nu se aplică la sisteme ale căror cerințe sunt perfect înțelese de la bun început, pentru că altfel de sisteme nu au nici un motiv să evolueze.**

Fals, deoarece legile evoluției software se aplică oricărui sistem indiferent dacă cerințele sunt înțelese sau nu de la bun început. Sistemele software oricum evoluează datorită schimbărilor inconjurătoare.

14. În procesul de dezvoltare Scrum, Burndown Char ne arată evoluția efortului de-a lungul întregii istorii a proiectului, mai exact oferă informații despre Sprint-urile deja încheiate, precum și numărul de Task-uri finalizate până în prezent în întregul proiect.

Fals. Burndown Chart ne arată evoluția efortului și numărul de task-uri finalizate și de finalizat de-a lungul unui Sprint, perioada în care se realizează un Increment dintr-un proiect. Burndown Chart-ul nu arată evoluția efortului pentru întregul proiect.

15. O problemă importantă a proceselor de dezvoltare iterative este aceea că adesea trebuie modificat codul care a fost scris într-o iteracție anterioară, și uneori anumite porțiuni de cod trebuie chiar șterse, ceea ce reprezintă o pierdere/risipă.

~~Fals - este una hinc decât să avem soft prost~~  
Corect. Aceasta este o problemă importantă a proceselor de dezvoltare iterative și este o risipă, în schimb, putem să refacem codul decât să pierdem timpul modificând același cod. Singurul cod care se pastrează este codul funcțional care face ceva util în cadrul softului.

16. În diagramele de UML de Use-Case relațiile <<extends>> și <<include>> sunt foarte asemănătoare și pot fi folosite interschimbabil însăcum ambele sunt folosite pentru a da "factor comun" descrierea unei anumite funcționalități.

Fals. Relațiile <<extends>> și <<include>> sunt ca și cum se poate de diferențe. Prima reprezintă un caz exceptionál al unui use case, pe când cea de-a două factorizează un comportament comun al mai multor use case-uri.

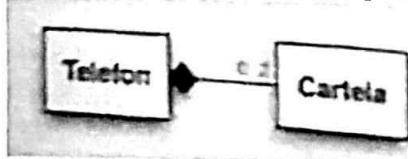
17. Diagramele UML de secvență (sequence diagrams) sunt folosite ca punct de plecare pentru sesiunile de CRC Cards în urma cărora se construiesc diagramele UML de clase (class diagrams).

Fals. Punctul de plecare pentru a crea un CRC card sunt diagramele use case care redau un scenariu identificând actorii și interacțiunile lor cu sistemul. În urma creării cardurilor CRC se realizează diagramele de clasa și apoi diagramele de secvență.

18. Într-o diagramă de secvență UML, toate obiectele care sunt instanțe ale aceleiași clase trebuie "comasate" într-un singur dreptunghi; adică nu este admis sau recomandat să reprezentăm fiecare obiect ca un dreptunghi separat, pentru că altfel s-ar încărca prea mult diagrama.

Fals. Toate instantele unei clase trebuie desenate în dreptunghiuri separate, întrucât fiecare obiect poate avea un comportament diferit chiar dacă sunt instante ale aceleiasi clase.

19. Relația din figura de mai jos modelează cazul unui telefon dual-sim (care poate ține simultan două cartele) cu cartele care pot fi oricând reutilizate și în alt telefon.



*RELATIE => refolosire  
COMPOUNERE => nu refolosire*

~~Adevarat~~. Avem clasa Telefon și clasa Cartela cu o relație de compunere între ele. Într-adevar, un obiect Cartela nu poate exista fără un obiect Telefon, dar putem avea instante ale clasei Cartela și în alte obiecte Telefon.

20. Un sistem care respectă criteriul de modularitate al Decompozabilității îl va respecta aproape sigur și pe cel al Compozabilității întrucât acesta din urmă se referă la posibilitatea compunerii unui sistem din module.

Fals. Dacă un sistem respectă criteriul de modularizare al decompozabilității nu înseamnă că îl va respecta și pe cel al compozabilității. Exemplu cu IKEA și LEGO.

21. Un avantaj principal al stilului arhitectural Stratificat (Layered Architecture) este asigurarea criteriului de modularitate al Protecției.

Adevarat. În cazul stilului arhitectural stratificat când se modifică interfața unui layer sau se adaugă noi facilități este afectat doar layer-ul adjacente, nu întregul sistem.

22. Ideea de bază în testarea whitebox este aceea de a ne asigura că cel puțin privită individual o funcție nu are nici un bug și nici o instrucțiune, întrucât toate instrucțiunile sunt verificate cu câte cel puțin un test.

Adevarat. Testarea de tip whitebox verifică toate instrucțiunile dintr-o funcție cu cel puțin un test. Astfel, funcția cel puțin privită individual nu are niciun bug.

23. Partițiile echivalente se folosesc la sisteme cu foarte multe funcții lungi și complexe, și reprezintă grupuri de funcții care sunt asemănătoare din punct de vedere al testării. Astfel, din fiecare grup (partiție) se vor alege una sau mai multe funcții pentru care se vor scrie suite de teste blackbox.

Fals. Partițiile echivalente se folosesc la sistemele cu domeniul datelor de intrare foarte mare și reprezintă clase de input-uri asemănătoare din punct de vedere al testării. Astfel, pentru fiecare partitie se aleg cazuri de teste pentru valorile corespunzătoare marginilor și mijlocului.

**24. Legile evoluției software-ului enunțate de către Lehman ne spun că nu există nici o modalitate de a încetini declinul calității software-ului.**

Fals. Legile scaderii calitatii a lui Lehman spun ca putem incetini procesul de deteriorare al software-ului daca il adaptam in mod continuu la schimbarile inconjuratoare.

**25. Printr-o proiectare riguroasă, implementarea software-ului poate fi transformată într-o activitate aproape perfect partaționabilă.**

Fals, o proiectare riguroasa ajuta intr-adevar la partitionarea implementarii softului, dar nu garanteaza o partitionare perfecta. Asta depinde de natura produsul, daca prin natura lui raportat la tehnologiile existente, el poate fi sau nu partitionat cat mai bine.

**26. Procesul de dezvoltare Extreme Programming spune că majoritatea efortului trebuie investit în programare și mai puțin în alte activități conexe cum ar fi captarea cerințelor sau testare, întrucât până la urmă produsul software propriu-zis este creat prin programare.**

Fals, Extreme Programming nu se refera la programarea in sensul de a scrie cod si atat. Este o metodologie bazata pe Agile care abordeaza extrem dezvoltarea iterativa. Este scrisa o noua versiune foarte des, incrementii sunt livrati clientului la fiecare doua trei saptamani, iar constructia e aprobată doar daca testele sunt trecute cu succes.

**27. Deși în procesele de dezvoltare iterative și incrementate trebuie uneori să schimbă sau chiar să rescrisce complet fragmente semnificative de cod ce au fost scrise în iterările anterioare, acestea nu reprezintă o pierdere întrucât schimbările sunt inevitabile.**

~~True~~  
~~Fals~~. Aceasta este o problema importanta a proceselor de dezvoltare iterative si este o risipa, in schimb, putem sa refacem codul decat sa pierdem timpul modificand acelasi cod. Singurul cod care se pastreaza este codul functional care face ceva util in cadrul softului.

**28. Tehnica de analiza cerințelor folosind Use-Case-uri este specifică procesului de dezvoltare Waterfall pentru că aici cerințele trebuie analizate riguros la începutul proiectului.**

Fals. Intr-adevar, procesul de tip Waterfall se foloseste atunci cand cerintele sunt foarte bine cunoscute de la inceput, dar asta nu inseamna ca folosirea use case-urilor este specifica doar acestui tip de proces.

**29. Doi sau mai mulți actori nu pot fi asociați (adică nu pot interacționa) cu același use-case pentru că aceasta ar însemna că sunt redundanți.**

Fals. Doi actori pot avea acelasi use case printre altele atata timp cat interactionarea lor cu sistemul este diferita in ansamblu. (Nu interactioneaza cu sistemul pentru aceeasi functionalitate a sistemului).

**30. Într-o diagramă de secvență se recomandă ca dacă apar mai multe instanțe ale aceleiași clase acestea să fie reprezentate într-un singur element pentru a nu se încărca excesiv diagrama.**

Fals. Toate instantele unei clase trebuie desenate în dreptunghiuri separate, intrucât fiecare obiect poate avea un comportament diferit chiar dacă sunt instante ale aceleiasi clase.

**31. Pentru a evita problema proliferării claselor prin modelare ca și clase a entităților externe sistemului trebuie să ținem cont că sunt clase doar acele entități care apelează alte entități (clase) din sistem.**

Fals. Clasele sunt aceleia care sunt apelate (primesc mesaje), nu cele care apelează (transmit mesaje).

**32. Un corp de mobilă modular care vine împachetat pe bucăți și care trebuie să îl asamblăm/compunem (gen IKEA) a fost proiectat urmărind în special criteriul de modularitate al compozabilității.**

Fals. Un corp de mobila modular de la Ikea a fost proiectat în astă fel încât să se poată realiza din acele piese numai un singur corp și numai într-un anumit fel. Acest lucru este în contradicție cu principiul modular al compozibilității care spune că putem compune un sistem modular în mod liber și cum ne dorim.

**33. Importanța criteriilor și regulilor de modularitate este cu atât mai mare cu cât anvergura sistemului software proiectat este mai mare, cu alte cuvinte: importanța modularizării este proporțională cu dimensiunea sistemului software.**

Adevarat. Într-un soft mic aproape că nu avem nevoie de modularizare pentru că nu avem în ce bucati să spargem. Pe când într-un soft mare e important să spargem pe bucati pentru structura frumoasă și pentru a realizare o comunicare între componente ușor de urmarit și componente ușor de reutilizat.

**34. Dacă dorim să verificăm cât mai timpuriu principalele puncte de control și decizie din sistem vom alege testarea de integrare de tip Bottom-Up.**

Fals. Testarea de integrare Bottom-Up verifică timpuriu procesarea de date de nivel scăzut. Pentru a verifica că mai timpuriu principalele puncte de control și decizie din sistem vom alege testarea de integrare de tip Top-Down.

**35. Valoarea complexității ciclomatrice a unei funcții nu este influențată de numărul de instrucțiuni de ciclare (for/while) întrucât singurele instrucțiuni care incrementează valoarea complexității ciclomatrice sunt cele de decizie de tip if-else.**

Fals. Valoarea complexității ciclomatrice este influențată atât de deciziile de tip if/else, cât și de numărul de instrucțiuni de ciclare (for/while), deoarece și aceste instrucțiuni contin câte o decizie simplă.

**36. Curba defectelor poate fi făcută să tindă în timp spre zero dacă cerințele sunt bine înțelese de la început, și dacă se aplică sistematic metode de testare eficiente.**

Fals. Chiar daca sunt bine intelese cerintele de la inceput si se aplica metode de testare eficiente, in timp, din cauza schimbarilor repeatate tot vor aparea erori in sistem, deci curba defectelor nu va tinde in timp spre 0.

**37. Utilizarea eficientă a instrumentelor CASE dedicate programării poate reduce semnificativ costurile totale ale proiectului având în vedere că într-un proiect software o parte semnificativă a costurilor sunt legate de activitatea de codare.**

Fals. Utilizarea eficienta CASE reduce costurile, dar ele se folosesc nu doar pentru partea de codare, ci si pentru alte etape ale procesului precum specificatii, design, testare, debugging. De-asemenea, o mare parte din costuri nu se duc pe partea de codare, ci pe partea de testare si planificare.

**38. În procesul de dezvoltare Waterfall se face doar un singur tip de activitate la un moment dat, spre deosebire de cele iterative unde într-o iteratie trebuie efectuate toate tipurile de activități.**

Adevarat. Procesul de dezvoltare Waterfall imparte proiectul in mai multe activitati ce sunt rezolvate pe rand. De exemplu, se realizeaza analiza si definirea cerintelor, dupa ce se incheie aceasta faza se realizeaza design-ul de sistem si software si asa mai departe.

**39. În procesul Rational Unified Process (RUP), deși este permisă efectuarea mai multor tipuri de activități în interiorul unei faze, se recomandă ca numărul de activități desfășurate în paralel să fie limitat.**

Fals. Intr-adevar, in cazul procesului Rational Unified Process putem efectua mai multe tipuri de activitati simultan in interiorul unei faze, dar nu exista nicio limitare cu privire la numarul de procese desfasurate in paralel. De verificat

**40. Use-Case-urile de tip Fish Level se folosesc atunci când vrem să detaliem fiecare acțiune principală din cadrul unui use-case de tip Sea Level.**

Fals. Use Case-urile de tip Sea Level ne arata cum userul interactioneaza cu sistemul, interactiunile sunt majore, iar scopul este bine precizat. Use Case-urile de tip Fish Level sunt acele use case-uri care se dau factor comun din use case-urile Sea-Level (cu <<include>>).

**41. Știm că am descoperit toate Use-Case-urile dintr-un sistem atunci când toate cerințele funcționale sunt acoperite de use-case-uri Sea Level, fiecare dintre acestea trebuind să fie conectate cu cel puțin un actor.**

Adevarat. Use case-urile de tip Sea Level prin definitie reprezinta interacciuni majore ale user-ilor cu sistemul cu un scop precis. Astfel daca toate cerintele funktionale sunt acoperite de Use Case-uri de tip Sea Level am descoperit toate Use Case-urile din

sistem. Si da, trebuie ca fiecare use case sea level sa fie legat la un user pentru ca altfel el nu are sens, un use case presupune existenta cel putin a unui user care sa-l foloseasca.

42. În tehnica CRC clasele sunt identificate pornind de la substantivele din descrierea use-case-urilor, iar responsabilitatea se identifică dintre verbele folosite în cadrul descrierii scenariilor.

Adevarat. In tehnica CRC clasele sunt identificate pornind de la substantive, intrucat programarea orientata pe obiecte cere ca si clasele sa poata defini entitati ale sistemului, deci ele nu pot fi decat substantive, iar responsabilitatile claselor reprezinta functionalitatea lor redată prin metode, iar aceasta functionalitate este data de verbele din scenariul unei diagrame use case.

43. Relația de compoziție este un caz special de agregare (relație parte-întreg) în care indicăm ca distrugerea clasei “întreg”, atrage după sine și distrugerea obiectelor “parte” conținute de către clasa “întreg”.

Adevarat. Relatia de compositie este o forma de agregare in care componentelete nu pot exista una fara cealalta.

44. Între specificatorul de acces “protected” și criteriul de modularitate al protecției există o legătură strânsă.

Fals. Specificatorul de acces „protected” face ca atributele si metodele unei clase sa fie vazute doar de clasele care mostenesc clasa respectiva. Criteriul de modularitate al protectiei ne spune ca daca apare o eroare de executie, aceasta se limiteaza la doar cateva module. Criteriul se poate asigura si cu specificatorul private, cu pachete etc.

45. În stilul arhitectural Repository diferentele componente care procesează date sunt total independente unele de altele, cu excepția faptului că folosesc același model de date.

Adevarat. Stilul arhitectural Repository este un mod eficient de a distribui mari cantitati de date in care producatorii si consumatorii de date sunt independenti, dar marele compromis este faptul ca folosesc acelasi model de date.

46. Principalul motiv pentru care testarea de integrare de tip Big Bang nu este recomandată este acela ca spre deosebire alte tehnici de testare de integrare aceasta identifică mai puține defecte.

Fals. Testarea de integrare de tip Big Bang combina toate componentelete in avans si testeaza intreg programul. In acest fel se creeaza un haos cu multe erori care la prima vedere nu se leaga. Corectarea acestora se realizeaza greu, dupa aceasta rezulta alte erori si astfel testarea pare sa intre intr-o buclu infinita.

47. Testarea ciclurilor pentru o funcție cu două cicluri imbricate(nested) implică scrierea a două cazuri de test; unul să parcurgă instrucțiunile din ciclul interior, și un al doilea care să testeze instrucțiunile din ciclul exterior.

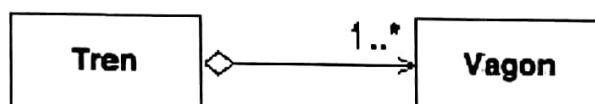
Adevarat. Testarea functiilor cu 2 cicluri imbricate incepe cu testarea buclei din interior care se face ca testarea unei bucle simple pastrand iteratorul buclei exterioare la valoarea minima. Dupa aceasta se trece la testarea celei de a doua bucle in maniera testarii unei bucle simple.

49. Precizati ce stil arhitectural s-ar asocia cel mai bine urmatoarele imagini:

- O ceapa - Layered Architecture intrucat aceasta are o structura stratificata, iar in mijloc se afla lastarul care este cea mai importanta parte a acesteia.
- O linie de productie - conducte si filtre. Conductele reprezinta benzile rulante care ajuta produsele sa se deplaseze prin fabrica, iarfiltrele sunt reprezentate de diferite masinarii care transforma simplele materii prime in produsul finit, pas cu pas.
- Un mediu de dezvoltare integrat (IDE), gen Eclipse, Visual Studio, JBuilder sau un instrument CASE complex - Repository deoarece un mediu de dezvoltare are mai multe unele: compilator, debugger etc, care sunt independente intre ele si care creeaza si folosesc acelasi tip de date.

#### Fundamente de inginerie software

1. Cum se reprezinta intr-o diagrama UML de clasa relatia intre o clasa "Vagon" si o clasa "Tren" considerand ca un tren este compus din unul sau mai multe vagoane si ca un obiect "Vagon" poate fi folosit in relatie cu diferite obiecte "Tren". Precizati cum se numeste acest tip de relatie descris si care au fost indiciile care v-au ajutat sa il identificati.



Raspuns:

Relatia descrisa se numeste **agregare**, si se reprezinta in UML ca in figura de mai jos:  
Relatia intre "Vagon" si "Tren" este in mod clar o relatie de tip "HAS-A" (intreg-partea), adica un obiect "Tren" este compus din obiecte "Vagon". Astfel, am oscilat intre o relatie de agregare si una de componitie. Indiciul din enunt care a facut diferenta este acela ca obiectul "Vagon" pot fi folosite in relatie cu diferite obiecte "Tren", ceea ce inseamna ca viata obiectelor "Vagon" este distincta de cea a obiectelor "Tren". Astfel, relatia este una de agregare.

2. Daca intr-un sistem ar trebui sa favorizati *performanta de timp*, ati opta pentru stilul arhitectural *Layered* (arhitectura stratificata)? Dar daca ar trebui sa favorizati *securitatea*? Justificati succint raspunsul.

Raspuns:

Daca trebuie favorizata performanta de timp, stilul architectural Layered, NU este recomandat. Motivul este acela ca apelarea unui serviciu la nivelul cel mai din exterior (accesibil clientului aplicatiei) implica cel mai adesea o dubla parcurgere a tuturor nivelurilor / straturilor, adica atat pentru transmiterea serviciului, cat si pentru receptarea rezultatului/rezultatelor.

Daca insa trebuie favorizata securitatea stilul architectural Layered, este extrem de indicat! Motivul este acela ca fiecare layer poate oferi un nivel distinct de securitate, ce poate fi

verificat atunci cand acel nivel este accesat. In plus, faptul ca fiecare nivel comunica doar cu nivelul imediat inferior (ca si client) respectiv cu nivelul imediat superior (ca si server) face ca incapsularea datelor, si deci securitatea lor sa fie mai buna.

3. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *La testare blackbox avem nevoie si de cod pentru a verifica daca datele de test acopera toate caile din program?*

**Nota:** raspunsurile “adevarat/fals” neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este fundamental FALSA. Motivul este acela ca testarea blackbox, prin insasi definitia sa, nu implica cunoasterea codului. In testarea blackbox, cazurile de test se scriu strict pe baza “contractului” modului testat, adica a (specificatiilor ?) datelor de intrare, respectiv a celor de iesire.

4. Care este diferența intre urmatoarele tipuri de relatii intre clase: *asociere, agregare, compozitie?*

*Raspuns:*

**Asocierea** reprezinta forma cea mai slaba (si mai generala) in care putem exprima relatia dintre doua clase. Atunci cand spunem ca o relatie este de “asociere” tot ce stim este ca intre cele doua clase exista o relatie.

Atunci cand afirmam ca o relatie intre doua clase este una de **agregare** sau **compozitia**, inseamna ca din descriere cerintelor putem spune ca intre cele doua clase implicate exista o relatie de tip “HAS-A”, o relatie de tip “parte-intreg” (*containment*). Mai departe, distinctia intre agregare si compozitie este de data masura in care obiectele “parte” pot fi *reutilizate* de diferite obiecte “intreg”: daca obiectele parte sunt create si folosite exclusiv de catre un singur obiect, atunci relatia este una de **compozitie**; in caz contrar, daca obiectele “parte” pot fi folosite *shared* de catre mai multe obiecte “intreg” atunci relatia este una de **agregare**. Pe scurt: compozitia este o forma de relatie mai stransa decat agregarea,

5. Enuntati legea lui Brooks, referitoare la extinderea echipei de dezvoltatori in timpul proiectului. De asemenea precizati, argumentand succint, valoarea de adevar a urmatoarei afirmatii: *Intr-un sistem cu o modularitate foarte buna legea lui Brooks nu se aplica pentru ca activitatea de construire a sistemului este o activitate perfect partitionabila.* (**Nota:** precizarea valorii de adevar a afirmatiei, neinsotita de argumentatie nu se puncteaza)

*Raspuns:*

Legea lui Brooks afirma ca adaugarea de programatori (ingineri software) la un proiect aflat in intarziere, va face ca proiectul sa intarzie si mai mult. Motivul este acela, ca un realizarea unui proiect software nu este o activitate perfect partitionabila, ci dimpotriva una ce implica foarte multe interactiuni. Iar adaugarea unor noi programatori ar implica o repartitionare (uneiori imposibil de realizat) a task-urilor din proiect. In plus fiecare om adus in plus, aduce dupa sine relatii de comunicare mai complexe.

In sisteme cu modularitate foarte buna, creste intr-adevar gradul de partitionare, si deci capacitatea de diviziune a task-urilor intre membrii echipei. Totusi, legea lui Brooks ramane valabila, din 2 motive: (i) surplusul de comunicare este semnificativ indiferent de modularitate si (ii) oricat de modular ar fi proiectat un sistem, totusi nu se poate ca sistemul sa devina unul perfect partitionabil.

6. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *Testele whitebox nu pot garanta ca toate erorile dintr-o functie vor fi detectate.*

*dar daca sunt dublate de teste blackbox se poate garanta ca vor fi gasite toate erorile din respectiva functie.*

**Nota:** raspunsurile “adevarat/fals” neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este fundamental FALSA. Nici o forma de testare, si nici o combinatie de tehnici de testare nu pot vreodata garanta absenta bug-urilor. Asa cum spunea Dijkstra, testarea poate evidenta doar prezenta bug-urilor, dar nu poate niciodata garanta absenta lor. Desigur, utilizarea concertata a mai multor tehnici de testare contribuie la reducerea numarului de bug-uri, dar ceea ce face enuntul de mai sus, complet fals este cuvantul “garanta”.

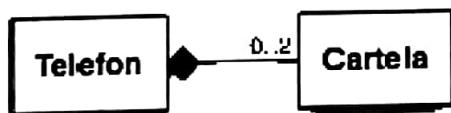
7. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *Testarea de regresie este bazata pe ideea ca atunci cand se opereaza schimbari intr-un modul, imediat dupa modificarile efectuate de testare trebuie concentrate exclusiv asupra modulului modificat.*

**Nota:** raspunsurile “adevarat/fals” neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este FALSA. Ideea de baza in testarea de regresie este reluarea unui subset de teste care au fost rulate anterior. Desigur, aceste teste se refera la modulul modificat, dar in nici un caz **exclusiv** asupra sa. In testarea de regresie, pe langa testele ce vizeaza modulul modificat se mai reiau doua clase de teste: (i) un set de teste reprezentative prin care se retesteaza principalele functionalitati ale sistemului; (ii) un set de teste ce vizeaza modulele direct conectate cu modul modificat, avand in vedere ca impactul schimbarii ar putea sa le fi influentat in primul rand pe acestea.

8. Precizati, si argumentati succint daca relatia descrisa de diagrama UML de mai jos modeleaza sau nu cazul unui telefon dual-sim (care poate tine simultan doua cartele) cu cartele care pot fi oricand reutilizate si in alt telefon.



*Raspuns:*

Relatia descrisa de diagrama UML NU modeleaza intrutotul ceea ce sustine enuntul de mai sus! Intr-adevar relatia de mai sus, indica faptul ca un obiect telefon are in compositia sa un numar de cartele ce poate varia intre niciuna si doua cartele... dar partea care nu este bine interpretata se refera la “reutilizarea oricand in alt telefon”. Relatia din figura este una de compositie, marcata prin rombul negru, si deci obiectele “Cartela” sunt construite specific pentru un anumit obiect “Telefon”, putand fi folosite doar de catre acel obiect.

9. Considerand urmatoarele doua procese de dezvoltare: *Waterfall* si respectiv *Extreme Programming*, precizati si argumentati succint pe care l-ati alege daca ati fi *pe rand* in urmatoarele cazuri:

*Cazul 1:* trebuie sa reimplementati de la zero, un sistem complex (cca. 1 milion de linii de cod) pentru gestiunea personalului si a salariilor, care sa inlocuiasca sistemul existent in prezent, fara nici o modificarile de cerinte.

*Cazul 2:* aveți de construit un sistem pentru plata taxelor si impozitelor pentru tara imaginara Ainamor cu o legislatie haotica si in permanenta modificarile.

**Nota:** cele 2 cazuri sunt complet independente.

*Raspuns:*

Pentru “Cazul 1” as alege procesul de dezvoltare Waterfall din urmatoarele motive: (i) e vorba de un sistem de mari dimensiuni, ce implica deci echipe de mari dimensiuni, ceea ce

face ca un proces mai formal sa fie dezirabil; in plus (ii) cerintele sistemului sunt f. bine cunoscute, si clar fixate, deci nu exista riscul de a trebui sa reluam toate fazele procesului (principalul dezavantaj la Waterfall) datorita unei schimbari de cerinte.

Pentru "Cazul 2" as alege in mod evident Extreme Programming datorita cerintelor in permanenta schimbare. Din acest motiv avem nevoie de un proces care sa se poata adapta bine, si mai ales rapid schimbarilor de cerinte, iar Waterfall nu corespunde acestor cerinte. Extreme Programming, are avantajul ca este un proces iterativ si incremental, si in plus are durata iteratiilor scurte.

10. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *Procesul de dezvoltare in cascada (waterfall) este recomandat in majoritatea proiectelor intrucat, datorita structurii sale rigide, poate constrange clientul sa descopere, inca de la inceput, toate posibilele cauze de schimbare din sistem.*

**Nota:** raspunsurile "adevarat/fals" neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este FALSA. Procesul de dezvoltare Waterfall este recomandat in foarte putine cazuri, tocmai datorita structurii sale rigide, nebazata pe iteratii si incremente, ce impiedica o adaptare rapida la schimbari de cerinte. Cu atat mai putin este recomandat procesul Waterfall atunci cand vine vorba de un sistem unde cerintele nu sunt bine intelese. Structura rigida a lui Waterfall NU poate sub nici o forma "constrange" (determina) clientul sa descopere de la inceput toate posibilele cauza de schimbare... Aceasta este o utopie. Chiar si in sisteme unde cerintele sunt clar specificate de la inceput, pot interveni in timp schimbari ce nu pot fi sub nici o forma prevazute. De aceea, trebuie favorizate procesele de dezvoltare (iterative si incrementale) care pot face fata schimbarilor neasteptate.

1. 50% din timpul si costurile totale pentru un proiect ar trebui alocat testarii, dar in multe cazuri in realitate, procentajul este mai mic.

Raspuns: Afirmatia este falsa. Conform regulilor lui Brooks, testarea ocupa intr-adevar 50% atat dpdv al timpului cat si al costului, iar in realitate, acest lucru este respectat. Prin testare se descopera eventuale erori in program, iar testarea insuficienta ar putea avea consecinte dezastruoase datorita anumitor erori nedepistate.

2. Forma reala a curbei ratei defectarii SW-ului este datorata faptului ca cerintele nu sunt intelese clar de la bun inceput, ceea ce duce la necesitatea de schimbari continue asupra sistemului.

Raspuns: Afirmatia este falsa. Schimbarile continue asupra sistemului constituie intr-adevar principalul motiv pentru care curba are acea forma, insa acestea pot aparea chiar daca cerintele au fost intelese perfect initial. De exemplu, s-ar putea ca pe parcurs una sau mai multe cerinte sa se schimbe, ceea ce evident inseamna ca va trebui ca sistemul sa fie modicat corespunzator.

3. Conceptul de time unboxing spune ca sarcinile alocate pt o anumita iteratie trebuie terminate in cadrul iteratiei in cauza

Raspuns: Afirmatia este falsa. In cazul in care sarcinile asignate unei anumite iteratii nu sunt finalizate la timp, ceea ce a ramas va fi inclus in iteratia urmatoare. Intervalul de timp pentru iteratii trebuie sa ramana intotdeauna fix, deoarece altfel echipa de programatori ar avea dificultati in a-si da seama de motivul pentru care nu a reusit sa finalizeze tot ce si-a propus in cadrul iteratiei in cauza.

4. In procesele de dezvoltare agile nu este necesar sa intelegem de la bun inceput cerintele sistemului.

Raspuns: Afirmatia este falsa. Chiar daca metodele agile se caracterizeaza prin adaptabilitatea rapida la modificari si prin faptul ca schimbarile sunt mereu bine-venite, acest lucru nu inseamna ca nu trebuie sa intelegem de la bun inceput ceea ce se cere. Daca pe parcurs programatorii si-ar da seama ca anumite cerinte au fost intelese gresit, acest lucru ar necesita modificarile care ar constitui o risipa de bani si de timp.

5. Daca avem la dispozitie diagrame UML foarte explicite pt un sistem, putem deduce diagramele de secventa pt acel sistem, adica toate interactiunile posibile dintre obiecte

Raspuns: Afirmatia este falsa. Diagramele UML de clasa descriu sistemul dpdv structural - prezinta clasele de obiecte impreuna cu atributiile si operatiile lor, precum si relatiile existente intre obiecte. Diagramele de secventa, pe de alta parte, descriu comportamentul sistemului. Aceste diagrame nu se pot deduce decat cu ajutorul codului sursa, iar acesta nu apare pe diagrama de clasa.

6. Un sistem nu poate fi actor pentru un alt sistem pentru ca acest lucru ar implica accesul la informatiile din sistemul mare, ceea ce incalca incapsularea.

Raspuns: Afirmatia este falsa. Actorul este o entitate care interactioneaza cu sistemul, iar aceasta entitate poate fi orice. Un om (actor) care doreste sa retraga o suma de bani

de la un bancomat (sistem) nu stie nimic despre detaliiile de implementare ale aparatului. Prin urmare, acest lucru se aplica si in cazul in care actorul nostru este chiar un alt sistem.

**7. Pentru a evita proliferarea claselor, clasa Family trebuie implementata ca (in cursul 4, slide-ul 4, varianta din dreapta.)**

Raspuns: Afirmatia este falsa. Modul in care trebuie implementate clasele depinde de ceea ce ne cere sistemul. De exemplu, daca o functionalitate ceruta de sistem ar fi "schimbaScutece()", atunci am opta pentru cea de-a doua varianta, deoarece aceasta operatie ar putea fi realizata de orice persoana din familie (mama, tata, frate, etc.). Daca insa sistemul are functionalitatea "naste()", cum mama este singura care poate face asta, varianta potrivita ar fi, evident, cea din stanga.

**8. Folosirea constantelor globale incalca criteriul de modularitate al continuitatii.**

Raspuns: Afirmatia este adevarata. Daca dorim sa dam variabilei globale alta valoare, acest lucru ar inseamna ca ar trebui sa modificam valoarea peste tot pe unde aceasta apare, ceea ce ar putea contraveni criteriului de continuitate in cazul in care variabila noastra ar aparea in multe locuri.

**9. Stilul arhitectural pipes and filters este avantajos dpdv al timpului**

Raspuns: Afirmatia este fundamental falsa. Trecerea prin fiecare filtru necesita parsarea si analizarea intregului flux de date, ceea ce este consumator de timp.

**10. Un program care nu are cicluri (for, while) are numarul ciclomatic=1**

Raspuns: Afirmatia este falsa. Numarul ciclomatic ne arata numarul de "cai" de executie posibile pentru un anumit program. Ca exemplu vom lua o bucată de cod constant dintr-o secvență if-then-else. Acest program nu are cicluri for sau while, insă există două posibilități de executie: ramura de if, respectivă cea de else, adică numarul ciclomatic =2.

**11. Principalul dezavantaj al testarii top down integration este crearea de drivere si stub-uri**

Raspuns: Afirmatia este parțial adevarata. Top-down integration presupune testarea sistemului de sus in jos, incepand cu modulul principal, si coborand in ierarhie spre modulele de nivel inferior. Stuburile sunt niste inlocuitori pentru modulele care nu au fost inca testate sau implementate, iar crearea lor, necesara pentru acest tip de testare, este consumatoare de timp, deci constituie un dezavantaj. Driverele sunt niste programe care preiau datele de intrare pentru un test, le proceseaza, si tiparesc rezultatele testului, insă ele trebuie create indiferent de procesul de testare ales, deci nu spune ca acestea constituie un dezavantaj specific testarii top-down integration.

FALS                  Drept de rosu de la scrierile mele

de tip  $\frac{1}{2}$

1. **Forma reala a curbei ratei defectarii software-ului este datorata faptului ca cerintele nu sunt intelese clar de la bun inceput, ceea ce duce la necesitatea de schimbari continue asupra sistemului.**

Fals. Curba reala difera de cea ideală din cauza apariției schimbărilor repetitive în soft. Schimbările apar atunci când se efectuează lucrări de menținere asupra softului sau modificări asupra softului.

2. **50% din timp și costuri ar trebui alocate testării, dar în multe cazuri procentajul alocat testării este mai mic.**

Fals. Testarea reprezintă într-adevar jumătate din costuri și din timp, dar aceste estimări nu sunt fictive, deoarece chiar atât reprezintă, trebuie să facem multă testare pe lângă partea de cod pentru a ne asigura că produsul software este bun și face ce trebuie.

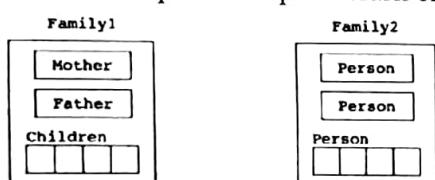
3. **Conceptul de time boxing spune că sarcinile alocate pentru o anumita iteratie trebuie terminate în cadrul iteratiei în cauză.**

Adevarat. Conceptul de time boxing aloca o perioadă fixă fiecărei iterării, numită time box. În cadrul acestei perioade de timp sarcinile alocate în cadrul iterării trebuie terminate. În cazul în care nu se termină task-urile alocate deadline-ul nu se modifică, mai degrabă se amână niste task-uri pentru time box-ul viitor.

4. **Dacă lucram cu procese de dezvoltare agile, asta înseamnă că nu trebuie să intelegem clar cerințele sistemului.**

Fals, cerințele sistemului trebuie intelese indiferent de procesele de dezvoltare, dar în cazul proceselor de dezvoltare agile acestea se pot modifica de la o iteratie la alta sau se mai pot adăuga alte cerinte de la o iteratie la alta, important este că acestea să nu se schimbe în timpul unei iterării.

5. **Pentru a evita problema proliferării claselor, ar trebui să implementăm clasa Family ca mai jos.**



Adevarat. Pentru a evita proliferarea claselor trebuie să fim atenți la comportamentul acestora. El este cel care decide: dacă comportamentul difera avem clase diferite, dacă nu difera avem roluri diferite ale acelasi clase.

6. **Dacă avem diagrame de clasa UML explicite pentru un sistem, putem deduce diagramele de secvență pentru acel sistem, adică toate interacțiunile posibile dintre obiecte.**

Fals. Diagramalele de clasa ne arată structura sistemului, atributele și metodele fiecărei clase, pe când diagramalele de secvență modelează scenariile dintr-un sistem. Acestea nu se pot deduce unele din altele.

**7. Un sistem nu poate fi actor pentru un alt sistem pentru ca acest lucru ar implica accesul la informatiile din sistemul mare, ceea ce incalca incapsularea.**

Fals, un sistem poate fi actor pentru un alt sistem, intrucat acesta interacționează cu interfața acestuia și nu are acces la codul acestuia, astfel nu se incalca principiul incapsularii.

**8. Stilul architectural *Pipes and Filters* este avantajos din punct de vedere al timpului.**

Fals. Aceasta este format din filtre care sunt independente unele față de celelalte și care prelucră datele și din conducte care sunt cai prin care datele circulă de la filtru la filtru. Filtrele nu au un format comun de date, iar din aceasta cauza fiecare filtru trebuie să facă "parse" și "un-parse" datelor, lucru care dă overhead și, implicit, la pierderea timpului.

**9. Un program care nu are cicluri (for, while) are numarul ciclomatic=1.**

Fals. Numarul ciclomatic este influențat de numărul deciziilor simple din cadrul unui program. Instrucțiunile de călcare de tip for/while contin și ele o astfel de instrucțiune, asadar numărul ciclomatic în cazul în care nu avem instrucțiuni de călcare în program este influențat doar de numărul deciziilor simple din cadrul acestuia.

**10. Principalul dezavantaj al testării top down integration este crearea de drivere și stub-uri .**

Fals. În cazul testării de tip top-down se creează doar stub-uri, care imită comportamentul metodelor apelate din funcția testată. Driver-ele sunt create la testarea de tip bottom-up și reprezintă un program principal simplu din care se apelează funcția testată.

**11. Folosirea constantei globale incalca criteriul de modularitate și continuitate.**

~~Fals. Folosirea constantei globale nu incalca principiul continuitatii.~~

**12.Curba reală a evoluției în timp a ratei de defecte (Failure Rate) diferă față de cea ideală în principal din cauza lipsei de experiență a programatorilor.**

Fals, deoarece curba reală diferește de cea ideală din cauza apariției schimbărilor repetitive în soft. Schimbările apar atunci când se efectuează lucrări de menenanță asupra softului. Lipsa de experiență a programatorilor nu este un motiv principal de apariție a erorilor în soft.

**13. Legile evoluției software-ului ale lui Lehman nu se aplică la sisteme ale căror cerințe sunt perfect înțelese de la bun început, pentru că altfel de sisteme nu au nici un motiv să evolueze.**

Fals, deoarece legile evoluției software se aplică oricărui sistem indiferent dacă cerințele sunt înțelese sau nu de la bun început. Sistemele software oricum evoluează datorită schimbărilor inconjurătoare.

14. În procesul de dezvoltare Scrum, Burndown Char ne arată evoluția efortului de-a lungul întregii istorii a proiectului, mai exact oferă informații despre Sprint-urile deja încheiate, precum și numărul de Task-uri finalizate până în prezent în întregul proiect.

Fals. Burndown Chart ne arată evoluția efortului și numărul de task-uri finalizate și de finalizat de-a lungul unui Sprint, perioada în care se realizează un Increment dintr-un proiect. Burndown Chart-ul nu arată evoluția efortului pentru întregul proiect.

15. O problemă importantă a proceselor de dezvoltare iterative este aceea că adesea trebuie modificat codul care a fost scris într-o iterare anterioară, și uneori anumite porțiuni de cod trebuie chiar șterse, ceea ce reprezintă o pierdere/risipă.

~~Fals - este una hinc decât să avem soft prost~~  
Corect. Aceasta este o problema importantă a proceselor de dezvoltare iterative și este o risipă, în schimb, putem să refacem codul decât să pierdem timpul modificând același cod. Singurul cod care se pastrează este codul funcțional care face ceva util în cadrul softului.

16. În diagramele de UML de Use-Case relațiile <<extends>> și <<include>> sunt foarte asemănătoare și pot fi folosite interschimbabil încărcăt ambele sunt folosite pentru a da "factor comun" descrierea unei anumite funcționalități.

Fals. Relațiile <<extends>> și <<include>> sunt ca și cum se poate de diferențe. Prima reprezintă un caz exceptional al unui use case, pe când cea de-a două factorizează un comportament comun al mai multor use case-uri.

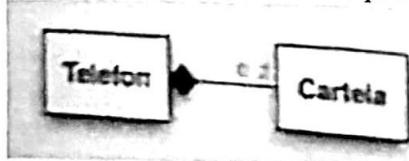
17. Diagramele UML de secvență (sequence diagrams) sunt folosite ca punct de plecare pentru sesiunile de CRC Cards în urma cărora se construiesc diagramele UML de clase (class diagrams).

Fals. Punctul de plecare pentru a crea un CRC card sunt diagramele use case care redau un scenariu identificând actorii și interacțiunile lor cu sistemul. În urma creării cardurilor CRC se realizează diagramele de clasa și apoi diagramele de secvență.

18. Într-o diagramă de secvență UML, toate obiectele care sunt instanțe ale aceleiași clase trebuie "comasate" într-un singur dreptunghi; adică nu este admis sau recomandat să reprezentăm fiecare obiect ca un dreptunghi separat, pentru că altfel s-ar încărca prea mult diagrama.

Fals. Toate instantele unei clase trebuie desenate în dreptunghiuri separate, întrucât fiecare obiect poate avea un comportament diferit chiar dacă sunt instante ale aceleiasi clase.

19. Relația din figura de mai jos modelează cazul unui telefon dual-sim (care poate ține simultan două cartele) cu cartele care pot fi oricând reutilizate și în alt telefon.



*ASOCIARE => reutilizare  
COMPOUNERE => nu reutilizare*

~~Adevarat~~. Avem clasa Telefon și clasa Cartela cu o relație de compunere între ele. Într-adevar, un obiect Cartela nu poate exista fără un obiect Telefon, dar putem avea instante ale clasei Cartela și în alte obiecte Telefon.

20. Un sistem care respectă criteriul de modularitate al Decompozabilității îl va respecta aproape sigur și pe cel al Compozabilității întrucât acesta din urmă se referă la posibilitatea compunerii unui sistem din module.

Fals. Dacă un sistem respectă criteriul de modularizare al decompozabilității nu înseamnă că îl va respecta și pe cel al compozabilității. Exemplu cu IKEA și LEGO.

21. Un avantaj principal al stilului arhitectural Stratificat (Layered Architecture) este asigurarea criteriului de modularitate al Protecției.

Adevarat. În cazul stilului arhitectural stratificat când se modifică interfața unui layer sau se adaugă noi facilități este afectat doar layer-ul adjacente, nu întregul sistem.

22. Ideea de bază în testarea whitebox este aceea de a ne asigura că cel puțin privită individual o funcție nu are nici un bug și nici o instrucțiune, întrucât toate instrucțiunile sunt verificate cu câte cel puțin un test.

Adevarat. Testarea de tip whitebox verifică toate instrucțiunile dintr-o funcție cu cel puțin un test. Astfel, funcția cel puțin privită individual nu are niciun bug.

23. Partițiile echivalente se folosesc la sisteme cu foarte multe funcții lungi și complexe, și reprezintă grupuri de funcții care sunt asemănătoare din punct de vedere al testării. Astfel, din fiecare grup (partiție) se vor alege una sau mai multe funcții pentru care se vor scrie suite de teste blackbox.

Fals. Partițiile echivalente se folosesc la sistemele cu domeniul datelor de intrare foarte mare și reprezintă clase de input-uri asemănătoare din punct de vedere al testării. Astfel, pentru fiecare partitie se aleg cazuri de teste pentru valorile corespunzătoare marginilor și mijlocului.

**24. Legile evoluției software-ului enunțate de către Lehman ne spun că nu există nici o modalitate de a încetini declinul calității software-ului.**

Fals. Legile scaderii calitatii a lui Lehman spun ca putem incetini procesul de deteriorare al software-ului daca il adaptam in mod continuu la schimbarile inconjuratoare.

**25. Printr-o proiectare riguroasă, implementarea software-ului poate fi transformată într-o activitate aproape perfect partaționabilă.**

Fals, o proiectare riguroasa ajuta intr-adevar la partitionarea implementarii softului, dar nu garanteaza o partitionare perfecta. Asta depinde de natura produsul, daca prin natura lui raportat la tehnologiile existente, el poate fi sau nu partitionat cat mai bine.

**26. Procesul de dezvoltare Extreme Programming spune că majoritatea efortului trebuie investit în programare și mai puțin în alte activități conexe cum ar fi captarea cerințelor sau testare, întrucât până la urmă produsul software propriu-zis este creat prin programare.**

Fals, Extreme Programming nu se refera la programarea in sensul de a scrie cod si atat. Este o metodologie bazata pe Agile care abordeaza extrem dezvoltarea iterativa. Este scrisa o noua versiune foarte des, incrementii sunt livrati clientului la fiecare doua trei saptamani, iar constructia e aprobată doar daca testele sunt trecute cu succes.

**27. Deși în procesele de dezvoltare iterative și incrementate trebuie uneori să schimbă sau chiar să rescrisce complet fragmente semnificative de cod ce au fost scrise în iterările anterioare, acestea nu reprezintă o pierdere întrucât schimbările sunt inevitabile.**

~~Fals~~ Aceasta este o problema importanta a proceselor de dezvoltare iterative si este o risipa, in schimb, putem sa refacem codul decat sa pierdem timpul modificand acelasi cod. Singurul cod care se pastreaza este codul functional care face ceva util in cadrul softului.

**28. Tehnica de analiza cerințelor folosind Use-Case-uri este specifică procesului de dezvoltare Waterfall pentru că aici cerințele trebuie analizate riguros la începutul proiectului.**

Fals. Intr-adevar, procesul de tip Waterfall se foloseste atunci cand cerintele sunt foarte bine cunoscute de la inceput, dar asta nu inseamna ca folosirea use case-urilor este specifica doar acestui tip de proces.

**29. Doi sau mai mulți actori nu pot fi asociați (adică nu pot interacționa) cu același use-case pentru că aceasta ar însemna că sunt redundanți.**

Fals. Doi actori pot avea acelasi use case printre altele atata timp cat interactionarea lor cu sistemul este diferita in ansamblu. (Nu interactioneaza cu sistemul pentru aceeasi functionalitate a sistemului).

**30. Într-o diagramă de secvență se recomandă ca dacă apar mai multe instanțe ale aceleiași clase acestea să fie reprezentate într-un singur element pentru a nu se încărca excesiv diagrama.**

Fals. Toate instantele unei clase trebuie desenate în dreptunghiuri separate, intrucât fiecare obiect poate avea un comportament diferit chiar dacă sunt instante ale aceleiași clase.

**31. Pentru a evita problema proliferării claselor prin modelare ca și clase a entităților externe sistemului trebuie să ținem cont că sunt clase doar acele entități care apelează alte entități (clase) din sistem.**

Fals. Clasele sunt aceleia care sunt apelate (primesc mesaje), nu cele care apelează (transmit mesaje).

**32. Un corp de mobilă modular care vine împachetat pe bucăți și care trebuie să îl asamblăm/compunem (gen IKEA) a fost proiectat urmărind în special criteriul de modularitate al compozabilității.**

Fals. Un corp de mobila modular de la Ikea a fost proiectat în astă fel încât să se poată realiza din acele piese numai un singur corp și numai într-un anumit fel. Acest lucru este în contradicție cu principiul modular al compozibilității care spune că putem compune un sistem modular în mod liber și cum ne dorim.

**33. Importanța criteriilor și regulilor de modularitate este cu atât mai mare cu cât anvergura sistemului software proiectat este mai mare, cu alte cuvinte: importanța modularizării este proporțională cu dimensiunea sistemului software.**

Adevarat. Într-un soft mic aproape că nu avem nevoie de modularizare pentru că nu avem în ce bucati să spargem. Pe când într-un soft mare e important să spargem pe bucati pentru structura frumoasă și pentru a realizare o comunicare între componente ușor de urmarit și componente ușor de reutilizat.

**34. Dacă dorim să verificăm cât mai timpuriu principalele puncte de control și decizie din sistem vom alege testarea de integrare de tip Bottom-Up.**

Fals. Testarea de integrare Bottom-Up verifică timpuriu procesarea de date de nivel scăzut. Pentru a verifica că mai timpuriu principalele puncte de control și decizie din sistem vom alege testarea de integrare de tip Top-Down.

**35. Valoarea complexității ciclomatrice a unei funcții nu este influențată de numărul de instrucțiuni de ciclare (for/while) întrucât singurele instrucțiuni care incrementează valoarea complexității ciclomatrice sunt cele de decizie de tip if-else.**

Fals. Valoarea complexității ciclomatrice este influențată atât de deciziile de tip if/else, cât și de numărul de instrucțiuni de ciclare (for/while), deoarece și aceste instrucțiuni contin câte o decizie simplă.

**36. Curba defectelor poate fi făcută să tindă în timp spre zero dacă cerințele sunt bine înțelese de la început, și dacă se aplică sistematic metode de testare eficiente.**

Fals. Chiar daca sunt bine intelese cerintele de la inceput si se aplica metode de testare eficiente, in timp, din cauza schimbarilor repeatate tot vor aparea erori in sistem, deci curba defectelor nu va tinde in timp spre 0.

**37. Utilizarea eficientă a instrumentelor CASE dedicate programării poate reduce semnificativ costurile totale ale proiectului având în vedere că într-un proiect software o parte semnificativă a costurilor sunt legate de activitatea de codare.**

Fals. Utilizarea eficienta CASE reduce costurile, dar ele se folosesc nu doar pentru partea de codare, ci si pentru alte etape ale procesului precum specificatii, design, testare, debugging. De-asemenea, o mare parte din costuri nu se duc pe partea de codare, ci pe partea de testare si planificare.

**38. În procesul de dezvoltare Waterfall se face doar un singur tip de activitate la un moment dat, spre deosebire de cele iterative unde într-o iteratie trebuie efectuate toate tipurile de activități.**

Adevarat. Procesul de dezvoltare Waterfall imparte proiectul in mai multe activitati ce sunt rezolvate pe rand. De exemplu, se realizeaza analiza si definirea cerintelor, dupa ce se incheie aceasta faza se realizeaza design-ul de sistem si software si asa mai departe.

**39. În procesul Rational Unified Process (RUP), deși este permisă efectuarea mai multor tipuri de activități în interiorul unei faze, se recomandă ca numărul de activități desfășurate în paralel să fie limitat.**

Fals. Intr-adevar, in cazul procesului Rational Unified Process putem efectua mai multe tipuri de activitati simultan in interiorul unei faze, dar nu exista nicio limitare cu privire la numarul de procese desfasurate in paralel. De verificat

**40. Use-Case-urile de tip Fish Level se folosesc atunci când vrem să detaliem fiecare acțiune principală din cadrul unui use-case de tip Sea Level.**

Fals. Use Case-urile de tip Sea Level ne arata cum userul interactioneaza cu sistemul, interactiunile sunt majore, iar scopul este bine precizat. Use Case-urile de tip Fish Level sunt acele use case-uri care se dau factor comun din use case-urile Sea-Level (cu <<include>>).

**41. Știm că am descoperit toate Use-Case-urile dintr-un sistem atunci când toate cerințele funcționale sunt acoperite de use-case-uri Sea Level, fiecare dintre acestea trebuind să fie conectate cu cel puțin un actor.**

Adevarat. Use case-urile de tip Sea Level prin definitie reprezinta interacciuni majore ale user-ilor cu sistemul cu un scop precis. Astfel daca toate cerintele funktionale sunt acoperite de Use Case-uri de tip Sea Level am descoperit toate Use Case-urile din

sistem. Si da, trebuie ca fiecare use case sea level sa fie legat la un user pentru ca altfel el nu are sens, un use case presupune existenta cel putin a unui user care sa-l foloseasca.

42. În tehnica CRC clasele sunt identificate pornind de la substantivele din descrierea use-case-urilor, iar responsabilitatea se identifică dintre verbele folosite în cadrul descrierii scenariilor.

Adevarat. In tehnica CRC clasele sunt identificate pornind de la substantive, intrucat programarea orientata pe obiecte cere ca si clasele sa poata defini entitati ale sistemului, deci ele nu pot fi decat substantive, iar responsabilitatile claselor reprezinta functionalitatea lor redată prin metode, iar aceasta functionalitate este data de verbele din scenariul unei diagrame use case.

43. Relația de compoziție este un caz special de agregare (relație parte-întreg) în care indicăm ca distrugerea clasei "întreg", atrage după sine și distrugerea obiectelor "parte" conținute de către clasa "întreg".

Adevarat. Relatia de compositie este o forma de agregare in care componentelete nu pot exista una fara cealalta.

44. Între specificatorul de acces "protected" și criteriul de modularitate al protecției există o legătură strânsă.

Fals. Specificatorul de acces „protected” face ca atributele si metodele unei clase sa fie vazute doar de clasele care mostenesc clasa respectiva. Criteriul de modularitate al protectiei ne spune ca daca apare o eroare de executie, aceasta se limiteaza la doar cateva module. Criteriul se poate asigura si cu specificatorul private, cu pachete etc.

45. În stilul arhitectural Repository diferentele componente care procesează date sunt total independente unele de altele, cu excepția faptului că folosesc același model de date.

Adevarat. Stilul arhitectural Repository este un mod eficient de a distribui mari cantitati de date in care producatorii si consumatorii de date sunt independenti, dar marele compromis este faptul ca folosesc acelasi model de date.

46. Principalul motiv pentru care testarea de integrare de tip Big Bang nu este recomandată este acela ca spre deosebire alte tehnici de testare de integrare aceasta identifică mai puține defecte.

Fals. Testarea de integrare de tip Big Bang combina toate componentelete in avans si testeaza intreg programul. In acest fel se creeaza un haos cu multe erori care la prima vedere nu se leaga. Corectarea acestora se realizeaza greu, dupa aceasta rezulta alte erori si astfel testarea pare sa intre intr-o bucla infinita.

47. Testarea ciclurilor pentru o funcție cu două cicluri imbricate(nested) implică scrierea a două cazuri de test; unul să parcurgă instrucțiunile din ciclul interior, și un al doilea care să testeze instrucțiunile din ciclul exterior.

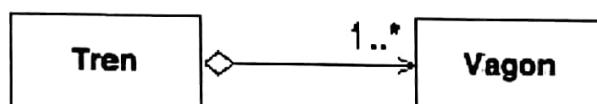
Adevarat. Testarea functiilor cu 2 cicluri imbricate incepe cu testarea buclei din interior care se face ca testarea unei bucle simple pastrand iteratorul buclei exterioare la valoarea minima. Dupa aceasta se trece la testarea celei de a doua bucle in maniera testarii unei bucle simple.

49. Precizati ce stil arhitectural s-ar asocia cel mai bine urmatoarele imagini:

- O ceapa - Layered Architecture intrucat aceasta are o structura stratificata, iar in mijloc se afla lastarul care este cea mai importanta parte a acesteia.
- O linie de productie - conducte si filtre. Conductele reprezinta benzile rulante care ajuta produsele sa se deplaseze prin fabrica, iarfiltrele sunt reprezentate de diferite masinarii care transforma simplele materii prime in produsul finit, pas cu pas.
- Un mediu de dezvoltare integrat (IDE), gen Eclipse, Visual Studio, JBuilder sau un instrument CASE complex - Repository deoarece un mediu de dezvoltare are mai multe unele: compilator, debugger etc, care sunt independente intre ele si care creeaza si folosesc acelasi tip de date.

#### Fundamente de inginerie software

1. Cum se reprezinta intr-o diagrama UML de clasa relatia intre o clasa "Vagon" si o clasa "Tren" considerand ca un tren este compus din unul sau mai multe vagoane si ca un obiect "Vagon" poate fi folosit in relatie cu diferite obiecte "Tren". Precizati cum se numeste acest tip de relatie descris si care au fost indiciile care v-au ajutat sa il identificati.



Raspuns:

Relatia descrisa se numeste **agregare**, si se reprezinta in UML ca in figura de mai jos:  
Relatia intre "Vagon" si "Tren" este in mod clar o relatie de tip "HAS-A" (intreg-partea), adica un obiect "Tren" este compus din obiecte "Vagon". Astfel, am oscilat intre o relatie de agregare si una de componitie. Indiciul din enunt care a facut diferenta este acela ca obiectul "Vagon" pot fi folosite in relatie cu diferite obiecte "Tren", ceea ce inseamna ca viata obiectelor "Vagon" este distincta de cea a obiectelor "Tren". Astfel, relatia este una de agregare.

2. Daca intr-un sistem ar trebui sa favorizati *performanta de timp*, ati opta pentru stilul arhitectural *Layered* (arhitectura stratificata)? Dar daca ar trebui sa favorizati *securitatea*? Justificati succint raspunsul.

Raspuns:

Daca trebuie favorizata performanta de timp, stilul architectural Layered, NU este recomandat. Motivul este acela ca apelarea unui serviciu la nivelul cel mai din exterior (accesibil clientului aplicatiei) implica cel mai adesea o dubla parcurgere a tuturor nivelurilor / straturilor, adica atat pentru transmiterea serviciului, cat si pentru receptarea rezultatului/rezultatelor.

Daca insa trebuie favorizata securitatea stilul architectural Layered, este extrem de indicat! Motivul este acela ca fiecare layer poate oferi un nivel distinct de securitate, ce poate fi

verificat atunci cand acel nivel este accesat. In plus, faptul ca fiecare nivel comunica doar cu nivelul imediat inferior (ca si client) respectiv cu nivelul imediat superior (ca si server) face ca incapsularea datelor, si deci securitatea lor sa fie mai buna.

3. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *La testare blackbox avem nevoie si de cod pentru a verifica daca datele de test acopera toate caile din program?*

**Nota:** raspunsurile “adevarat/fals” neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este fundamental FALSA. Motivul este acela ca testarea blackbox, prin insasi definitia sa, nu implica cunoasterea codului. In testarea blackbox, cazurile de test se scriu strict pe baza “contractului” modului testat, adica a (specificatiilor ?) datelor de intrare, respectiv a celor de iesire.

4. Care este diferența intre urmatoarele tipuri de relatii intre clase: *asociere, agregare, compozitie?*

*Raspuns:*

**Asocierea** reprezinta forma cea mai slaba (si mai generala) in care putem exprima relatia dintre doua clase. Atunci cand spunem ca o relatie este de “asociere” tot ce stim este ca intre cele doua clase exista o relatie.

Atunci cand afirmam ca o relatie intre doua clase este una de **agregare** sau **compozitia**, inseamna ca din descriere cerintelor putem spune ca intre cele doua clase implicate exista o relatie de tip “HAS-A”, o relatie de tip “parte-intreg” (*containment*). Mai departe, distinctia intre agregare si compozitie este de data masura in care obiectele “parte” pot fi *reutilizate* de diferite obiecte “intreg”: daca obiectele parte sunt create si folosite exclusiv de catre un singur obiect, atunci relatia este una de **compozitie**; in caz contrar, daca obiectele “parte” pot fi folosite *shared* de catre mai multe obiecte “intreg” atunci relatia este una de **agregare**. Pe scurt: compozitia este o forma de relatie mai stransa decat agregarea,

5. Enuntati legea lui Brooks, referitoare la extinderea echipei de dezvoltatori in timpul proiectului. De asemenea precizati, argumentand succint, valoarea de adevar a urmatoarei afirmatii: *Intr-un sistem cu o modularitate foarte buna legea lui Brooks nu se aplica pentru ca activitatea de construire a sistemului este o activitate perfect partitionabila.* (**Nota:** precizarea valorii de adevar a afirmatiei, neinsotita de argumentatie nu se puncteaza)

*Raspuns:*

Legea lui Brooks afirma ca adaugarea de programatori (ingineri software) la un proiect aflat in intarziere, va face ca proiectul sa intarzie si mai mult. Motivul este acela, ca un realizarea unui proiect software nu este o activitate perfect partitionabila, ci dimpotriva una ce implica foarte multe interactiuni. Iar adaugarea unor noi programatori ar implica o repartitionare (uneiori imposibil de realizat) a task-urilor din proiect. In plus fiecare om adus in plus, aduce dupa sine relatii de comunicare mai complexe.

In sisteme cu modularitate foarte buna, creste intr-adevar gradul de partitionare, si deci capacitatea de diviziune a task-urilor intre membrii echipei. Totusi, legea lui Brooks ramane valabila, din 2 motive: (i) surplusul de comunicare este semnificativ indiferent de modularitate si (ii) oricat de modular ar fi proiectat un sistem, totusi nu se poate ca sistemul sa devina unul perfect partitionabil.

6. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *Testele whitebox nu pot garanta ca toate erorile dintr-o functie vor fi detectate.*

*dar daca sunt dublate de teste blackbox se poate garanta ca vor fi gasite toate erorile din respectiva functie.*

**Nota:** raspunsurile “adevarat/fals” neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este fundamental FALSA. Nici o forma de testare, si nici o combinatie de tehnici de testare nu pot vreodata garanta absenta bug-urilor. Asa cum spunea Dijkstra, testarea poate evidenta doar prezenta bug-urilor, dar nu poate niciodata garanta absenta lor. Desigur, utilizarea concertata a mai multor tehnici de testare contribuie la reducerea numarului de bug-uri, dar ceea ce face enuntul de mai sus, complet fals este cuvantul “garanta”.

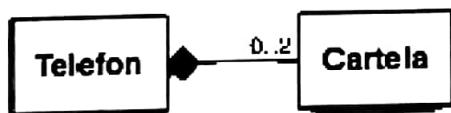
7. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *Testarea de regresie este bazata pe ideea ca atunci cand se opereaza schimbari intr-un modul, imediat dupa modificarile efectuate de testare trebuie concentrate exclusiv asupra modulului modificat.*

**Nota:** raspunsurile “adevarat/fals” neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este FALSA. Ideea de baza in testarea de regresie este reluarea unui subset de teste care au fost rulate anterior. Desigur, aceste teste se refera la modulul modificat, dar in nici un caz **exclusiv** asupra sa. In testarea de regresie, pe langa testele ce vizeaza modulul modificat se mai reiau doua clase de teste: (i) un set de teste reprezentative prin care se retesteaza principalele functionalitati ale sistemului; (ii) un set de teste ce vizeaza modulele direct conectate cu modul modificat, avand in vedere ca impactul schimbarii ar putea sa le fi influentat in primul rand pe acestea.

8. Precizati, si argumentati succint daca relatia descrisa de diagrama UML de mai jos modeleaza sau nu cazul unui telefon dual-sim (care poate tine simultan doua cartele) cu cartele care pot fi oricand reutilizate si in alt telefon.



*Raspuns:*

Relatia descrisa de diagrama UML NU modeleaza intrutotul ceea ce sustine enuntul de mai sus! Intr-adevar relatia de mai sus, indica faptul ca un obiect telefon are in compositia sa un numar de cartele ce poate varia intre niciuna si doua cartele... dar partea care nu este bine interpretata se refera la “reutilizarea oricand in alt telefon”. Relatia din figura este una de compositie, marcata prin rombul negru, si deci obiectele “Cartela” sunt construite specific pentru un anumit obiect “Telefon”, putand fi folosite doar de catre acel obiect.

9. Considerand urmatoarele doua procese de dezvoltare: *Waterfall* si respectiv *Extreme Programming*, precizati si argumentati succint pe care l-ati alege daca ati fi *pe rand* in urmatoarele cazuri:

*Cazul 1:* trebuie sa reimplementati de la zero, un sistem complex (cca. 1 milion de linii de cod) pentru gestiunea personalului si a salariilor, care sa inlocuiasca sistemul existent in prezent, fara nici o modificarile de cerinte.

*Cazul 2:* aveți de construit un sistem pentru plata taxelor si impozitelor pentru tara imaginara Ainamor cu o legislatie haotica si in permanenta modificarile.

**Nota:** cele 2 cazuri sunt complet independente.

*Raspuns:*

Pentru “Cazul 1” as alege procesul de dezvoltare Waterfall din urmatoarele motive: (i) e vorba de un sistem de mari dimensiuni, ce implica deci echipe de mari dimensiuni, ceea ce

face ca un proces mai formal sa fie dezirabil; in plus (ii) cerintele sistemului sunt f. bine cunoscute, si clar fixate, deci nu exista riscul de a trebui sa reluam toate fazele procesului (principalul dezavantaj la Waterfall) datorita unei schimbari de cerinte.

Pentru "Cazul 2" as alege in mod evident Extreme Programming datorita cerintelor in permanenta schimbare. Din acest motiv avem nevoie de un proces care sa se poata adapta bine, si mai ales rapid schimbarilor de cerinte, iar Waterfall nu corespunde acestor cerinte. Extreme Programming, are avantajul ca este un proces iterativ si incremental, si in plus are durata iteratiilor scurte.

10. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *Procesul de dezvoltare in cascada (waterfall) este recomandat in majoritatea proiectelor intrucat, datorita structurii sale rigide, poate constrange clientul sa descopere, inca de la inceput, toate posibilele cauze de schimbare din sistem.*

**Nota:** raspunsurile "adevarat/fals" neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este FALSA. Procesul de dezvoltare Waterfall este recomandat in foarte putine cazuri, tocmai datorita structurii sale rigide, nebazata pe iteratii si incremente, ce impiedica o adaptare rapida la schimbari de cerinte. Cu atat mai putin este recomandat procesul Waterfall atunci cand vine vorba de un sistem unde cerintele nu sunt bine intelese. Structura rigida a lui Waterfall NU poate sub nici o forma "constrange" (determina) clientul sa descopere de la inceput toate posibilele cauza de schimbare... Aceasta este o utopie. Chiar si in sisteme unde cerintele sunt clar specificate de la inceput, pot interveni in timp schimbari ce nu pot fi sub nici o forma prevazute. De aceea, trebuie favorizate procesele de dezvoltare (iterative si incrementale) care pot face fata schimbarilor neasteptate.

1. 50% din timpul si costurile totale pentru un proiect ar trebui alocat testarii, dar in multe cazuri in realitate, procentajul este mai mic.

Raspuns: Afirmatia este falsa. Conform regulilor lui Brooks, testarea ocupa intr-adevar 50% atat dpdv al timpului cat si al costului, iar in realitate, acest lucru este respectat. Prin testare se descopera eventuale erori in program, iar testarea insuficienta ar putea avea consecinte dezastruoase datorita anumitor erori nedepistate.

2. Forma reala a curbei ratei defectarii SW-ului este datorata faptului ca cerintele nu sunt intelese clar de la bun inceput, ceea ce duce la necesitatea de schimbari continue asupra sistemului.

Raspuns: Afirmatia este falsa. Schimbarile continue asupra sistemului constituie intr-adevar principalul motiv pentru care curba are acea forma, insa acestea pot aparea chiar daca cerintele au fost intelese perfect initial. De exemplu, s-ar putea ca pe parcurs una sau mai multe cerinte sa se schimbe, ceea ce evident inseamna ca va trebui ca sistemul sa fie modicat corespunzator.

3. Conceptul de time unboxing spune ca sarcinile alocate pt o anumita iteratie trebuie terminate in cadrul iteratiei in cauza

Raspuns: Afirmatia este falsa. In cazul in care sarcinile asignate unei anumite iteratii nu sunt finalizate la timp, ceea ce a ramas va fi inclus in iteratia urmatoare. Intervalul de timp pentru iteratii trebuie sa ramana intotdeauna fix, deoarece altfel echipa de programatori ar avea dificultati in a-si da seama de motivul pentru care nu a reusit sa finalizeze tot ce si-a propus in cadrul iteratiei in cauza.

4. In procesele de dezvoltare agile nu este necesar sa intelegem de la bun inceput cerintele sistemului.

Raspuns: Afirmatia este falsa. Chiar daca metodele agile se caracterizeaza prin adaptabilitatea rapida la modificari si prin faptul ca schimbarile sunt mereu bine-venite, acest lucru nu inseamna ca nu trebuie sa intelegem de la bun inceput ceea ce se cere. Daca pe parcurs programatorii si-ar da seama ca anumite cerinte au fost intelese gresit, acest lucru ar necesita modificarile care ar constitui o risipa de bani si de timp.

5. Daca avem la dispozitie diagrame UML foarte explicite pt un sistem, putem deduce diagramele de secventa pt acel sistem, adica toate interactiunile posibile dintre obiecte

Raspuns: Afirmatia este falsa. Diagramele UML de clasa descriu sistemul dpdv structural - prezinta clasele de obiecte impreuna cu atributiile si operatiile lor, precum si relatiile existente intre obiecte. Diagramele de secventa, pe de alta parte, descriu comportamentul sistemului. Aceste diagrame nu se pot deduce decat cu ajutorul codului sursa, iar acesta nu apare pe diagrama de clasa.

6. Un sistem nu poate fi actor pentru un alt sistem pentru ca acest lucru ar implica accesul la informatiile din sistemul mare, ceea ce incalca incapsularea.

Raspuns: Afirmatia este falsa. Actorul este o entitate care interactioneaza cu sistemul, iar aceasta entitate poate fi orice. Un om (actor) care doreste sa retraga o suma de bani

de la un bancomat (sistem) nu stie nimic despre detaliiile de implementare ale aparatului. Prin urmare, acest lucru se aplica si in cazul in care actorul nostru este chiar un alt sistem.

7. Pentru a evita proliferarea claselor, clasa Family trebuie implementata ca (in cursul 4, slide-ul 4, varianta din dreapta.)

Raspuns: Afirmatia este falsa. Modul in care trebuie implementate clasele depinde de ceea ce ne cere sistemul. De exemplu, daca o functionalitate ceruta de sistem ar fi "schimbaScutece()", atunci am opta pentru cea de-a doua varianta, deoarece aceasta operatie ar putea fi realizata de orice persoana din familie (mama, tata, frate, etc.). Daca insa sistemul are ere functionalitatea "naste()", cum mama este singura care poate face asta, varianta potrivita ar fi, evident, cea din stanga.

8. Folosirea constantelor globale incalca criteriul de modularitate al continuitatii.

Raspuns: Afirmatia este adevarata. Daca dorim sa dam variabilei globale alta valoare, acest lucru ar inseamna ca ar trebui sa modificam valoarea peste tot pe unde aceasta apare, ceea ce ar putea contraveni criteriului de continuitate in cazul in care variabila noastra ar aparea in multe locuri.

9. Stilul arhitectural pipes and filters este avantajos dpdv al timpului

Raspuns: Afirmația este fundamental falsă. Trecerea prin fiecare filtru necesită parsarea și analizarea întregului flux de date, ceea ce este consumator de timp.

10. Un program care nu are cicluri (for, while) are numarul ciclomatic=1

Raspuns: Afirmatia este falsa. Numarul ciclomatic ne arata numarul de "cai" de executie posibile pentru un anumit program. Ca exemplu vom lua o bucată de cod constând dintr-o secvență if-then-else. Acest program nu are cicluri for sau while, însă există două posibilități de executie: ramura de if, respectiv cea de else, adică numarul ciclomatic =2.

11. Principalul dezavantaj al testarii top down integration este crearea de drivere si stub-uri

Raspuns: Afirmatia este parcial adevarata. Top-down integration presupune testarea sistemului de sus in jos, incepand cu modulul principal, si coborand in ierarhie spre modulele de nivel inferior. Stuburile sunt niste inlocuitori pentru modulele care nu au fost inca testate sau implementate, iar crearea lor, necesara pentru acest tip de testare, este consumatoare de timp, deci constituie un dezavantaj. Driverele sunt niste programe care preiau datele de intrare pentru un test, le proceseaza, si tiparesc rezultatele testului, insa ele trebuie create indiferent de procesul de testare ales, deci nu as spune ca acestea constituie un dezavantaj specific testarii top-down integration.

FALS      Dunderde niet voorsoe die dender  
as tip brotters y

34. Daca dorim sa verificam cat mai timpuriu principalele puncte de control si decizie din sistem vom alege testarea de integrare de tip Bottom-Up.

Fals. Testarea de integrare Bottom-Up verifica timpuriu procesarea de date de nivel scazut. Pentru a verifica cat mai timpuriu principalele puncte de control si decizie din sistem vom alege testarea de integrare de tip Top-Down.

35. Valoarea complexitatii ciclomatice a unei functii nu este influentata de numarul de instructiuni de ciclare(for/while) intrucat singurele instructiuni care incrementeaza valoarea complexitatii ciclomatice sunt cele de divizie de tip if-else.

Fals. Valoarea complexitatii ciclomatice este influentata atat de deciziile if/else, cat si de numarul de instructiuni de ciclare (for/while), deoarece si aceste instructiuni contin cate o decenzie simpla.

36. Curba defectelor poate fi facuta sa tinda in timp spre zero daca cerintele sunt bine intelese de la inceput, si daca se aplica sistematic metode de testare eficiente.

Fals. Chiar daca sunt bine intelese cerintele de la inceput si se aplica metode de testare eficiente, in timp, din cauza schimbarilor repetate tot vor aparea erori in sistem, deci curba defectelor nu va tinde in timp spre 0.

37. Utilizarea eficienta a instrumentelor CASE dedicate programarii poate reduce semnificativ costurile totale ale proiectului avand in vedere ca intr-un proiect software o parte semnificativa a costurilor sunt legate de activitatea de codare.

Fals. Utilizarea CASE reduce costurile, dar ele se folosesc nu doar pentru partea de codare, ci si pentru alte etape ale procesului precum specificatii, design, testare, debugging. De- asemenea, o mare parte din costuri nu se duc pe partea de codare, ci pe partea de testare si planificare.

38. In procesul de dezvoltare Waterfall se face doar un singur tip de activitate la un moment dat, spre deosebire de cele iterative unde intr-o iteratie trebuie efectuate toate tipurile de activitate.

Adevarat. Procesul de dezvoltare Waterfall imparte proiectul in mai multe activitati ce sunt rezolvate pe rand. De exemplu, se realizeaza analiza si definirea cerintelor, dupa ce se incheie aceasta faza se realizeaza design-ul de sistem si software si asa mai departe.

39. In procesul Rational Unified Process (RUP), desi este permisa efectuarea mai multor tipuri de activitati in interiorul unei faze, se recomanda ca numarul de activitati desfasurate in paralele sa fie limitat.

Fals. Intr-adevar, in cazul procesului Rational Unified Process putem efectua mai multe tipuri de activitati simultan in interiorul unei faze, dar nu exista nicio limitare cu privire la numarul de procese desfasurate in paralel.

40. Use Case-urile de tip Fish Level se folosesc atunci cand vrem sa detaliem fiecare actiune principala din cadrul unui use-case de tip Sea Level.

Fals. Use Case-urile de tip Sea Level ne arata cum userul interactioneaza cu sistemul, interactiunile sunt majore, iar scopul este bine precizat. Use Case-urile de tip Fish Level sunt acele use case-uri care se dau factor comun din use case-urile Sea Level(cu <<include>>)

41. Stim ca am descoperit toate Use Case-urile dintr-un sistem atunci cand toate cerintele functionale sunt acoperite de use case-uri Sea Level, fiecare dintre acestea trebuind sa fie conectate cu cel putin un actor.

Adevarat. Use Case-urile de tip Sea Level prin definite reprezinta interacțiuni majore ale utilizatorilor sistemului cu un scop precis. Astfel daca toate cerintele functionale sunt acoperite de Use Case-uri de tip Sea Level am descoperit toate Use Case-urile din sistem. Si da, trebuie ca fiecare use case sea level sa fie legat la un user pentru ca altfel nu are sens, un use case presupune existenta cel putin a unui user care sa-l foloseasca.

42. In tehnica CRC clasele sunt identificate pornind de la substantivale din descrierea use case-urilor, iar responsabilitatea se identifica dintre verbele folosite in cadrul descrierii scenarior.

Adevarat. In tehnica CRC clasele sunt identificate pornind de la substantivale, intrucat programarea orientata pe obiecte cere ca si clasele sa poata defini entitati ale sistemului, deci ele nu pot fi decat substantivale, iar responsabilitatile claselor reprezinta functionalitatea lor redată prin metode, iar aceasta functionalitate este data de verbele din scenariul unei diagrame use case.

43. Relatia de compositie este un caz special de agregare (relatie parte-intreg) in care indicam ca distrugerea clasei "intreg", atrage dupa sine si distrugerea obiectelor "parte" continute de catre clasa "intreg".

Adevarat. Relatia de compositie este o forma de agregare in care componentele nu pot exista una fara cealalta.

44. Intre specificatorul de acces "protected" si criteriul de modularitate al protectiei exista o legatura stransa.

Fals. Specificatorul de acces "protected" face ca atributele si metodele unei clase sa fie vazute doar de clasele care mostenesc clasa respectiva. Criteriul de modularitate al protectiei ne spune ca daca apare o eroare de executie, aceasta se limiteaza la doar cateva module. Criteriul se poate asigura si cu specificatorul private, cu pachete etc.

45. In stilul arhitectural Repository diferentele componente care proceseaza date sunt total independente unele de altele, cu exceptia faptului ca folosesc acelasi model de date.

Adevarat. Stilul arhitectural Repository este un mod eficient de a distribui mari cantitati de date in care producatorii si consumatorii de date sunt independenti, dar marele compromis este faptul ca folosesc acelasi model de date.

46. Principalul motiv pentru care testarea de integrare de tip Big Bang nu este recomandata este acela ca spre deosebire de alte tehnici de testare de integrare aceasta identifica mai putine defecte.

Fals. Testarea de integrare de tip Big Bang combina oate componentelete in avans si testeaza intreg programul. In acest fel se creeaza un haos cu multe erori care la prima vedere nu se leaga. Corectarea acestora se realizeaza greu, dupa aceasta rezulta alte erori si astfel testarea pare sa intre intr-o bucla infinita.

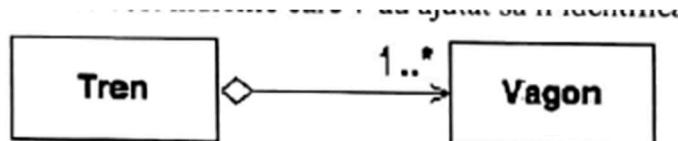
47. Testarea ciclurilor pentru o functie cu doua cicluri imbicate (nested) implica scrierea a doua cazuri de test; unul sa parcurga instructiunile din ciclul interior, si un al doilea care sa testeze instructiunile din ciclul exterior.

Adevarat. Testarea functiilor cu 2 cicluri imbicate incepe cu testarea buclei din interior care face ca testarea unei bucle simple pastrand iteratorul buclei exterioare la valoarea minima. Dupa aceasta se trece la testarea celei de a doua bucle in maniera testarii unei bucle simple.

48. Precizati ce stil arhitectural s-ar asocia cel mai bine urmatoarele imagini:

- O ceapa - Layered Architecture intrucat aceasta are o structura stratificata, iar in mijloc se afla lastarul care este cea mai importanta parte a acesteia
- O linie de productie - conducte si filtre. Conductele reprezinta benzile rulante care ajuta produsele sa se deplaseze prin fabrica, iar filtrele sunt reprezentate de diferite masinarii care transforma simplele materii prime in produsul finit, pas cu pas.
- Un mediu de dezvoltare integrat(IDE), gen Eclipse, Visual Studio, JBuilder sau un instrument CASE complex - Repository, deoarece e un mediu de dezvoltare care are mai multe unelte: compilator, debugger, etc. care sunt independente intre ele si care creeaza si folosesc acelasi tip de date.

49. Cum se reprezinta intr-o diagrama UML de clasa relatia intre o clasa "vagon" si o clasa "tren" considerand ca un tren este compus din unul sau mai multe vagoane si ca un obiect "vagon" poate fi refolosit in relatia cu diferite obiecte "tren". Precizati cum se numeste acest tip de relatie descris si care au fost indiciile care v-au ajutat sa il identificati.



Relatia descrisa se numeste agregare, si se reprezinta in UML ca in figura de mai jos: Relatia intre "Vagon" si "tren" este in mod clar o relatia de tip "HAS-A" (intre-parte), adica un obiect "tren" este compus din obiecte "vagon". Astfel, am oscilat intre o relatia de agregare si una de compositie. Indiciul din enunt care a facut diferența este acela ca obiectul "Vagon" poate fi refolosit in relatia cu diferite obiecte "tren", ceea ce inseamna ca viata obiectelor "vagon" este distincta de cea a obiectelor "vagon". Astfel, relatia este una de agregare.

50. Daca intr-un sistem ar trebui sa favorizati performanta de timp, ati opta pentru stilul arhitectural Layered (arhitectura stratificata)? Dar daca ar trebui sa favorizati securitatea?

Daca trebuie favorizata performanta de timp, stilul arhitectural Layered, NU este recomandat. Motivul este acela ca apelarea unui serviciu la nivelul cel mai din exterior implica cel mai adesea o dubla parcurgere a tuturor nivelurilor/straturilor, adica atat pentru transimterea serviciului, cat si pentru receptarea rezultatului/ rezultatelor.

Daca insa trebuie favorizata securitatea, stilul Layered, este extrem de indicat. Motivul este acela ca fiecare layer poate oferi un nivel distinct de securitate, ce poate fi verificat atunci cand acel nivel este accesat. In plus, faptul ca fiecare nivel comunica doar cu nivelul imediat inferior respectiv superior face ca incapsularea datelor, si deci securitatea sa fie mai buna.

51. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint raspunsul: La testare blackbox avem nevoie si de cod pentru a verifica daca datele de test acopera toate calele din program ?

FALS. Motivul este ca acela ca testarea blackbox, prin insasi definitia sa, nu implica cunoasterea codului. In acest black box, cazurile de test se scriu strict pe baza "contractului" modului testat, adica a datelor de intrare, respectiv a celor de iesire.

52. Care este diferența intre urmatoarele tipuri de relatii intre clase: asociere, agregare, componitie ?

Asocierea reprezinta forma cea mai slaba in care putem exprima relatia dintre doua clase. Atunci cand spunem ca o relatie este de "asociere" tot ce stim este ca intre cele doua clase exista o relatie.

Atunci cand afirmam ca o relatie intre doua clase este una de agregare sau componitie, inseamna ca din descrierea cerintelor putem spune ca intre cele doua clase implicate exista o relatie de tip "HAS-A", o relatie de tip "parte-intreg". Mai departe, distinctia intre agregare si componitie este data de masura in care obiectele "parte" pot fi reutilizate de diferite obiecte "intreg": daca obiectele parte sunt create si folosite exclusiv de catre un singur obiect, atunci relatia este una de componitie; in caz contrar, daca obiectele "parte" pot fi folosite shared de catre mai multe obiecte "intreg" atunci relatia este una de agregare. Pe scurt: componitia este o forma de relatie mai stransa decat agregarea.

53. Enuntati legea lui Brooks, referitoare la extinderea echipelor de dezvoltatori in timpul proiectului. De asemenea precizati, argumentand succint, valoarea de adevar a urmatoarei afirmatii: Intr-un sistem cu o modularitate foarte buna legea lui Brooks nu se aplica pentru ca activitatea de construire a sistemului este o activitate perfect partitionabila.

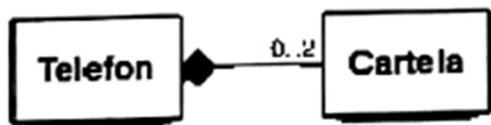
Legea lui Brooks afirma ca adaugarea de programatori la un proiect aflat in intarziere, va face proiectul sa intarzie si mai mult. Motivul este acela, ca in realizarea unui proiect software nu este o activitate perfect partitionabila, ci dimpotriva una ce implica foarte multe interactiuni. Iar adaugarea unor noi programatori ar implica o repartitionare a taskurilor din proiect. In plus fiecare om adus in plus, aduce dupa sine relatii de comunicare mai complexe.

In sisteme cu modularitate foarte buna, creste intr-adefar gradul de partitionare, si deci capacitatea de diviziune a task-urilor intre membrii echipei. Totusi, legea lui Brooks ramane valabila, din 2 motive: 1. Surplusul de comunicare este semnificativ indiferent de modularitate si 2. Oricat de modular ar fi proiectat un sistem, totusi nu se poate ca sistemul sa devina unul perfect partitionabil.

54. Testele whitebox nu pot garanta ca toate erorile dintr-o functie vor fi detectate, dar daca sunt dublate de teste blackbox se poate garanta ca vor fi gasite toate erorile din respectiva functie.

FALS. Ideea de baza in testarea de regresie este reluarea unui subset de teste care au fost rulate anterior. Desigur, aceste teste se refereaza la modulul modificat, dar nici un caz exclusiv asupra sa. In testarea de regresie, pe langa testele ce vizeaza modulul modificat se mai reiau doua clase de teste: 1. Un set de teste reprezentative prin care se retesteaza principalele functionalitati ale sistemului 2. Un set de teste ce vizeaza modulele direct conectate cu modul modificat, avand in vedere ca impactul schimbarii ar putea sa le fie influentat in primul rand pe acestea.

55. Modeleaza sau nu cazul unui telefon dual-sim cu cartele care pot fi reutilizate oricand in alt telefon



FALS. Diagrama UML nu modeleaza intrutotul ceea ce sustine enuntul de mai sus. Intr-adefar relatia de mai sus, indica faptul ca un obiect telefon are in componitia sa un numar de cartele ce poate varia intre niciuna si doua cartele, dar partea care nu este bine interpretata se refera la reutilizare. Relatia din figura este una de compositie, marcată prin rombul negru, deci obiectele "Cartela" sunt construite specific pentru un anumit obiect "Telefon", putand fi folosite doar de catre acel obiect.

56. Waterfall si Extreme Programming, precizati si argumentati succint pe care l-ati alege daca ati fi pe rand in urmatoarele cazuri:

- I. Trebuie sa reimplementati de la zero, un sistem complex pentru gestiunea personalului si a salariilor, care sa inlocuiasca sistemul existent in prezent, daca nici o modificare de cerinte.
- II. Aveti de construit un sistem pentru plata taxelor si impozitelor pentru tara imaginara Ainand cu o legislatie haotica si in permanenta modificare

Pentru I. se alege procesul de dezvoltare Waterfall, deoarece e vorba de un sistem de mari dimensiuni, ce implica deci echipe de mari dimensiuni, ceea ce face ca un proces mai formal sa fie dezirabil; in plus cerintele sunt foarte bine cunoscute deci nu exista riscul sa relua toate fazele procesului.

Pentru II. as alege in mod evident Extreme Programming datorita cerintelor in permanenta schimbare. Din acest motiv avem nevoie de un proces care sa se poata adapta bine, si mai ales rapid schimbarilor de cerinte, iar Waterfall nu corespunde acestor cerinte.

Extreme Programming, are avantajul ca este un proces iterativ si incremental, si in plus are duarata iteratiilor scurte.

57. Procesul de dezvoltare waterfall este recomandat in majoritatea proiectelor intrucat, datorita structurii sale rigide, poate constrange clientul sa descopere, inca de la inceput, toate posibilele cauze de schimbare din sistem.

FALS. Procesul de dezvoltare Waterfall este recomandat in foarte putine cazuri, tocmai datorita structurii sale rigide, nebazata pe iteratii si incremente, ce impiedica o adaptare rapida la schimbari de cerinte. Cu atat mai putin este recomandat procesul Waterfall atunci cand vine vorba de un sistem unde cerintele nu sunt bine intelese.

58. 50% din timpul si costurile totale pentru un proiect ar trebui alocat testarii, dar in multe cazuri in realitate, procentajul este mai mic.

FALS. Conform regulilor lui Brooks, testarea ocupa intr-adevar 50% atat dpdv al timpului cat si al costului, iar in realitate, acest lucru este respectat. Prin testare se descopera eventuale erori in program, iar testarea insuficienta ar putea avea consecinte dezastruoase datorita anumitor erori nedepistate.

59. Forma reala a curbei ratei defectarii SW-ului este datorata faptului ca cerintele nu sunt intelese clar de la bun inceput, ceea ce duce la necesitatea de schimbari continue asupra sistemului.

FALS. Schimbarile continue asupra sistemului constituie intr-adevar principalul motiv pentru care curba are acea forma, insa acestea pot aparea chiar daca cerintele au fost intelese perfect initial. De exemplu, s-ar putea ca pe parcurs una sau mai multe cerinte sa se schimbe, ceea ce evident inseamna ca va trebui ca sistemul sa fie modificat corespunzator.

60. Conceptul de time unboxing spune ca sarcinile alocate pt o anumita iteratie trebuie terminate in cadrul iteratiei in cauza

FALS. In cazul in care sarcinile asignate unei anumite iteratii nu sunt finalizate la timp, ceea ce a rarnas va fi inclus in iteratia urmatoare, Intervalul de timp pentru iteratii trebuie sa ramana intotdeauna fix, deoarece altfel echipa de programatori ar avea dificultati in a si da seama de motivul pentru care nu a reusit sa finalizeze tot ce si-a propus in cadrul iteratiei in cauza.

61. In procesele de dezvoltare agile nu este necesar sa intelegem de la bun inceput cerintele sistemului

FALS. Chiar daca metodele agile se caracterizeaza prin adaptabilitatea rapida la modificari si prin faptul ca schimbarile sunt mereu bine-venite, acest lucru nu inseamna ca nu trebuie sa intelegem de la bun inceput ceea ce se cere. Daca pe parcurs programatori si-ar da seama ca anumite cerinte au fost intelese gresit, acest lucru ar necesita modificari care ar constitui o risipa de bani si de timp.

62. Daca avem la dispozitie diagrame UML foarte explicite pt un sistem, putem deduce diagramele de secventa pt acel sistem, adica toate interactiunile posibile dintre obiecte

FALS. Diagramele UML de clasa descriu sistemul dpdv structural - prezinta clasele de obiecte existente intre obiecte. Diagramele de secventa, pe de alta parte, descriu comportamentul sistemului. impreuna cu atributele si operatiile lor, precum si relatiile Aceste diagrame nu se pot deduce decat cu ajutorul codului sursa, iar acesta nu apare pe diagrama de clasa

63. Un sistem nu poate fi actor pentru un alt sistem pentru ca acest lucru ar implica accesul la informatiile din sistemul mare, ceea ce incalca incapsularea.

FALS. Actorul este o entitate care interactioneaza cu sistemul, iar aceasta entitate poate fi orice, Un om (actor) care doreste sa retraga o suma de bani de la un bancomat (sistem) nu stie nimic despre detaliiile de implementare ale aparaturii. Prin urmare acest lucru se aplica si in cazul in care actorul nostru este chiar un alt sistem.

64. Pentru a evita proliferarea claselor, clasa Family trebuie implementata ca (in cursul 4, slide-ul 4, varianta din dreapta.)

FALS. Modul in care trebuie implementate clasele depinde de ceea ce ne cere sistemul. De exemplu, daca o functionalitate ceruta de sistem ar fi "schimbaScutece". atunci am opta pentru cea de-a doua varianta, deoarece aceasta operatie ar putea fi realizata de orice persoana din familie (mama, tata, frate, etc.). Daca insa sistemul ar cere functionalitatea "naste()", cum mama este singura care poate face asta varianta potrivita ar fi, evident, cea din stanga.

65. Folosirea constantelor globale incalca criteriul de modularitate al continuitatii

ADEVARAT. Daca dorim sa dam variabilei globale alta valoare, acest lucru ar inseamna ca ar trebui sa modificam valoarea peste tot pe unde aceasta apare, ceea ce ar putea contraveni criteriului de continuitate in cazul in care variabila noastra ar aparea in multe locuri

66. Stilul arhitectural pipes and filters este avantajos dpdv al timpului

FALS, Trecerea prin fiecare filtru necesita parsarea si analizarea intregului flux de date, ceea ce este consumator de timp

67. Un program care nu are cicluri (for, while) are numarul ciclomatic 1

FALS. Numarul ciclomatic ne arata numarul de "cai" de executie posibile pentru un anumit program. Ca exemplu vom lua o bucată de cod constant dintr-o secvență if- then-else. Acest program nu are cicluri for sau while, insă există două posibilități de execuție: ramura de if, respectiv cea de else, adică numarul ciclomatic =2.

68. Principalul dezavantaj al testarii top down integration este crearea de drivere și stub-uri

FALS. Driverele și stub-urile se folosesc la testarea de tip Bottom-Up.

**1.Criză software din anii 60 a fost cauzată de hardul insuficient de puternic fata de complexitatea excesiva a softwareului existent la acel moment.**

Afirmatia este evident falsa. Criza software din anii 60 a fost cauzata de cresterea rapida a puterii computerelor( de unde deducem ca hardul nu era insuficient) si a complexitatii problemelor ce trebuiau rezolvate. Masinariile au devenit cu cateva ordine de marime mai puternice.(au devenit mult mai puternice decat erau oamenii capabili sa le programeze).

**2.Cunoasterea perfectă a limbajului de programare în care se va implementa un proiect de mari dimensiuni e o condiție necesară și suficientă pentru terminarea cu succes a proiectului.**

In mod evident, este necesara cunoasterea limbajului de implementare pentru a termina cu succes un proiect. Insa afirmatia este falsa, deoarece indiferent cat de bine stiu ar fi limbajul de programare, acesta nu asigura terminarea proiectului cu succes, incat si alti factori precum: intelegera cerintelor, realizarea designului si testarea corespunzatoare sunt extrem de importanti. Prin cunoasterea limbajului de programare la perfectie se salveaza timp, insa doar 1/6 din timpul necesar realizarii unui proiect ii este alocat programarii.

**R 3.Intr-un sistem cu o modularitate foarte buna legea lui Brooks nu se aplica deoarece construirea sistemului e o activitate perfect partitionabila.**

Aceasta afirmatie este fundamental falsa. Legea lui Brooks ne spune ca nu e eficient sa adaugam mai multi oameni la un program aflat in intarziere, deoarece acest fapt il va face sa intarzie si mai mult. Acest lucru se datoreaza faptului ca construirea sistemului NU este o activitate perfect partitionabila, incat este absolut necesara comunicarea intre membrii echipei. Evident, in cazul unui sistem cu o modularitate foarte buna, taskurile sunt mai usor partitionabile. Totusi, chiar si in acest caz:

- 1) indiferent de cat de buna este modularitatea, exista un grad mare de comunicare necesar
- 2) indiferent cat de buna e modularitatea, construirea sistemului nu e o activitate perfect partitionabila

**4.Procesul de dezvoltare in cascada e recomandat in majoritatea proiectului, deoarece datorita structurii sale rigide poate constrange clientul sa descopere inca de la inceput toate posibilele cauze de schimbare din sistem.**

Afirmatia este falsa. Procesul de dezvoltare in cascada nu este recomandat in majoritatea proiectelor ,ci doar in cazul proiectelor care se desfasoara in mai multe locatii. Structura rigida nu poate sa constranga clientul sa descopere inca de la inceput toate posibilele cauze de schimbare din sistem, fiind indicat doar cand:

- a)cerintele sunt extrem de clare de la bun inceput
- b)schimbarile cerintelor sunt extrem de limitate

Desi se poate anticipa costul si eventualele erori se elimina in faza initiala, dureaza mult pana clientul primeste versiunea finala a proiectului.

**5. Procesele de dezvoltare “agile” se numesc astăzi pentru că atunci când le folosim, dezvoltarea întregului sistem durează garantat mai puțin decât dacă folosim alte procese.**

Afirmatia de mai sus este falsa. Desi agile este un proces cu o abordare iterativa, care produce satisfactia clientilor livrand codul frecvent(saptamanal) este gresit sa afirmam ca numele acestuia este datorita faptului ca procesul dureaza mai putin ca alte procese. Nu se poate stii cu certitudine daca acest proces dureaza garantat mai putin, incat pricepiile agile nu doresc sa obtina acest lucru,

ci mai de graba: interacțiunea f2f cu clientii, primirea cu bratele deschise a schimbarilor tarzii, atenția continuă acordată perfectiunii tehnice și a designului, etc.

**6.In procesele de dezvoltare iterative se stabileste pentru fiecare iteratie un set de funcionalitati care trebuie adaugate iar daca se constata ca timpul prevazut pentru iteratie nu este suficient pentru implementarea intregului set de funcionalitati, se prelungeste corespunzator durata iteratiei si se creste preventiv si durata urmatoarei iteratii.**

Afirmatia precedenta este falsa. Este adevarat ca in cazul proceselor iterative pentru fiecare iteratie se stabileste un set de functionalitati ce trebuesc adaugate, insa NICIODATA durata iteratiei nu se maresteste, chiar daca timpul prevazut a fost insuficient. Daca se constata ca timpul este insuficient se poate prelungi durata catorva functionalitati din iteratie.

**6.Ar trebui modelate in sistem doar clasele care transmit mesaje, nu cele care primesc mesaje**

Afirmatia de mai sus este falsa, incat conform problemei proliferarii claselor, clasele sunt aceleia care primesc mesaje, nu aceleia care primesc mesaje.

**7.Relatia de mai jos se interpreteaza astfel "clasa Car contine clasa Engine", mai precis cand se distruse clasa car se distrug si clasa engine.**

Fals. Intr-adevar in imagine reprezentata relatia de compositie, relatie de tip "parte-intreg", conform careia obiectele parte sunt create si folosite de un singur un singur obiect de tip intreg. Afirmatia este totusi falsa, deoarece relatia de compositie nu se stabileste intre clase, ci intre obiectele claselor. Nu clasele sunt cele distruse, ci obiectele claselor.

**8.Scenariile aferente use caseurilor si diagramele de secventa UML sunt doua modalitati alternative de modelare a interacțiunii intre clase. Cu alte cuvinte, daca dorim sa observam cum colaboreaza clasele, putem folosie fie usecase uri, fie diagrame de secventa.**

Afirmatia precedenta este falsa. Diagramale use case suprind interacțiunile dintre actori si sistem, insa pe baza lor nu se poate observa cum colaboreaza clasele. Diagramale UML, in schimb descriu clasele de obiecte: inclusiv atributiile si operatiile lor, precum si relatiile dintre ele, fiind potrivite daca dorim sa observam cum colaboreaza acestea.

**9.Precizati si argumentati succint daca relatia descrisa(telefon dual sim) cartelele pot fi oricand reutilizate si in alt telefon.**

Relatia descrisa de diagrama UML NU modeleaza in totalitatea ceea ce sustine enuntul. In mod cert, realtia din figura, este o relatie de compositie, si indica faptul ca un obiect telefon are in compositie sa un numar de cartele ce poate varia intre niciuna si doua. Partea din enunt care ma face sa remarc ca este fals e aceea ca "cartele pot fi oricand reutilizate si in alt telefon", complet fals in cazul relatiei de compositie, conform careia obiectele parte sunt create si folosite exclusiv de catre un singur obiect intreg. Astfel, obiectele "cartela" sunt construite pentru un singur obiect "telefon", putand fi folosite doar de catre acele obiecte.

**10. Un actor nu poate introduce informatii in sistemul modelat ci poate doar extrage informatii din acesta.**

Afirmatia este falsa. Actorul specifica rolul jucat de user in cadrul interacțiunii cu sistemul. Aceasta poate atat oferi cat si primii informatii din acestea, cat timp este exterior sistemului.

**11. Use caseul “Retrage bani de la ATM” e intr o relatie de tip “include” cu use case ul “Retrage bani de la ATM ramas fara cash” pentru ca primul e mai cuprinzator.**

Afirmatia este in mod evident falsa. Relatia de tip “<<include>>” este intradevar folosita pentru a factoriza functionalitatile comune ale un use case, dar este evident din enunt ca este vorba de o relatie de tip “<<extends>>” prin intermediul careia marcam o situatie exceptionala, un caz rar intalnit cum este cazul in care bancomatul ramane fara bani.

**12. In tehnica FAST de analiza a cerintelor, fiecare participant trebuie sa construiasca cate 4 liste independent de ceilalti si fara presiunea ca listele sa fie complete.**

Afirmatia de mai sus este adevarata. In cadrul tehnicii FAST dupa ce s-au facut 1-2 pagini cu cerinte, fiecare participant trebuie sa vina cu 4 liste(care contin: obiect, servicii, constrangeri, criterii de performanta), fara presiunea ca listele sa fie complete, incat se doreste doar observarea a cat mai multe puncte de vedere, motiv pentru care este important ca listele sa fie facute in mod independent. In cadrul tehnicii FAST nimic nu se “arunca”, ci din toate listele se face in final una singura eliminand evident redundantele.

**13. Testarea ocupa 25% din costul total al dezvoltarii unui sistem soft**

Afirmatia este falsa. Conform regulilor lui Brooks de impartire a programului, testarii ii este asociat jumatate din costul total al dezvoltarii(1/4 se duce pe testarea de componente si ¼ pe testarea sistemului).

**14. Pentru a evita proliferarea claselor, clasa Family tr implementata ca in C4 sl4, var dreapta**

Afirmaria este falsa. Comportamentul este cel care decide: daca este diferit avem clase, iar daca nu doar roluri ale aceleiasi clase. Evident, depinde de DOMENIUL MODELAT DE APPLICATIE(de ceea ce cere sistemul). De exemplu, daca o functionalitate ceruta de sistem ar fi “schimbaScutece()” atunci am opta pentru a doua varianta, deoarece aceasta operatie poate fi realizata de orice membru al familiei. In schimb, daca functionalitatea ceruta ar fi ”naste()” este evident ca doar mama ar putea face asta, varianta potrivita fiind cea din stanga.

**15. Folosirea constantei globale incalca criteriul de modularitate al continuitatii.**

Conform criteriului de modularitate al continuitatii schimbarile mici au ca efect producerea de schimbare in doar cateva module(nu afecteaza intreaga arhitectura). Astfel, este evident ca constantele globale incalca acest criteriu, incat modificarea lor intr-un loc determina modificarea lor in toate functiile in care apar. Acest fapt e in antiteza cu criteriul de modularitate al continuitatii, deci este FALS.

**? 16. 50% din timpul si costurile totale pentru un proiect ar trebui alocat testarii, dar in multe cazuri in realitate, procentajul este mai mic.**

Intr-adevar, conform lui Brooks modul in care ar trebui construit programul este: 1/6 codare, 1/3 planificare, ½ pentru testare, deci testarii I-ar trebui alocare jumata din timpul si costurile totale pentru un proiect. Afirmatia este insa falsa, incat, in realitate, procentajul nu este mai mic, deoarece oamenii sunt constienti de efectele dezastroase produse de testarea insuficienta.

**? 17. Forma reala a curbei ratei defectarii sw este datorata faptului ca cerintele nu sunt intelese clar de la bun inceput, ceea ce duce la necesitatea de schimbari continue asupra sistemului**

Forma reala a curbei defectarii sw este datorata schimbarilor permanente ce se produc in procesul de dezvoltare software( fie ca ne referim la noi cerinte, ori la procesul de mentenanta). Afirmatia este totusi falsa, deoarece indiferent cat de bine sunt intelese cerintele la inceput, schimbarile sunt inevitabile.

**18. Conceptul de time boxing spune ca sarcinile alocate pentru o anumita iteratie trebuie terminate in cadrul iterataiei in cauza.**

Conceptul de time boxing spune ca fiecarei iteratii ii este atribuita o perioada fixa de timp( numita time box), timp in care pot fi indeplinite anumite sarcini. In schimb, nu spune nicaieri ca o anumita iteratie trebuie terminata strict in cadrul iteratiei respective. In cazul in care timpul este insuficient, timpul alocat iteratiei NU se maresteste, insa se pot asigna anumite sarcini urmatoarei iteratii. Astfel, afirmatia este FALSA.

**19. In procesele de dezvoltare agile nu este necesar sa intelegem de la bun inceput cerintele sistemului.**

Afirmatia este cu certitudine falsa. Intr-adevar, procesele de dezvoltare agile se remarcă prin flexibilitate, incat chiar si schimbarile tarzii sunt bine-venite, insa este gresit sa credem ca asta inseamna ca cerintele nu trebuie clar intelese de la bun inceput. In orice proces de dezvoltare este esential ca cerintele sa fie intelese, insa comparativ cu alte procese(cum ar fi Waterfall), in cadrul carora cerintele trebuie sa aiba un numar limitat de schimbari si sa fie perfect clare de la bun inceput, metodologia agile se bazeaza mai mult pe comunicare f2f cu oamenii si este deschisa spre schimbari.

**20. Daca avem la dispozitie diagrame UML foarte explicite pentru un sistem, putem deduce diagramele de secventa pentru acel sistem, adica toate interactiunile posibile dintre obiecte.**

Diagramele UML descriu clasele din punct de vedere structural, impreuna cu atributele si operatiile lor, prezantand de asemenea relatiile ce exista intre acestea, fara a prezenta vreun amanunt legat de implementarea acestora. In schimb, diagramele de secventa descriu sistemul din punct de vedere al interactiunilor, fiind extrem de folositoare pentru a vedea daca un obiect lipseste( desi consuma mult timp, merita investit) iar construitea lor necesita accesul la cod. Evident, afirmatia este falsa deoarece indiferent cat de explicite ar fi diagramele UML pentru un sistem, acestea nu ajuta la construirea diagramelor de secventa(si reciproc).

**21. Un sistem nu poate fi actor pentru un alt sistem pentru ca acest lucru ar implica accesul la informatiile din sistemul mare, ceea ce incalca incapsularea.**

Actorul este cel care specifica rolul jucat de un user in interactiunea cu sistemul. Este important de stiut ca actorul interactioneaza cu sistemul din exterior, fara ca acesta sa aiba acces la informatiile din sistem. Astfel, in mod evident, afirmatia este falsa incat un sistem poate fi un actor pentru un alt sistem, iar acest fapt nu incalca incapsularea.(De ex daca un om (actor) vrea sa retraga o suma de bani de la bancomat(sistem), acesta nu trebuie sa stie cum a fost implementat bancomatul).

**22. Relatia din figura(compozitie, rombul colorat) modeleaza cazul unui telefon dual-sim, cand cartela poate fi oricat reutilizata si in alt telefon.**

Relatia din figura alaturata este evident o relatii de tip "parte-intreg", iar rombul colorat ne indica faptul ca este o relatii de compozitie. Intr-adevar, relatia modeleaza cazul unui telefon dual-sim, incat din figura deducem ca putem avea de la 0 la 2 cartele. Partea din enunt care indica ca este FALS e aceea ca o cartela poate fi reutilizata si in alt telefon. Daca relatia ar fi fost de agregare, atunci enuntul ar fi fost adevarat, dar cum in cazul relatii de compozitie ,obiectului intreg ii este asociat un singur obiect de tip parte(iar odata cu distrugerea intregului este distrus si obiectul/obiectele parte), este fals.

COMPOZITIE → obiectele parte sunt create si folosite exclusiv de catre un singur obiect intreg  
AGREGARE → obiectele parte pot fi reutilizate de diferite obiecte intreg

**23. Intr-o diagrama de secventa obiectele care sunt instante ale aceleiasi clase trebuie sa fie comasate intr-un singur dreptunghi, adica nu este admis sau recomandat sa reprezentam fiecare obiect ca un dreptunghi separat pentru ca astfel s-ar incarca prea mult diagrama**

Intr-o diagrama de secventa, obiectele care sunt instante ale aceleiasi clase NU trebuie sa fie comasate intr-un singur dreptunghi, in cat fiecare obiect are propriul sau rol. Daca ele ar fi comasate intr-un singur dreptunghi nu am mai veda interactiunile asa cum ar fi firesc, deci afirmatia este fundamental falsa.

**24. Scenariile aferente use caseurilor si diagramele de secventa UML sunt doua modalitati de modelare a interactiunilor intre clase.**

Scenariile use case sunt folosite pentru a arata interactiunea dintre actori si sistem, pe cand diagramele UML de secventa prezinta comportamentul intre 2 sau > entitati in termeni de interactiune si ordinea in care sunt mesajele schimbate. Astfel, enuntul este fals, in cat use caseurile nu pot fi folosite pentru a arata interactiunea claselor.

**25. In modelarea cerintelor se poate aplica urmatoarea regula: aproape orice substantiv intalnit intr-un document de cerinte e un actor si aproape orice verb e un USE CASE.**

Afirmatia este cu certitudine falsa. Actorul este acela care arata rolul jucat de un user in interactiunea cu sistemul, insa nu orice substantiv intalnit in documentul de cerinte indeplineste un rol. De asemenea, use caseurile, actiunile ce definesc interactiunea dintre sistem si actori nu sunt reprezentate de aproximativ orice verbe. Doar actiunile care au un anumit scop trebuie extrase, nu si pasii intermediari. De regula use caseul este descris printr-un substantiv si un verb.

**26. Un actor nu poate introduce informatii intr-un sistem modelat ci poate fi doar beneficiarul responsabil de sistem(nu poate extrage informatii din sistem)**

Actorul, cel care specifica rolul jucat de un user in interactiunea cu sistemul, poate atat sa trimita cat si sa primeasca informatii de la sistem, cat timp interactioneaza cu acesta din exterior. Astfel, afirmatia este falsa.

**27. In procesul de dezvoltare Scrum Burndown Chart ne arata evolutia efortului de-a lungul intregii istorii a proiectului, mai exact ne ofera informatii despre sprinturile deja incheiate precum si numarul de taskuri finalizate pana in prezent in intregul proiect.**

Afirmatia este falsa. In procesul de dezvoltare Scrum, Burndown Chart-ul este o reprezentare grafica a muncii care mai trebuie finalizata si timpul ramas pentru terminarea ei, fiind utilizat pentru a anticipa cand se va finaliza proiectul, in niciun caz pentru masurarea evolutiei efortului de-a lungul intregului proiect.

**28. O problema importanta a proceselor de dezvoltare iterative e aceea ca adesea trebuie sa se modifice codul care a fost scris intr-o iteratie anterioara si uneori anumite portiuni de cod trebuie sa se stearga, ceea ce reprezinta o pierdere/risipa.**

Afirmatia este adevarata deoarece intr-adevar in cadrul proceselor de dezvoltare iterative, modificarea codului scris intr-o iteratie anterioara ori chiar sergerea anumitor portiuni de cod reprezinta o risipa de timp, deci implicit de cost. In schimb, pentru a reduce aceasta pierdere, codul ar trebui rescris, in cat se pierde mult mai mult timp in incercarea de a "repara" un cod prost decat se pierde pentru rescriere.

**29. In procesul de dezvoltare Scrum, daca functionalitatile prevazute pentru un Sprint nu pot fi terminate intr-un timp prevazut, Sprintul poate fi prelungit, dar nu mai mult de cateva zile.**

In cadrul procesului de dezvoltare Scrum, se aloca o perioada fixa de timp (numita Sprint) in care trebuie finalizate anumite functionalitati. Chiar daca aceste nu sunt finalizate la timp, Sprintul NU se prelungeste, in cat acest fapt poate duce la cresterea complexitatii, a riscului si evident, a costului.

Astfel, afirmatia este falsa, iar sprintul nu poate fi prelungit nici macar cateva zile, insa pot fi eliminate din functionalitati.

**30. Legile evolutiei software ale lui Lehman nu se aplica la sisteme ale caror cerinte sunt perfect intelese de la bun inceput, pentru ca astfel de sisteme nu au niciun motiv sa evolueze.**

Afirmatia este in totalitate falsa, incat nu exista niciun sistem care sa nu trebuiasca sa evolueze (exceptand poate sistemele care nu sunt folosite niciodata). De asemenea, faptul ca cerintele sunt perfect intelese de la bun inceput nu indica faptul ca sistemul nu ar trebui sa evolueze, ci poate ajuta la finalizarea mai alerta a sistemului Software. Legile lui Lehman se aplică tuturor sistemelor, incat toate sistemele trebuie sa evolueze, caci in caz contrar devin din ce in ce mai nesatisfacatoare pentru clienti.

**31. Daca avem diagrame de clasa UML explicite pentru un sistem, atunci putem deduce diagramele de secventa pentru el, adica toate interactiunile posibile din sistem.**

Diagramele UML descriu clasele din punct de vedere structural, incluzand operatiile si atributele lor, reprezentand de asemenea si relatiile dintre ele. Diagramele de secventa, pe de alta parte, descriu comportamentul din punct de vedere al interactiunilor si ordinea in care mesajele sunt schimbate. Astfel, este evident gresit sa credem ca diagramele UML, indiferent de cat de explicite ar fi ele, ajuta la deducerea diagramelor de secventa(cu atat mai putin cu cat pentru a realiza o diagrama de secventa trebuie sa avem acces la cod, iar diagramele UML nu ofera acest lucru).

**32. Stilul arhitectural Pipes and Filters este avantajos din punct de vedere al timpului.**

Afirmatia este in mod evident falsa. Filtrele, independente unele fata de celalte, se ocupă cu prelucrarea de date, iar conductele sunt cele care transporta datele de la un filtru la altul. Este insa cunoscut faptul ca pipes and Filters nu este un stil arhitectural avantajos din punct de vedere al timpului, incat au nevoie de un format comun de date, iar fiecare filtru trebuie sa isi faca "parse" si "unparse" datelor, ceea ce duce la OVERHEAD, deci, implicit, la pierderea timpului.

**33. Diagramele UML de secventa sunt folosite ca punct de plecare pentru sesiunile de CRC cards, in urma carora se construiește diagramele UML de clase.**

Afirmatia este in mod cert falsa. CRC cards sunt folosite pentru identificarea claselor ce indica responsabilitatile si colaboratorii lor, pe cand diagramele UML de secventa descriu comportamentul din punct de vedere al interactiunilor si surprind ordinea in care se activeaza mesajele. In mod evident, diagramele de secventa nu pot fi folosite ca punct de plecare pentru realizarea CRC cards, punctul de plecare al acestora fiind de fapt use case-urile. Este insa adevarat ca pe baza crc cards se realizeaza diagramele de clasa.

**34. Un sistem care respecta criteriul de modularitate al compozibilitatii il va respecta aproape sigur si pe cel al decompozibilitatii incat primul se refera la posibilitatea de a crea un sistem din module.**

Criteriul de modularitate al decompozibilitatii presupune descompunerea problemelor in probleme mai mici care pot fi rezolvate separat, pe cand criteriul de modularitate al compozibilitatii presupune compunerea libera a modulelor pentru a produce sisteme. In mod cert, afirmatia este falsa, incat cele doua criterii sunt independente si complet opuse. De exemplu Ikea, ne exemplifica un caz in care este respectat criteriul decompozibilitatii, dar nu si cel al compozibilitatii/

**35. Un avantaj principal al stilului arhitectural Stratificat este asigurarea criteriului de modularitate al protectiei.**

Afirmatia este adevarata. Criteriul de modularitate al protectiei spune ca efectele unei conditii de executie anormale sunt limitate la cateva module. Stilul arhitectural Stratificat indeplineste cu siguranta acest criteriu, incat, in cazul in care se modifica interfata unui Layer sau se adauga facilitati noi, doar layerele adiacente vor fi afectate, nu tot sistemul.

**36. Procesul de dezvoltare Extreme Programming spune ca majoritatea timpului ar trebui investit in programare si mai putin in alte activitati conexe precum pactarea cerintelor/testare, incat produsul software este creat prin programare.**

Extreme programming este un proces de dezvoltare ce abordeaza extrem dezvoltarea iterativa: o noua versiune in fiecare zi/noapte, incrementii sunt livrati la fiecare 2-3 saptamani, iar toate teste trebuiesc rulate pentru fiecare constructie. Afirmatia este insa falsa, incat EP nu spune ca majoritatea timpului ar trebui investit in programare, incat testarea, designul si alte activitati sunt la fel de importante.

**40. Use Case-urile de tip Fish Level se folosesc atunci cand vrem sa detaliem fiecare actiune principala din cadrul unui use case de tip sea Level.**

Use Case-urile de tip Sea Level sunt acele care descriu interactiunea dintre user si sistem, interactiunea fiind majora, iar scopul bine precizat. Use case-urile de tip Fish Level le folosim cand vrem sa factorizam functionalitati comune folosind <<include>>. In mod cert, afirmatia este falsa.

**41. Stim ca am descoperit toate Use-Caseurile dintr-un sistem atunci cand toate uc sunt acoperite de use case-uri Sea Level, fiecare dintre acestea trebuind sa fie conectate cu cate cel putin un actor.**

Use case-urile de tip Sea Level sunt folosite pentru a descrie interactiunea dintre sistem si actori, interactiunile fiind majore si cu un scop bine precizat. Astfel, putem intr-adevar spune ca am descoperit toate use-caseurile din sistem cand toate uc sunt acoperite de uc Sea level. De asemenea, este adevarat ca fiecare dintre aceste trebuie conectat cu cate cel putin un actor, caci altfel nu ar avea sens, un use case presupun existenta a minim un user care sa il foloseasca.

**42. In tehnica CRC clasele sunt identificate pornind de la substantivele din descrierea uc, iar responsabilitatea dse identifica dintre verbele folosite in cadrul descrierii.**

**Adevarat.** In tehnica CRC clasele sunt identificate pornind de la substantive, incat POO CERE CA SI CLASELE SA POATA DEFINI ENTITATI ALE SISTEMULUI, deci ele nu pot fi decat substantive, iar responsabilitatile claselor reprezinta functionalitatea lor REDATA PRIN METODE,, iar aceasta functionalitate este redata de verbele din scenariul unei diagrame use case/

**43. In stilul architectural Repository diferitele componente care proceseaza date sunt total independente unele de altele, cu exceptia faptului ca folosesc acelasi model de date.**

Stilul repository este indicat de folosit pentru distribuirea unor cantitati mari de date, consumatorii si producatorii fiind independenti intre ei. Intr-adevar un dezavantaj este faptul ca folosesc acelasi model de date. Astfel, afirmatia este adevarata.

**44. Un program care nu are cicluri are numarul ciclomatic = 1**

Complexitatea cilcomatica este influentata de numarul de decizii simple din program. Astfel, este absurd sa credem ca daca un program nu are cicluri, acesta are complexitate = 1, incat si instructiunile de tip if/else contin astfel de decizii. Astfel, numarul ciclomatic depinde de cum este structurat programul.

**45. Principalul dezavantaj al testarii top down integration este crearea de stuburi si drivere.**

In cazul integrarii de tip top down se integreaza modulele de sus in jos, testand intai modulul principal. Intr-adevar un dezavantaj al testarii top-down este necesitatea de a crea stub-rui, care inlocuiesc modulele care sunt subordonate componentei testate, insa in cazul acestei testari nu se creeaza drivere, acestea din urma fiind caracteristice testarii bottom-up, in cadrul careia integrarea si testarea incepe de jos in sus, testand intai cele mai mici module. Driverele se folosesc in cadrul testarii bottom up drept un program principal in care se apeleaza functia testata.

**46. Ideea de baza in testarea whitebox e aceea de a ne asigura ca cel putin privita individual functia nu are niciun bug in nicio instructiune, incat toate instructiunile sunt verificate cu cate cel putin un test.**

Afirmatia este adevarata in totalita. In cazul testarii de tip white box, dupa cum spunea Pressman scopul e sa ne asiguram ca fiecare conditie a fost executata macar o data, deci implicit, ca functia nu are niciun bun in nicio instructiune cel putin privita individual. In cazul acestui tip de testare se face un minim de teste astfel: se parcurge bucla de 0,1,2 ori, de max-1 si de max ori.

**47. Partitiile echivalente se folosesc la sistemele cu domeniul datelor de intrare foarte mare si reprezinta clase de input-uri asemanatoare din punct de vedere al testarii. Astfel, pentru fiecare partitie se aleg una sau mai multe functii pentru care se vor scrie suite black box.**

Intr-adevar partitiile echivalente se folosesc in cazul sistemelor cu un domeniu al datelor de intrare foarte mare, in cadrul careia clasele au inputuri asemanatoare din punct de vedere al testarii. Ceea ce face enuntul sa fie fals, este a doua propozitie, incat in cazul partitiilor echivalente testelete se fac pentru valorile corespunzatoare marginilor si mijlocului.

**48. Daca dorim sa verificam cat mai timpuriu principalele puncte de control si de decizie din sistem vom alege testarea de integrare de timp Bottom up.**

Testarea de integrare de tip Bottom up presupune integrarea si testarea modulelor de jos in sus, intai fiind testate cele mai mici module, astfel ca prin aceasta metoda nu se verifica timpuriu principalele puncte de control si de decizie din sistem, acest avantaj fiind caracteristic testarii Top Down, in cadrul careia sunt integrate modulele de sus in jos, deci prima data se testeaza modulul principal.

**49. Valoarea complexitatii ciclomatice a unei functii nu e influentata de numarul de instructiuni de ciclare incat singurele instructiuni care incrementeaza caloare complexitatii sunt cele de decizie de tip if-else.**

Afirmatia este cu siguranta falsa. Complexitatea ciclomatica ne spune cat de complex este codul, iar aceasta poate fi calculata ca fiind numarul de decizii simple + 1. Este drept ca instructiunile de decizie de tip if-else contin astfel de decizii simple, insa acest fapt nu inseamna ca instructiunile de ciclare nu contin astfel de decizii, incat complexitatea ciclomatica a unui program este influentata si de de buclele for, while etc.

**50. Principalul motiv pentru care nu este recomandata testarea de tip Big Bang e acela ca aceasta depisteaza mai putine erori.**

Afirmatia este falsa. In cazul testarii de tip Big Bang tot programul este testat o singura data, motiv pentru care se creeaza un haos si este extrem de dificila depistarea erorilor, iar corectarea lor la fel de dificila incat cand aceste sunt corectate apar altele, iar testarea pare ca intra intr-o bucla infinita.

**51. Testarea ciclurilor pentru o functie cu doua cicluri imbricate implica scrierea a doua cazuri de test: unul care sa parcurga instructiunile din ciclul interior si un al doilea care sa testeze instructiunile din ciclul exterior.**

Afirmatia este adevarata. In cazul functiei cu doua cicluri imbricate intai se testeaza bucla din interior( care se face ca testarea unei bucle simple) pastrand iteratioul buclei exterioare la valoare minima. Apoi, se aplica acelasi procedeu si in cazul buclei exterioare, iar acelasi procedeu s-ar fi aplicat indiferent de cate cicluri imbricate ar fi fost.

**52. Precizati ce stil arhitectural s-ar asocia cel mai bine cu fiecare dintre imagini:**

**a. O ceapa**→In mod evident, aceasta imagine ne duce cu gandul la stilul arhitectural Layered, tocmai datorita aspectului stratificat, fiecare strat comunicand doar cu straturile adiacente.

**b. O linie de productie**→Aceasta imagine ne duce cu gandul la stilul arhitectural pipes and filters. Filtrele din imagine le asociem cu diferite masinarii care prelucreaza datele, respectiv transforma

materiile prime in produs finit, iar conductele le asociem cu scarile rulante care transporta produsele prin fabrica.

c.IDE→ Ne duce cu gandul la Repository, in cadrul caruia se gasesc mai multe unelte, independente intre ele care folosesc un model comun de date

**Numarul ciclomatic ne arata numarul de “cai” de executie posibile pentru un anumit program.**

- 7.1. Forma reala a curbei ratei defectarii software-ului este datorata faptului ca cerintele nu sunt intelese clar de la bun inceput, ceea ce duce la necesitatea de schimbari continue asupra sistemului.

Fals. Curba reala difera de cea ideală din cauza apariției schimbărilor repetitive în soft. Schimbările apar atunci când se efectuează lucrări de menținere asupra softului sau modificări asupra softului.

2. 50% din timp și costuri ar trebui alocate testării, dar în multe cazuri procentajul alocat testării este mai mic.

Fals. Testarea reprezintă într-adevar jumătate din costuri și din timp, dar aceste estimări nu sunt fictive, deoarece chiar atât reprezintă, trebuie să facem multă testare pe lângă partea de cod pentru a ne asigura că produsul software este bun și face ce trebuie.

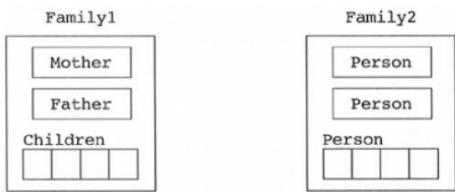
3. Conceptul de time boxing spune că sarcinile alocate pentru o anumită iteratie trebuie terminate în cadrul iteratiei în cauză.

Adevarat. Conceptul de time boxing aloca o perioadă fixă fiecărei iterării, numita time box. În cadrul acestei perioade de timp sarcinile alocate în cadrul iterării trebuie terminate. În cazul în care nu se termină task-urile alocate deadline-ul nu se modifică, mai degrabă se amână niste task-uri pentru time box-ul viitor.

4. Dacă lucram cu procese de dezvoltare agile, asta înseamnă că nu trebuie să intelegem clar cerintele sistemului.

Fals, cerintele sistemului trebuie intelese indiferent de procesele de dezvoltare, dar în cazul proceselor de dezvoltare agile acestea se pot modifica de la o iteratie la alta sau se mai pot adăuga alte cerinte de la o iteratie la alta, important este că acestea să nu se schimbe în timpul unei iterării.

5. Pentru a evita problema proliferării claselor, ar trebui să implementăm clasa Family ca mai jos.



Adevarat. Pentru a evita proliferarea claselor trebuie să fim atenți la comportamentul acestora. El este cel care decide: dacă comportamentul difera avem clase diferite, dacă nu difera avem roluri diferite ale acelasi clase.

6. Dacă avem diagrame de clasa UML explicite pentru un sistem, putem deduce diagramele de secvență pentru acel sistem, adică toate interacțiunile posibile dintre obiecte.

Fals. Diagramalele de clasa ne arată structura sistemului, atributele și metodele fiecărei clase, pe când diagramele de secvență modelează scenariile dintr-un sistem. Acestea nu se pot deduce unele din altele.

7. Un sistem nu poate fi actor pentru un alt sistem pentru ca acest lucru ar implica accesul la informatiile din sistemul mare, ceea ce incalca incapsularea.

Fals, un sistem poate fi actor pentru un alt sistem, intrucat acesta interactioneaza cu interfata acestuia si nu are acces la codul acestuia, astfel nu se incalca principiul incapsularii.

8. Stilul architectural *Pipes and Filters* este avantajos din punct de vedere al timpului.

Fals. Acesta este format din filtre care sunt independente unele fata de celelalte si care prelucreaza datele si din conducte care sunt cai prin care datele circula de la filtru la filtru. Filtrele nu au un format comun de date, iar din aceasta cauza fiecare filtru trebuie sa faca "parse" si "un-parse" datelor, lucru care duce la overhead si, implicit, la pierderea timpului

9. Un program care nu are cicluri (for, while) are numarul ciclomatic=1.

Fals. Numarul ciclomatic este influentat de numarul deciziilor simple din cadrul unui program. Instructiunile de ciclare de tip for/while contin si ele o astfel de instructiune, asadar numarul ciclomatic in cazul in care nu avem instructiuni de ciclare in program este influentat doar de numarul deciziilor simple din cadrul acestuia.

10. Principalul dezavantaj al testarii top down integration este crearea de drivere si stub-uri .

Fals. In cazul testarii de tip top-down se creeaza doar stub-uri, care imita comportamentul metodelor apelate din functia testata. Driver-ele sunt create la testarea de tip bottom-up si reprezinta un program principal simplu din care se apeleaza functia testata.

11. Folosirea constantelor globale incalca criteriul de modularitate al continuitatii.

Fals. Folosirea constantelor globale nu incalca principiul continuitatii.

12. Curba reală a evoluției în timp a ratei de defecte (Failure Rate) diferă față de cea ideală în principal din cauza lipsei de experiență a programatorilor.

Fals, deoarece curba reala difera de cea ideală din cauza aparitiei schimbarilor repeatate in soft. Schimbarile apar atunci cand se efectueaza lucrari de mentenanta asupra softului. Lipsa de experienta a programatorilor nu este un motiv principal de aparitie a erorilor in soft.

13. Legile evoluției software-ului ale lui Lehman nu se aplică la sisteme ale căror cerințe sunt perfect înțelese de la bun început, pentru că altfel de sisteme nu au nici un motiv să evolueze.

Fals, deoarece legile evolutiei software se aplica oricarui sistem indiferent daca cerintele sunt intelese sau nu de la bun inceput. Sistemele software oricum evolueaza datorita schimbarilor inconjuratoare.

14. În procesul de dezvoltare Scrum, Burndown Char ne arată evoluția efortului de-a lungul întregii istorii a proiectului, mai exact oferă informații despre Sprint-urile deja încheiate, precum și numărul de Task-uri finalizate până în prezent în întregul proiect.

Fals. Burndown Chart ne arata evolutia efortului si numarul de task-uri finalizate si de finalizat de-a lungul unui Sprint, perioada in care se realizeaza un Increment dintr-un proiect. Burndown Chart-ul nu arata evolutia efortului pentru intregul proiect.

15. O problemă importantă a proceselor de dezvoltare iterative este aceea că adesea trebuie modificat codul care a fost scris într-o iteracție anterioară, și uneori anumite porțiuni de cod trebuie chiar șterse, ceea ce reprezintă o pierdere/risipă.

Corect. Aceasta este o problema importantă a proceselor de dezvoltare iterative și este o risipa, în schimb, putem să refacem codul decât să pierdem timpul modificand același cod. Singurul cod care se pastrează este codul funcțional care face ceva util în cadrul softului.

16. În diagramele de UML de Use-Case relațiile <<extends>> și <<include>> sunt foarte asemănătoare și pot fi folosite interschimbabil însăcumă ambele sunt folosite pentru a da “factor comun” descrierea unei anumite funcționalități.

Fals. Relațiile <<extends>> și <<include>> sunt ceea ce poate să fie diferite. Prima reprezintă un caz exceptionál al unui use case, pe când cea de-a două factorizează un comportament comun al mai multor use case-uri.

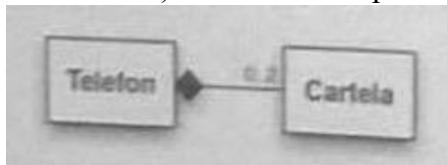
17. Diagramele UML de secvență (sequence diagrams) sunt folosite ca punct de plecare pentru sesiunile de CRC Cards în urma cărora se construiesc diagramele UML de clase (class diagrams).

Fals. Punctul de plecare pentru a crea un CRC card sunt diagramele use case care redau un scenariu identificând actorii și interacțiunile lor cu sistemul. În urma creării cardurilor CRC se realizează diagramele de clasa și apoi diagramele de secvență.

18. Într-o diagramă de secvență UML, toate obiectele care sunt instanțe ale aceleiași clase trebuie “comasate” într-un singur dreptunghi; adică nu este admis sau recomandat să reprezentăm fiecare obiect ca un dreptunghi separat, pentru că altfel s-ar încărca prea mult diagrama.

Fals. Toate instantele unei clase trebuie desenate în dreptunghiuri separate, întrucât fiecare obiect poate avea un comportament diferit chiar dacă sunt instante ale aceleiasi clase.

19. Relația din figura de mai jos modelează cazul unui telefon dual-sim (care poate ține simultan două cartele) cu cartele care pot fi oricând reutilizate și în alt telefon.



Adevarat. Avem clasa Telefon și clasa Cartela cu o relație de compunere între ele. Întradevar, un obiect Cartela nu poate exista fără un obiect Telefon, dar putem avea instante ale clasei Cartela și în alte obiecte Telefon.

20. Un sistem care respectă criteriul de modularitate al Decompozabilității îl va respecta aproape sigur și pe cel al Compozabilității întrucât acesta din urmă se referă la posibilitatea compunerii unui sistem din module.

Fals. Dacă un sistem respectă criteriul de modularizare al decompozabilității nu înseamnă că îl va respecta și pe cel al compozabilității. Exemplu cu Ikea și lego.

21. Un avantaj principal al stilului arhitectural Stratificat (Layered Architecture) este asigurarea criteriului de modularitate al Protecției.

Adevarat. În cazul stilului arhitectural stratificat cand se modifică interfața unui layer sau se adaugă noi facilități este afectat doar layer-ul adjacente, nu întregul sistem.

22. Ideea de bază în testarea whitebox este aceea de a ne asigura că cel puțin privită individual o funcție nu are nici un bug în nici o instrucțiune, întrucât toate instrucțiunile sunt verificate cu câte cel puțin un test.

Adevarat. Testarea de tip whitebox verifică toate instrucțiunile dintr-o funcție cu cel puțin un test. Astfel, funcția cel puțin privată individual nu are niciun bug.

23. Partițiile echivalente se folosesc la sisteme cu foarte multe funcții lungi și complexe, și reprezintă grupuri de funcții care sunt asemănătoare din punctul de vedere al testării. Astfel, din fiecare grup (partiție) se vor alege una sau mai multe funcții pentru care se vor scrie suite de teste blackbox.

Fals. Partițiile echivalente se folosesc la sistemele cu domeniul datelor de intrare foarte mare și reprezintă clase de input-uri asemănătoare din punct de vedere al testării. Astfel, pentru fiecare partiție se aleg cazuri de teste pentru valorile corespunzătoare marginilor și mijlocului.

24. Legile evoluției software-ului enumărate de către Lehman ne spun că nu există nici o modalitate de a încetini declinul calității software-ului.
- Fals. Legile scaderii calitatii a lui Lehman spun ca putem incetini procesul de deteriorare al software-ului daca il adaptam in mod continuu la schimbarile inconjuratoare.
25. Printr-o proiectare riguroasă, implementarea software-ului poate fi transformată într-o activitate aproape perfect partaționabilă.
- Fals, o proiectare riguroasa ajuta intr-adevar la partitionarea implementarii softului, dar nu garanteaza o partitionare perfecta. Asta depinde de natura produsul, daca prin natura lui raportat la tehnologiile existente, el poate fi sau nu partitionat cat mai bine.
26. Procesul de dezvoltare Extreme Programming spune că majoritatea efortului trebuie investit în programare și mai puțin în alte activități conexe cum ar fi captarea cerințelor sau testare, întrucât până la urmă produsul software propriu-zis este creat prin programare.
- Fals, Extreme Programming nu se refera la programarea in sensul de a scrie cod si atat. Este o metodologie bazata pe Agile care abordeaza extrem dezvoltarea iterativa. Este scrisa o noua versiune foarte des, incrementii sunt livrati clientului la fiecare doua trei saptamani, iar constructia e aprobată doar daca testele sunt trecute cu succes.
27. Deși în procesele de dezvoltare iterative și incrementate trebuie uneori schimbate sau chiar rescrise complet fragmente semnificative de cod ce au fost scrise în iterațiile anterioare, acestea nu reprezintă o pierdere întrucât schimbările sunt inevitabile.
- Fals. Aceasta este o problema importantă a proceselor de dezvoltare iterative și este o risipa, in schimb, putem sa refacem codul decat sa pierdem timpul modificand acelasi cod. Singurul cod care se pastreaza este codul functional care face ceva util in cadrul softului.
28. Tehnica de analiza cerințelor folosind Use-Case-uri este specifică procesului de dezvoltare Waterfall pentru că aici cerințele trebuie analizate riguros la începutul proiectului.
- Fals. Intr-adevar, procesul de tip Waterfall se foloseste atunci cand cerintele sunt foarte bine cunoscute de la inceput, dar asta nu inseamna ca folosirea use case-urilor este specifică doar acestui tip de proces.
29. Doi sau mai mulți actori nu pot fi asociați (adică nu pot interacționa) cu același use-case pentru că aceasta ar însemna că sunt redundanți.
- Fals. Doi actori pot avea același use case printre altele atata timp cat interactionarea lor cu sistemul este diferita in ansamblu. (Nu interactioneaza cu sistemul pentru aceeasi functionalitate a sistemului).

30. Într-o diagramă de secvență se recomandă ca dacă apar mai multe instanțe ale aceleiași clase acestea să fie reprezentate într-un singur element pentru a nu se încărca excesiv diagrama.

Fals. Toate instantele unei clase trebuie desenate în dreptunghiuri separate, întrucât fiecare obiect poate avea un comportament diferit chiar dacă sunt instante ale aceleiasi clase.

31. Pentru a evita problema proliferării claselor prin modelare ca și clase a entităților externe sistemului trebuie să ținem cont că sunt clase doar acele entități care apelează alte entități (clase) din sistem.

Fals. Clasele sunt acelea care sunt apelate (primesc mesaje), nu cele care apelează (transmit mesaje).

32. Un corp de mobilă modular care vine împachetat pe bucăți și care trebuie să îl asamblăm/compunem (gen IKEA) a fost proiectat urmărind în special criteriul de modularitate al compozabilității.

Fals. Un corp de mobila modular de la Ikea a fost proiectat în astă fel încât să se poată realiza din acele piese numai un singur corp și numai într-un anumit fel. Acest lucru este în contradicție cu principiul modular al compozibilității care spune că putem compune un sistem modular în mod liber și cum ne dorim.

33. Importanța criteriilor și regulilor de modularitate este cu atât mai mare cu cât anvergura sistemului software proiectat este mai mare, cu alte cuvinte: importanța modularizării este proporțională cu dimensiunea sistemului software.

Adevarat. Într-un soft mic aproape că nu avem nevoie de modularizare pentru că nu avem în ce bucati să spargem. Pe când într-un soft mare este important să spargem pe bucati pentru structura frumoasă și pentru a realizare a comunicare între componente ușor de urmarit și componente ușor de reutilizat.

34. Dacă dorim să verificăm cât mai timpuriu principalele puncte de control și decizie din sistem vom alege testarea de integrare de tip Bottom-Up.

Fals. Testarea de integrare Bottom-Up verifică timpuriu procesarea de date de nivel scazut. Pentru a verifica că mai timpuriu principalele puncte de control și decizie din sistem vom alege testarea de integrare de tip Top-Down.

35. Valoarea complexității ciclomатice a unei funcții nu este influențată de numărul de instrucțiuni de ciclare (for/while) întrucât singurele instrucțiuni care incrementează valoarea complexității ciclomатice sunt cele de decizie de tip if-else.

Fals. Valoarea complexității ciclomатice este influențată atât de deciziile de tip if/else, cât și de numărul de instrucțiuni de ciclare (for/while), deoarece și aceste instrucțiuni contin câte o decizie simplă.

36. Curba defectelor poate fi făcută să tindă în timp spre zero dacă cerințele sunt bine înțelese de la început, și dacă se aplică sistematic metode de testare eficiente.

Fals. Chiar dacă sunt bine înțelese cerințele de la început și se aplică metode de testare eficiente, în timp, din cauza schimbărilor repetitive tot vor apărea erori în sistem, deci curba defectelor nu va tinde în timp spre 0.

37. Utilizarea eficientă a instrumentelor CASE dedicate programării poate reduce semnificativ costurile totale ale proiectului având în vedere că într-un proiect software o parte semnificativă a costurilor sunt legate de activitatea de codare.

Fals. Utilizarea eficientă CASE reduce costurile, dar ele se folosesc nu doar pentru partea de codare, ci și pentru alte etape ale procesului precum specificații, design, testare, debugging. De-asemenea, o mare parte din costuri nu se duc pe partea de codare, ci pe partea de testare și planificare.

38. În procesul de dezvoltare Waterfall se face doar un singur tip de activitate la un moment dat, spre deosebire de cele iterative unde într-o iteratăie trebuie efectuate toate tipurile de activitate.

Adevarat. Procesul de dezvoltare Waterfall împarte proiectul în mai multe activități ce sunt rezolvate pe rand. De exemplu, se realizează analiza și definirea cerințelor, după ce se încheie aceasta fază se realizează design-ul de sistem și software și astăzi departe.

39. În procesul Rational Unified Process (RUP), deși este permisă efectuarea mai multor tipuri de activități în interiorul unei faze, se recomandă ca numărul de activități desfășurate în paralel să fie limitat.

Fals. Într-adevar, în cazul procesului Rational Unified Process putem efectua mai multe tipuri de activități simultan în interiorul unei faze, dar nu există nicio limitare cu privire la numărul de procese desfășurate în paralel. De verificat

40. Use-Case-urile de tip Fish Level se folosesc atunci când vrem să detaliem fiecare acțiune principală din cadrul unui use-case de tip Sea Level.

Fals. Use Case-urile de tip Sea Level ne arată cum userul interacționează cu sistemul, interacțiunile sunt majore, iar scopul este bine precizat. Use Case-urile de tip Fish Level sunt acele use case-uri care se dau factor comun din use case-urile Sea-Level (cu <<include>>).

41. Știm că am descoperit toate Use-Case-urile dintr-un sistem atunci când toate cerințele funcționale sunt acoperite de use-case-uri Sea Level, fiecare dintre acestea trebuind să fie conectate cu cel puțin un actor.

Adevarat. Use case-urile de tip Sea Level prin definiție reprezintă interacțiuni majore ale user-ilor cu sistemul cu un scop precis. Astfel dacă toate cerințele funcționale sunt acoperite de Use Case-uri de tip Sea Level am descoperit toate Use Case-urile din

sistem. Si da, trebuie ca fiecare use case sea level sa fie legat la un user pentru ca altfel el nu are sens, un use case presupune existenta cel putin a unui user care sa-l foloseasca.

42. În tehnica CRC clasele sunt identificate pornind de la substantivale din descrierea use-case-urilor, iar responsabilitatea se identifică dintre verbele folosite în cadrul descrierii scenariilor.

**Adevarat.** In tehnica CRC clasele sunt identificate pornind de la substantivale, intrucat programarea orientata pe obiecte cere ca si clasele sa poata defini entitati ale sistemului, deci ele nu pot fi decat substantivale, iar responsabilitatile claselor reprezinta functionalitatea lor redată prin metode, iar aceasta functionalitate este data de verbele din scenariul unei diagrame use case.

43. Relația de compoziție este un caz special de agregare (relație parte-întreg) în care indicăm ca distrugerea clasei "întreg", atrage după sine și distrugerea obiectelor "parte" conținute de către clasa "întreg".

**Adevarat.** Relatia de compositie este o forma de agregare in care componentele nu pot exista una fara cealalta.

44. Între specificatorul de acces "protected" și criteriul de modularitate al protecției există o legătură strânsă.

**Fals.** Specificatorul de acces „protected” face ca atributele și metodele unei clase să fie vizibile doar de clasele care mostenesc clasa respectiva. Criteriul de modularitate al protecției ne spune că dacă apare o eroare de execuție, aceasta se limitează la doar câteva module. Criteriul se poate asigura și cu specificatorul private, cu pachete etc.

45. În stilul arhitectural Repository diferențele componente care procesează date sunt total independente unele de altele, cu excepția faptului că folosesc același model de date.

**Adevarat.** Stilul arhitectural Repository este un mod eficient de a distribui mari cantități de date între producătorii și consumatorii de date sunt independenți, dar marele compromis este faptul că folosesc același model de date.

46. Principalul motiv pentru care testarea de integrare de tip Big Bang nu este recomandată este acela că spre deosebire altor tehnici de testare de integrare aceasta identifică mai puține defecte.

**Fals.** Testarea de integrare de tip Big Bang combina toate componente în avans și testează întreg programul. În acest fel se creează un haos cu multe erori care la prima vedere nu se leagă. Corectarea acestora se realizează greu, după aceasta rezultă alte erori și astfel testarea pare să intre într-o buclă infinită.

47. Testarea ciclurilor pentru o funcție cu două cicluri imbricate(nested) implică scrierea a două cazuri de test; unul să parcurgă instrucțiunile din ciclul interior, și un altul care să testeze instrucțiunile din ciclul exterior.

**Adevarat.** Testarea funcțiilor cu 2 cicluri imbricate începe cu testarea buclei din interior care se face ca testarea unei bucle simple să parcurgă iteratorul buclei exterioare la valoarea minima. Dupa aceasta se trece la testarea celei de a două bucle în maniera testării unei bucle simple.

49. Precizati ce stil arhitectural s-ar asocia cel mai bine urmatoarele imagini:

- a. O ceapa - **Layered Architecture** intrucat aceasta are o structura stratificata, iar in mijloc se afla lastarul care este cea mai importanta parte a acesteia.
- b. O linie de productie - **conducte si filtre**. Conductele reprezinta benzile rulante care ajuta produsele sa se deplaseze prin fabrica, iarfiltrele sunt reprezentate de diferite masinarii care transforma simplele materii prime in produsul finit, pas cu pas.
- c. Un mediu de dezvoltare integrat (IDE), gen Eclipse, Visual Studio, JBuilder sau un instrument CASE complex - **Repository** deoarece un mediu de dezvoltare are mai multe unele: compilator, debugger etc, care sunt independente intre ele si care creeaza si folosesc acelasi tip de date.

1. **Forma reala a curbei ratei defectarii software-ului este datorata faptului ca cerintele nu sunt intelese clar de la bun inceput, ceea ce duce la necesitatea de schimbari continue asupra sistemului.**

Fals. Curba reala difera de cea ideală din cauza apariției schimbărilor repetitive în soft. Schimbările apar atunci când se efectuează lucrări de menținere asupra softului sau modificări asupra softului.

2. **50% din timp și costuri ar trebui alocate testării, dar în multe cazuri procentajul alocat testării este mai mic.**

Fals. Testarea reprezintă într-adevar jumătate din costuri și din timp, dar aceste estimări nu sunt fictive, deoarece chiar atât reprezintă, trebuie să facem multă testare pe lângă partea de cod pentru a ne asigura că produsul software este bun și face ce trebuie.

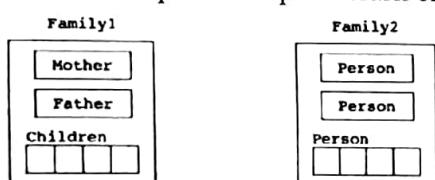
3. **Conceptul de time boxing spune că sarcinile alocate pentru o anumita iteratie trebuie terminate în cadrul iteratiei în cauză.**

Adevarat. Conceptul de time boxing aloca o perioadă fixă fiecărei iterării, numită time box. În cadrul acestei perioade de timp sarcinile alocate în cadrul iterării trebuie terminate. În cazul în care nu se termină task-urile alocate deadline-ul nu se modifică, mai degrabă se amână niste task-uri pentru time box-ul viitor.

4. **Dacă lucram cu procese de dezvoltare agile, asta înseamnă că nu trebuie să intelegem clar cerințele sistemului.**

Fals, cerințele sistemului trebuie intelese indiferent de procesele de dezvoltare, dar în cazul proceselor de dezvoltare agile acestea se pot modifica de la o iteratie la alta sau se mai pot adăuga alte cerinte de la o iteratie la alta, important este că acestea să nu se schimbe în timpul unei iterării.

5. **Pentru a evita problema proliferării claselor, ar trebui să implementăm clasa Family ca mai jos.**



Adevarat. Pentru a evita proliferarea claselor trebuie să fim atenți la comportamentul acestora. El este cel care decide: dacă comportamentul difera avem clase diferite, dacă nu difera avem roluri diferite ale acelasi clase.

6. **Dacă avem diagrame de clasa UML explicite pentru un sistem, putem deduce diagramele de secvență pentru acel sistem, adică toate interacțiunile posibile dintre obiecte.**

Fals. Diagramalele de clasa ne arată structura sistemului, atributele și metodele fiecărei clase, pe când diagramalele de secvență modelează scenariile dintr-un sistem. Acestea nu se pot deduce unele din altele.

7. Un sistem nu poate fi actor pentru un alt sistem pentru ca acest lucru ar implica accesul la informatiile din sistemul mare, ceea ce incalca incapsularea.

Fals, un sistem poate fi actor pentru un alt sistem, intrucat acesta interacioneaza cu interfata acestuia si nu are acces la codul acestuia, astfel nu se incalca principiul incapsularii.

8. Stilul architectural *Pipes and Filters* este avantajos din punct de vedere al timpului.

Fals. Acesta este format din filtre care sunt independente unele fata de celelalte si care prelucreaza datele si din conducte care sunt cai prin care datele circula de la filtru la filtru. Filtrele nu au un format comun de date, iar din aceasta cauza fiecare filtru trebuie sa faca "parse" si "un-parse" datelor, lucru care duce la overhead si, implicit, la pierderea timpului.

9. Un program care nu are cicluri (for, while) are numarul ciclomatic=1.

Fals. Numarul ciclomatic este influentat de numarul deciziilor simple din cadrul unui program. Instructiunile de ciclare de tip for/while contin si ele o astfel de instructiune, asadar numarul ciclomatic in cazul in care nu avem instructiuni de ciclare in program este influentat doar de numarul deciziilor simple din cadrul acestuia.

10. Principalul dezavantaj al testarii top down integration este crearea de drivere si stub-uri .

Fals. In cazul testarii de tip top-down se creeaza doar stub-uri, care imita comportamentul metodelor apelate din functia testata. Driver-ele sunt create la testarea de tip bottom-up si reprezinta un program principal simplu din care se apeleaza functia testata.

11. Folosirea constantelor globale incalca criteriul de modularitate al continuitatii.

~~Fals. Folosirea constantelor globale nu incalca principiul continuitatii.~~

12.Curba reală a evoluției în timp a ratei de defecte (Failure Rate) diferă față de cea ideală în principal din cauza lipsei de experiență a programatorilor.

Fals, deoarece curba reala difera de cea ideală din cauza aparitiei schimbarilor repeatate in soft. Schimbarile apar atunci cand se efectueaza lucrari de mentenanta asupra softului. Lipsa de experienta a programatorilor nu este un motiv principal de aparitie a erorilor in soft.

13. Legile evoluției software-ului ale lui Lehman nu se aplică la sisteme ale căror cerințe sunt perfect înțelese de la bun început, pentru că altfel de sisteme nu au nici un motiv să evolueze.

Fals, deoarece legile evolutiei software se aplica oricarui sistem indiferent daca cerintele sunt intelese sau nu de la bun inceput. Sistemele software oricum evolueaza datorita schimbarilor inconjuratoare.

14. În procesul de dezvoltare Scrum, Burndown Char ne arată evoluția efortului de-a lungul întregii istorii a proiectului, mai exact oferă informații despre Sprint-urile deja încheiate, precum și numărul de Task-uri finalizate până în prezent în întregul proiect.

Fals. Burndown Chart ne arată evoluția efortului și numărul de task-uri finalizate și de finalizat de-a lungul unui Sprint, perioada în care se realizează un Increment dintr-un proiect. Burndown Chart-ul nu arată evoluția efortului pentru întregul proiect.

15. O problemă importantă a proceselor de dezvoltare iterative este aceea că adesea trebuie modificat codul care a fost scris într-o iterare anterioară, și uneori anumite porțiuni de cod trebuie chiar șterse, ceea ce reprezintă o pierdere/risipă.

~~Fals - este una hinc decât să avem soft prost~~  
Corect. Aceasta este o problema importantă a proceselor de dezvoltare iterative și este o risipă, în schimb, putem să refacem codul decât să pierdem timpul modificând același cod. Singurul cod care se pastrează este codul funcțional care face ceva util în cadrul softului.

16. În diagramele de UML de Use-Case relațiile <<extends>> și <<include>> sunt foarte asemănătoare și pot fi folosite interschimbabil încărcăt ambele sunt folosite pentru a da "factor comun" descrierea unei anumite funcționalități.

Fals. Relațiile <<extends>> și <<include>> sunt ca și cum se poate de diferențe. Prima reprezintă un caz exceptional al unui use case, pe când cea de-a două factorizează un comportament comun al mai multor use case-uri.

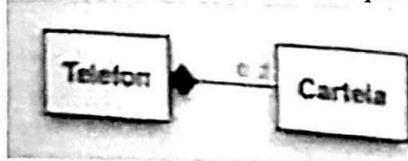
17. Diagramele UML de secvență (sequence diagrams) sunt folosite ca punct de plecare pentru sesiunile de CRC Cards în urma cărora se construiesc diagramele UML de clase (class diagrams).

Fals. Punctul de plecare pentru a crea un CRC card sunt diagramele use case care redau un scenariu identificând actorii și interacțiunile lor cu sistemul. În urma creării cardurilor CRC se realizează diagramele de clasa și apoi diagramele de secvență.

18. Într-o diagramă de secvență UML, toate obiectele care sunt instanțe ale aceleiași clase trebuie "comasate" într-un singur dreptunghi; adică nu este admis sau recomandat să reprezentăm fiecare obiect ca un dreptunghi separat, pentru că altfel s-ar încărca prea mult diagrama.

Fals. Toate instantele unei clase trebuie desenate în dreptunghiuri separate, întrucât fiecare obiect poate avea un comportament diferit chiar dacă sunt instante ale aceleiasi clase.

19. Relația din figura de mai jos modelează cazul unui telefon dual-sim (care poate ține simultan două cartele) cu cartele care pot fi oricând reutilizate și în alt telefon.



*RELATIE => refolosire  
COMPOUNERE => nu refolosire*

~~Adevarat~~. Avem clasa Telefon și clasa Cartela cu o relație de compunere între ele. Într-adevar, un obiect Cartela nu poate exista fără un obiect Telefon, dar putem avea instante ale clasei Cartela și în alte obiecte Telefon.

20. Un sistem care respectă criteriul de modularitate al Decompozabilității îl va respecta aproape sigur și pe cel al Compozabilității întrucât acesta din urmă se referă la posibilitatea compunerii unui sistem din module.

Fals. Dacă un sistem respectă criteriul de modularizare al decompozabilității nu înseamnă că îl va respecta și pe cel al compozabilității. Exemplu cu IKEA și LEGO.

21. Un avantaj principal al stilului arhitectural Stratificat (Layered Architecture) este asigurarea criteriului de modularitate al Protecției.

Adevarat. În cazul stilului arhitectural stratificat când se modifică interfața unui layer sau se adaugă noi facilități este afectat doar layer-ul adjacente, nu întregul sistem.

22. Ideea de bază în testarea whitebox este aceea de a ne asigura că cel puțin privită individual o funcție nu are nici un bug și nici o instrucțiune, întrucât toate instrucțiunile sunt verificate cu câte cel puțin un test.

Adevarat. Testarea de tip whitebox verifică toate instrucțiunile dintr-o funcție cu cel puțin un test. Astfel, funcția cel puțin privată individual nu are niciun bug.

23. Partițiile echivalente se folosesc la sisteme cu foarte multe funcții lungi și complexe, și reprezintă grupuri de funcții care sunt asemănătoare din punct de vedere al testării. Astfel, din fiecare grup (partiție) se vor alege una sau mai multe funcții pentru care se vor scrie suite de teste blackbox.

Fals. Partițiile echivalente se folosesc la sistemele cu domeniul datelor de intrare foarte mare și reprezintă clase de input-uri asemănătoare din punct de vedere al testării. Astfel, pentru fiecare partitie se aleg cazuri de teste pentru valorile corespunzătoare marginilor și mijlocului.

**24. Legile evoluției software-ului enunțate de către Lehman ne spun că nu există nici o modalitate de a încetini declinul calității software-ului.**

Fals. Legile scaderii calitatii a lui Lehman spun ca putem incetini procesul de deteriorare al software-ului daca il adaptam in mod continuu la schimbarile inconjuratoare.

**25. Printr-o proiectare riguroasă, implementarea software-ului poate fi transformată într-o activitate aproape perfect partaționabilă.**

Fals, o proiectare riguroasa ajuta intr-adevar la partitionarea implementarii softului, dar nu garanteaza o partitionare perfecta. Asta depinde de natura produsul, daca prin natura lui raportat la tehnologiile existente, el poate fi sau nu partitionat cat mai bine.

**26. Procesul de dezvoltare Extreme Programming spune că majoritatea efortului trebuie investit în programare și mai puțin în alte activități conexe cum ar fi captarea cerințelor sau testare, întrucât până la urmă produsul software propriu-zis este creat prin programare.**

Fals, Extreme Programming nu se refera la programarea in sensul de a scrie cod si atat. Este o metodologie bazata pe Agile care abordeaza extrem dezvoltarea iterativa. Este scrisa o noua versiune foarte des, incrementii sunt livrati clientului la fiecare doua trei saptamani, iar constructia e aprobată doar daca testele sunt trecute cu succes.

**27. Deși în procesele de dezvoltare iterative și incrementate trebuie uneori să schimbă sau chiar să rescrisce complet fragmente semnificative de cod ce au fost scrise în iterările anterioare, acestea nu reprezintă o pierdere întrucât schimbările sunt inevitabile.**

~~True~~  
~~Fals~~. Aceasta este o problema importanta a proceselor de dezvoltare iterative si este o risipa, in schimb, putem sa refacem codul decat sa pierdem timpul modificand acelasi cod. Singurul cod care se pastreaza este codul functional care face ceva util in cadrul softului.

**28. Tehnica de analiza cerințelor folosind Use-Case-uri este specifică procesului de dezvoltare Waterfall pentru că aici cerințele trebuie analizate riguros la începutul proiectului.**

Fals. Intr-adevar, procesul de tip Waterfall se foloseste atunci cand cerintele sunt foarte bine cunoscute de la inceput, dar asta nu inseamna ca folosirea use case-urilor este specifica doar acestui tip de proces.

**29. Doi sau mai mulți actori nu pot fi asociați (adică nu pot interacționa) cu același use-case pentru că aceasta ar însemna că sunt redundanți.**

Fals. Doi actori pot avea acelasi use case printre altele atata timp cat interactionarea lor cu sistemul este diferita in ansamblu. (Nu interactioneaza cu sistemul pentru aceeasi functionalitate a sistemului).

**30. Într-o diagramă de secvență se recomandă ca dacă apar mai multe instanțe ale aceleiași clase acestea să fie reprezentate într-un singur element pentru a nu se încărca excesiv diagrama.**

Fals. Toate instantele unei clase trebuie desenate în dreptunghiuri separate, intrucât fiecare obiect poate avea un comportament diferit chiar dacă sunt instante ale aceleiasi clase.

**31. Pentru a evita problema proliferării claselor prin modelare ca și clase a entităților externe sistemului trebuie să ținem cont că sunt clase doar acele entități care apelează alte entități (clase) din sistem.**

Fals. Clasele sunt aceleia care sunt apelate (primesc mesaje), nu cele care apelează (transmit mesaje).

**32. Un corp de mobilă modular care vine împachetat pe bucăți și care trebuie să îl asamblăm/compunem (gen IKEA) a fost proiectat urmărind în special criteriul de modularitate al compozabilității.**

Fals. Un corp de mobila modular de la Ikea a fost proiectat în astă fel încât să se poată realiza din acele piese numai un singur corp și numai într-un anumit fel. Acest lucru este în contradicție cu principiul modular al compozibilității care spune că putem compune un sistem modular în mod liber și cum ne dorim.

**33. Importanța criteriilor și regulilor de modularitate este cu atât mai mare cu cât anvergura sistemului software proiectat este mai mare, cu alte cuvinte: importanța modularizării este proporțională cu dimensiunea sistemului software.**

Adevarat. Într-un soft mic aproape că nu avem nevoie de modularizare pentru că nu avem în ce bucati să spargem. Pe când într-un soft mare e important să spargem pe bucati pentru structura frumoasă și pentru a realizare o comunicare între componente ușor de urmarit și componente ușor de reutilizat.

**34. Dacă dorim să verificăm cât mai timpuriu principalele puncte de control și decizie din sistem vom alege testarea de integrare de tip Bottom-Up.**

Fals. Testarea de integrare Bottom-Up verifică timpuriu procesarea de date de nivel scăzut. Pentru a verifica că mai timpuriu principalele puncte de control și decizie din sistem vom alege testarea de integrare de tip Top-Down.

**35. Valoarea complexității ciclomatrice a unei funcții nu este influențată de numărul de instrucțiuni de ciclare (for/while) întrucât singurele instrucțiuni care incrementează valoarea complexității ciclomatrice sunt cele de decizie de tip if-else.**

Fals. Valoarea complexității ciclomatrice este influențată atât de deciziile de tip if/else, cât și de numărul de instrucțiuni de ciclare (for/while), deoarece și aceste instrucțiuni contin câte o decizie simplă.

**36. Curba defectelor poate fi făcută să tindă în timp spre zero dacă cerințele sunt bine înțelese de la început, și dacă se aplică sistematic metode de testare eficiente.**

Fals. Chiar daca sunt bine intelese cerintele de la inceput si se aplica metode de testare eficiente, in timp, din cauza schimbarilor repeatate tot vor aparea erori in sistem, deci curba defectelor nu va tinde in timp spre 0.

**37. Utilizarea eficientă a instrumentelor CASE dedicate programării poate reduce semnificativ costurile totale ale proiectului având în vedere că într-un proiect software o parte semnificativă a costurilor sunt legate de activitatea de codare.**

Fals. Utilizarea eficienta CASE reduce costurile, dar ele se folosesc nu doar pentru partea de codare, ci si pentru alte etape ale procesului precum specificatii, design, testare, debugging. De-asemenea, o mare parte din costuri nu se duc pe partea de codare, ci pe partea de testare si planificare.

**38. În procesul de dezvoltare Waterfall se face doar un singur tip de activitate la un moment dat, spre deosebire de cele iterative unde într-o iteratie trebuie efectuate toate tipurile de activități.**

Adevarat. Procesul de dezvoltare Waterfall imparte proiectul in mai multe activitati ce sunt rezolvate pe rand. De exemplu, se realizeaza analiza si definirea cerintelor, dupa ce se incheie aceasta faza se realizeaza design-ul de sistem si software si asa mai departe.

**39. În procesul Rational Unified Process (RUP), deși este permisă efectuarea mai multor tipuri de activități în interiorul unei faze, se recomandă ca numărul de activități desfășurate în paralel să fie limitat.**

Fals. Intr-adevar, in cazul procesului Rational Unified Process putem efectua mai multe tipuri de activitati simultan in interiorul unei faze, dar nu exista nicio limitare cu privire la numarul de procese desfasurate in paralel. De verificat

**40. Use-Case-urile de tip Fish Level se folosesc atunci când vrem să detaliem fiecare acțiune principală din cadrul unui use-case de tip Sea Level.**

Fals. Use Case-urile de tip Sea Level ne arata cum userul interactioneaza cu sistemul, interactiunile sunt majore, iar scopul este bine precizat. Use Case-urile de tip Fish Level sunt acele use case-uri care se dau factor comun din use case-urile Sea-Level (cu <<include>>).

**41. Știm că am descoperit toate Use-Case-urile dintr-un sistem atunci când toate cerințele funcționale sunt acoperite de use-case-uri Sea Level, fiecare dintre acestea trebuind să fie conectate cu cel puțin un actor.**

Adevarat. Use case-urile de tip Sea Level prin definitie reprezinta interacciuni majore ale user-ilor cu sistemul cu un scop precis. Astfel daca toate cerintele funktionale sunt acoperite de Use Case-uri de tip Sea Level am descoperit toate Use Case-urile din

sistem. Si da, trebuie ca fiecare use case sea level sa fie legat la un user pentru ca altfel el nu are sens, un use case presupune existenta cel putin a unui user care sa-l foloseasca.

42. În tehnica CRC clasele sunt identificate pornind de la substantivele din descrierea use-case-urilor, iar responsabilitatea se identifică dintre verbele folosite în cadrul descrierii scenariilor.

Adevarat. In tehnica CRC clasele sunt identificate pornind de la substantive, intrucat programarea orientata pe obiecte cere ca si clasele sa poata defini entitati ale sistemului, deci ele nu pot fi decat substantive, iar responsabilitatile claselor reprezinta functionalitatea lor redată prin metode, iar aceasta functionalitate este data de verbele din scenariul unei diagrame use case.

43. Relația de compoziție este un caz special de agregare (relație parte-întreg) în care indicăm ca distrugerea clasei “întreg”, atrage după sine și distrugerea obiectelor “parte” conținute de către clasa “întreg”.

Adevarat. Relatia de compositie este o forma de agregare in care componentelete nu pot exista una fara cealalta.

44. Între specificatorul de acces “protected” și criteriul de modularitate al protecției există o legătură strânsă.

Fals. Specificatorul de acces „protected” face ca atributele si metodele unei clase sa fie vazute doar de clasele care mostenesc clasa respectiva. Criteriul de modularitate al protectiei ne spune ca daca apare o eroare de executie, aceasta se limiteaza la doar cateva module. Criteriul se poate asigura si cu specificatorul private, cu pachete etc.

45. În stilul arhitectural Repository diferentele componente care procesează date sunt total independente unele de altele, cu excepția faptului că folosesc același model de date.

Adevarat. Stilul arhitectural Repository este un mod eficient de a distribui mari cantitati de date in care producatorii si consumatorii de date sunt independenti, dar marele compromis este faptul ca folosesc acelasi model de date.

46. Principalul motiv pentru care testarea de integrare de tip Big Bang nu este recomandată este acela ca spre deosebire alte tehnici de testare de integrare aceasta identifică mai puține defecte.

Fals. Testarea de integrare de tip Big Bang combina toate componentelete in avans si testeaza intreg programul. In acest fel se creeaza un haos cu multe erori care la prima vedere nu se leaga. Corectarea acestora se realizeaza greu, dupa aceasta rezulta alte erori si astfel testarea pare sa intre intr-o buclu infinita.

47. Testarea ciclurilor pentru o funcție cu două cicluri imbricate(nested) implică scrierea a două cazuri de test; unul să parcurgă instrucțiunile din ciclul interior, și un al doilea care să testeze instrucțiunile din ciclul exterior.

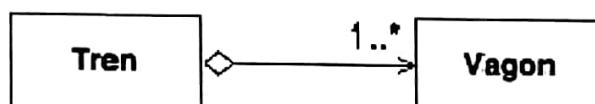
Adevarat. Testarea functiilor cu 2 cicluri imbricate incepe cu testarea buclei din interior care se face ca testarea unei bucle simple pastrand iteratorul buclei exterioare la valoarea minima. Dupa aceasta se trece la testarea celei de a doua bucle in maniera testarii unei bucle simple.

49. Precizati ce stil arhitectural s-ar asocia cel mai bine urmatoarele imagini:

- O ceapa - Layered Architecture intrucat aceasta are o structura stratificata, iar in mijloc se afla lastarul care este cea mai importanta parte a acesteia.
- O linie de productie - conducte si filtre. Conductele reprezinta benzile rulante care ajuta produsele sa se deplaseze prin fabrica, iarfiltrele sunt reprezentate de diferite masinarii care transforma simplele materii prime in produsul finit, pas cu pas.
- Un mediu de dezvoltare integrat (IDE), gen Eclipse, Visual Studio, JBuilder sau un instrument CASE complex - Repository deoarece un mediu de dezvoltare are mai multe unele: compilator, debugger etc, care sunt independente intre ele si care creeaza si folosesc acelasi tip de date.

#### Fundamente de inginerie software

1. Cum se reprezinta intr-o diagrama UML de clasa relatia intre o clasa "Vagon" si o clasa "Tren" considerand ca un tren este compus din unul sau mai multe vagoane si ca un obiect "Vagon" poate fi folosit in relatie cu diferite obiecte "Tren". Precizati cum se numeste acest tip de relatie descris si care au fost indiciile care v-au ajutat sa il identificati.



Raspuns:

Relatia descrisa se numeste **agregare**, si se reprezinta in UML ca in figura de mai jos:  
Relatia intre "Vagon" si "Tren" este in mod clar o relatie de tip "HAS-A" (intreg-partea), adica un obiect "Tren" este compus din obiecte "Vagon". Astfel, am oscilat intre o relatie de agregare si una de componitie. Indiciul din enunt care a facut diferenta este acela ca obiectul "Vagon" pot fi folosite in relatie cu diferite obiecte "Tren", ceea ce inseamna ca viata obiectelor "Vagon" este distincta de cea a obiectelor "Tren". Astfel, relatia este una de agregare.

2. Daca intr-un sistem ar trebui sa favorizati *performanta de timp*, ati opta pentru stilul arhitectural *Layered* (arhitectura stratificata)? Dar daca ar trebui sa favorizati *securitatea*? Justificati succint raspunsul.

Raspuns:

Daca trebuie favorizata performanta de timp, stilul architectural Layered, NU este recomandat. Motivul este acela ca apelarea unui serviciu la nivelul cel mai din exterior (accesibil clientului aplicatiei) implica cel mai adesea o dubla parcurgere a tuturor nivelurilor / straturilor, adica atat pentru transmiterea serviciului, cat si pentru receptarea rezultatului/rezultatelor.

Daca insa trebuie favorizata securitatea stilul architectural Layered, este extrem de indicat! Motivul este acela ca fiecare layer poate oferi un nivel distinct de securitate, ce poate fi

verificat atunci cand acel nivel este accesat. In plus, faptul ca fiecare nivel comunica doar cu nivelul imediat inferior (ca si client) respectiv cu nivelul imediat superior (ca si server) face ca incapsularea datelor, si deci securitatea lor sa fie mai buna.

3. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *La testare blackbox avem nevoie si de cod pentru a verifica daca datele de test acopera toate caile din program?*

**Nota:** raspunsurile “adevarat/fals” neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este fundamental FALSA. Motivul este acela ca testarea blackbox, prin insasi definitia sa, nu implica cunoasterea codului. In testarea blackbox, cazurile de test se scriu strict pe baza “contractului” modului testat, adica a (specificatiilor ?) datelor de intrare, respectiv a celor de iesire.

4. Care este diferența intre urmatoarele tipuri de relatii intre clase: *asociere, agregare, compozitie?*

*Raspuns:*

**Asocierea** reprezinta forma cea mai slaba (si mai generala) in care putem exprima relatia dintre doua clase. Atunci cand spunem ca o relatie este de “asociere” tot ce stim este ca intre cele doua clase exista o relatie.

Atunci cand afirmam ca o relatie intre doua clase este una de **agregare** sau **compozitia**, inseamna ca din descriere cerintelor putem spune ca intre cele doua clase implicate exista o relatie de tip “HAS-A”, o relatie de tip “parte-intreg” (*containment*). Mai departe, distinctia intre agregare si compozitie este de data masura in care obiectele “parte” pot fi *reutilizate* de diferite obiecte “intreg”: daca obiectele parte sunt create si folosite exclusiv de catre un singur obiect, atunci relatia este una de **compozitie**; in caz contrar, daca obiectele “parte” pot fi folosite *shared* de catre mai multe obiecte “intreg” atunci relatia este una de **agregare**. Pe scurt: compozitia este o forma de relatie mai stransa decat agregarea,

5. Enuntati legea lui Brooks, referitoare la extinderea echipei de dezvoltatori in timpul proiectului. De asemenea precizati, argumentand succint, valoarea de adevar a urmatoarei afirmatii: *Intr-un sistem cu o modularitate foarte buna legea lui Brooks nu se aplica pentru ca activitatea de construire a sistemului este o activitate perfect partitionabila.* (**Nota:** precizarea valorii de adevar a afirmatiei, neinsotita de argumentatie nu se puncteaza)

*Raspuns:*

Legea lui Brooks afirma ca adaugarea de programatori (ingineri software) la un proiect aflat in intarziere, va face ca proiectul sa intarzie si mai mult. Motivul este acela, ca un realizarea unui proiect software nu este o activitate perfect partitionabila, ci dimpotriva una ce implica foarte multe interactiuni. Iar adaugarea unor noi programatori ar implica o repartitionare (uneiori imposibil de realizat) a task-urilor din proiect. In plus fiecare om adus in plus, aduce dupa sine relatii de comunicare mai complexe.

In sisteme cu modularitate foarte buna, creste intr-adevar gradul de partitionare, si deci capacitatea de diviziune a task-urilor intre membrii echipei. Totusi, legea lui Brooks ramane valabila, din 2 motive: (i) surplusul de comunicare este semnificativ indiferent de modularitate si (ii) oricat de modular ar fi proiectat un sistem, totusi nu se poate ca sistemul sa devina unul perfect partitionabil.

6. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *Testele whitebox nu pot garanta ca toate erorile dintr-o functie vor fi detectate.*

*dar daca sunt dublate de teste blackbox se poate garanta ca vor fi gasite toate erorile din respectiva functie.*

**Nota:** raspunsurile “adevarat/fals” neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este fundamental FALSA. Nici o forma de testare, si nici o combinatie de tehnici de testare nu pot vreodata garanta absenta bug-urilor. Asa cum spunea Dijkstra, testarea poate evidenta doar prezenta bug-urilor, dar nu poate niciodata garanta absenta lor. Desigur, utilizarea concertata a mai multor tehnici de testare contribuie la reducerea numarului de buguri, dar ceea ce face enuntul de mai sus, complet fals este cuvantul “garanta”.

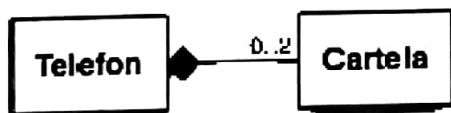
7. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *Testarea de regresie este bazata pe ideea ca atunci cand se opereaza schimbari intr-un modul, imediat dupa modificarile efectuate de testare trebuie concentrate exclusiv asupra modulului modificat.*

**Nota:** raspunsurile “adevarat/fals” neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este FALSA. Ideea de baza in testarea de regresie este reluarea unui subset de teste care au fost rulate anterior. Desigur, aceste teste se refera la modulul modificat, dar in nici un caz **exclusiv** asupra sa. In testarea de regresie, pe langa testele ce vizeaza modulul modificat se mai reiau doua clase de teste: (i) un set de teste reprezentative prin care se retesteaza principalele functionalitati ale sistemului; (ii) un set de teste ce vizeaza modulele direct conectate cu modul modificat, avand in vedere ca impactul schimbarii ar putea sa le fi influentat in primul rand pe acestea.

8. Precizati, si argumentati succint daca relatia descrisa de diagrama UML de mai jos modeleaza sau nu cazul unui telefon dual-sim (care poate tine simultan doua cartele) cu cartele care pot fi oricand reutilizate si in alt telefon.



*Raspuns:*

Relatia descrisa de diagrama UML NU modeleaza intratotul ceea ce sustine enuntul de mai sus! Intr-adevar relatia de mai sus, indica faptul ca un obiect telefon are in compositia sa un numar de cartele ce poate varia intre niciuna si doua cartele... dar partea care nu este bine interpretata se refera la “reutilizarea oricand in alt telefon”. Relatia din figura este una de compositie, marcata prin rombul negru, si deci obiectele “Cartela” sunt construite specific pentru un anumit obiect “Telefon”, putand fi folosite doar de catre acel obiect.

9. Considerand urmatoarele doua procese de dezvoltare: *Waterfall* si respectiv *Extreme Programming*, precizati si argumentati succint pe care l-ati alege daca ati fi *pe rand* in urmatoarele cazuri:

*Cazul 1:* trebuie sa reimplementati de la zero, un sistem complex (cca. 1 milion de linii de cod) pentru gestiunea personalului si a salariilor, care sa inlocuiasca sistemul existent in prezent, fara nici o modificare de cerinte.

*Cazul 2:* aveți de construit un sistem pentru plata taxelor si impozitelor pentru tara imaginara Ainamor cu o legislatie haotica si in permanenta modificare

**Nota:** cele 2 cazuri sunt complet independente.

*Raspuns:*

Pentru “Cazul 1” as alege procesul de dezvoltare Waterfall din urmatoarele motive: (i) e vorba de un sistem de mari dimensiuni, ce implica deci echipe de mari dimensiuni, ceea ce

face ca un proces mai formal sa fie dezirabil; in plus (ii) cerintele sistemului sunt f. bine cunoscute, si clar fixate, deci nu exista riscul de a trebui sa reluam toate fazele procesului (principalul dezavantaj la Waterfall) datorita unei schimbari de cerinte.

Pentru "Cazul 2" as alege in mod evident Extreme Programming datorita cerintelor in permanenta schimbare. Din acest motiv avem nevoie de un proces care sa se poata adapta bine, si mai ales rapid schimbarilor de cerinte, iar Waterfall nu corespunde acestor cerinte. Extreme Programming, are avantajul ca este un proces iterativ si incremental, si in plus are durata iteratiilor scurte.

10. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *Procesul de dezvoltare in cascada (waterfall) este recomandat in majoritatea proiectelor intrucat, datorita structurii sale rigide, poate constrange clientul sa descopere, inca de la inceput, toate posibilele cauze de schimbare din sistem.*

**Nota:** raspunsurile "adevarat/fals" neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este FALSA. Procesul de dezvoltare Waterfall este recomandat in foarte putine cazuri, tocmai datorita structurii sale rigide, nebazata pe iteratii si incremente, ce impiedica o adaptare rapida la schimbari de cerinte. Cu atat mai putin este recomandat procesul Waterfall atunci cand vine vorba de un sistem unde cerintele nu sunt bine intelese. Structura rigida a lui Waterfall NU poate sub nici o forma "constrange" (determina) clientul sa descopere de la inceput toate posibilele cauza de schimbare... Aceasta este o utopie. Chiar si in sisteme unde cerintele sunt clar specificate de la inceput, pot interveni in timp schimbari ce nu pot fi sub nici o forma prevazute. De aceea, trebuie favorizate procesele de dezvoltare (iterative si incrementale) care pot face fata schimbarilor neasteptate.

1. 50% din timpul si costurile totale pentru un proiect ar trebui alocat testarii, dar in multe cazuri in realitate, procentajul este mai mic.

Raspuns: Afirmatia este falsa. Conform regulilor lui Brooks, testarea ocupa intr-adevar 50% atat dpdv al timpului cat si al costului, iar in realitate, acest lucru este respectat. Prin testare se descopera eventuale erori in program, iar testarea insuficienta ar putea avea consecinte dezastruoase datorita anumitor erori nedepistate.

2. Forma reala a curbei ratei defectarii SW-ului este datorata faptului ca cerintele nu sunt intelese clar de la bun inceput, ceea ce duce la necesitatea de schimbari continue asupra sistemului.

Raspuns: Afirmatia este falsa. Schimbarile continue asupra sistemului constituie intr-adevar principalul motiv pentru care curba are acea forma, insa acestea pot aparea chiar daca cerintele au fost intelese perfect initial. De exemplu, s-ar putea ca pe parcurs una sau mai multe cerinte sa se schimbe, ceea ce evident inseamna ca va trebui ca sistemul sa fie modicat corespunzator.

3. Conceptul de time unboxing spune ca sarcinile alocate pt o anumita iteratie trebuie terminate in cadrul iteratiei in cauza

Raspuns: Afirmatia este falsa. In cazul in care sarcinile asignate unei anumite iteratii nu sunt finalizate la timp, ceea ce a ramas va fi inclus in iteratia urmatoare. Intervalul de timp pentru iteratii trebuie sa ramana intotdeauna fix, deoarece altfel echipa de programatori ar avea dificultati in a-si da seama de motivul pentru care nu a reusit sa finalizeze tot ce si-a propus in cadrul iteratiei in cauza.

4. In procesele de dezvoltare agile nu este necesar sa intelegem de la bun inceput cerintele sistemului.

Raspuns: Afirmatia este falsa. Chiar daca metodele agile se caracterizeaza prin adaptabilitatea rapida la modificari si prin faptul ca schimbarile sunt mereu bine-venite, acest lucru nu inseamna ca nu trebuie sa intelegem de la bun inceput ceea ce se cere. Daca pe parcurs programatorii si-ar da seama ca anumite cerinte au fost intelese gresit, acest lucru ar necesita modificarile care ar constitui o risipa de bani si de timp.

5. Daca avem la dispozitie diagrame UML foarte explicite pt un sistem, putem deduce diagramele de secventa pt acel sistem, adica toate interactiunile posibile dintre obiecte

Raspuns: Afirmatia este falsa. Diagramele UML de clasa descriu sistemul dpdv structural - prezinta clasele de obiecte impreuna cu atributiile si operatiile lor, precum si relatiile existente intre obiecte. Diagramele de secventa, pe de alta parte, descriu comportamentul sistemului. Aceste diagrame nu se pot deduce decat cu ajutorul codului sursa, iar acesta nu apare pe diagrama de clasa.

6. Un sistem nu poate fi actor pentru un alt sistem pentru ca acest lucru ar implica accesul la informatiile din sistemul mare, ceea ce incalca incapsularea.

Raspuns: Afirmatia este falsa. Actorul este o entitate care interactioneaza cu sistemul, iar aceasta entitate poate fi orice. Un om (actor) care doreste sa retraga o suma de bani

de la un bancomat (sistem) nu stie nimic despre detaliiile de implementare ale aparatului. Prin urmare, acest lucru se aplica si in cazul in care actorul nostru este chiar un alt sistem.

**7. Pentru a evita proliferarea claselor, clasa Family trebuie implementata ca (in cursul 4, slide-ul 4, varianta din dreapta.)**

Raspuns: Afirmatia este falsa. Modul in care trebuie implementate clasele depinde de ceea ce ne cere sistemul. De exemplu, daca o functionalitate ceruta de sistem ar fi "schimbaScutece()", atunci am opta pentru cea de-a doua varianta, deoarece aceasta operatie ar putea fi realizata de orice persoana din familie (mama, tata, frate, etc.). Daca insa sistemul are functionalitatea "naste()", cum mama este singura care poate face asta, varianta potrivita ar fi, evident, cea din stanga.

**8. Folosirea constantelor globale incalca criteriul de modularitate al continuitatii.**

Raspuns: Afirmatia este adevarata. Daca dorim sa dam variabilei globale alta valoare, acest lucru ar inseamna ca ar trebui sa modificam valoarea peste tot pe unde aceasta apare, ceea ce ar putea contraveni criteriului de continuitate in cazul in care variabila noastra ar aparea in multe locuri.

**9. Stilul arhitectural pipes and filters este avantajos dpdv al timpului**

Raspuns: Afirmatia este fundamental falsa. Trecerea prin fiecare filtru necesita parsarea si analizarea intregului flux de date, ceea ce este consumator de timp.

**10. Un program care nu are cicluri (for, while) are numarul ciclomatic=1**

Raspuns: Afirmatia este falsa. Numarul ciclomatic ne arata numarul de "cai" de executie posibile pentru un anumit program. Ca exemplu vom lua o bucată de cod constant dintr-o secvență if-then-else. Acest program nu are cicluri for sau while, insă există două posibilități de executie: ramura de if, respectivă cea de else, adică numarul ciclomatic =2.

**11. Principalul dezavantaj al testarii top down integration este crearea de drivere si stub-uri**

Raspuns: Afirmatia este parțial adevarata. Top-down integration presupune testarea sistemului de sus in jos, incepand cu modulul principal, si coborand in ierarhie spre modulele de nivel inferior. Stuburile sunt niste inlocuitori pentru modulele care nu au fost inca testate sau implementate, iar crearea lor, necesara pentru acest tip de testare, este consumatoare de timp, deci constituie un dezavantaj. Driverele sunt niste programe care preiau datele de intrare pentru un test, le proceseaza, si tiparesc rezultatele testului, insă ele trebuie create indiferent de procesul de testare ales, deci nu spune ca acestea constituie un dezavantaj specific testarii top-down integration.

FALS                  Drept de rosu de la scrierile mele

de tip  $\frac{1}{2}$

1. **Forma reala a curbei ratei defectarii software-ului este datorata faptului ca cerintele nu sunt intelese clar de la bun inceput, ceea ce duce la necesitatea de schimbari continue asupra sistemului.**

Fals. Curba reala difera de cea ideală din cauza apariției schimbărilor repetitive în soft. Schimbările apar atunci când se efectuează lucrări de menținere asupra softului sau modificări asupra softului.

2. **50% din timp și costuri ar trebui alocate testării, dar în multe cazuri procentajul alocat testării este mai mic.**

Fals. Testarea reprezintă într-adevar jumătate din costuri și din timp, dar aceste estimări nu sunt fictive, deoarece chiar atât reprezintă, trebuie să facem multă testare pe lângă partea de cod pentru a ne asigura că produsul software este bun și face ce trebuie.

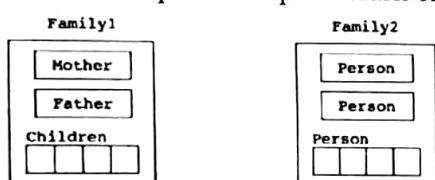
3. **Conceptul de time boxing spune că sarcinile alocate pentru o anumita iteratie trebuie terminate în cadrul iteratiei în cauză.**

Adevarat. Conceptul de time boxing aloca o perioadă fixă fiecărei iterării, numită time box. În cadrul acestei perioade de timp sarcinile alocate în cadrul iterării trebuie terminate. În cazul în care nu se termină task-urile alocate deadline-ul nu se modifică, mai degrabă se amână niste task-uri pentru time box-ul viitor.

4. **Dacă lucram cu procese de dezvoltare agile, asta înseamnă că nu trebuie să intelegem clar cerințele sistemului.**

Fals, cerințele sistemului trebuie intelese indiferent de procesele de dezvoltare, dar în cazul proceselor de dezvoltare agile acestea se pot modifica de la o iteratie la alta sau se mai pot adăuga alte cerinte de la o iteratie la alta, important este că acestea să nu se schimbe în timpul unei iterării.

5. **Pentru a evita problema proliferării claselor, ar trebui să implementăm clasa Family ca mai jos.**



Adevarat. Pentru a evita proliferarea claselor trebuie să fim atenți la comportamentul acestora. El este cel care decide: dacă comportamentul difera avem clase diferite, dacă nu difera avem roluri diferite ale acelasi clase.

6. **Dacă avem diagrame de clasa UML explicite pentru un sistem, putem deduce diagramele de secvență pentru acel sistem, adică toate interacțiunile posibile dintre obiecte.**

Fals. Diagramalele de clasa ne arată structura sistemului, atributele și metodele fiecărei clase, pe când diagramalele de secvență modelează scenariile dintr-un sistem. Acestea nu se pot deduce unele din altele.

**7. Un sistem nu poate fi actor pentru un alt sistem pentru ca acest lucru ar implica accesul la informatiile din sistemul mare, ceea ce incalca incapsularea.**

Fals, un sistem poate fi actor pentru un alt sistem, intrucat acesta interacționează cu interfața acestuia și nu are acces la codul acestuia, astfel nu se incalca principiul incapsularii.

**8. Stilul architectural *Pipes and Filters* este avantajos din punct de vedere al timpului.**

Fals. Aceasta este format din filtre care sunt independente unele față de celelalte și care prelucră datele și din conducte care sunt cai prin care datele circulă de la filtru la filtru. Filtrele nu au un format comun de date, iar din aceasta cauza fiecare filtru trebuie să facă "parse" și "un-parse" datelor, lucru care dă overhead și, implicit, la pierderea timpului.

**9. Un program care nu are cicluri (for, while) are numarul ciclomatic=1.**

Fals. Numarul ciclomatic este influențat de numărul deciziilor simple din cadrul unui program. Instrucțiunile de călcare de tip for/while contin și ele o astfel de instrucțiune, asadar numărul ciclomatic în cazul în care nu avem instrucțiuni de călcare în program este influențat doar de numărul deciziilor simple din cadrul acestuia.

**10. Principalul dezavantaj al testării top down integration este crearea de drivere și stub-uri .**

Fals. În cazul testării de tip top-down se creează doar stub-uri, care imită comportamentul metodelor apelate din funcția testată. Driver-ele sunt create la testarea de tip bottom-up și reprezintă un program principal simplu din care se apelează funcția testată.

**11. Folosirea constantei globale incalca criteriul de modularitate și continuitate.**

~~Fals. Folosirea constantei globale nu incalca principiul continuitatii.~~

**12.Curba reală a evoluției în timp a ratei de defecte (Failure Rate) diferă față de cea ideală în principal din cauza lipsei de experiență a programatorilor.**

Fals, deoarece curba reală diferește de cea ideală din cauza apariției schimbărilor repetitive în soft. Schimbările apar atunci când se efectuează lucrări de menenanță asupra softului. Lipsa de experiență a programatorilor nu este un motiv principal de apariție a erorilor în soft.

**13. Legile evoluției software-ului ale lui Lehman nu se aplică la sisteme ale căror cerințe sunt perfect înțelese de la bun început, pentru că altfel de sisteme nu au nici un motiv să evolueze.**

Fals, deoarece legile evoluției software se aplică oricărui sistem indiferent dacă cerințele sunt înțelese sau nu de la bun început. Sistemele software oricum evoluează datorită schimbărilor inconjurătoare.

14. În procesul de dezvoltare Scrum, Burndown Char ne arată evoluția efortului de-a lungul întregii istorii a proiectului, mai exact oferă informații despre Sprint-urile deja încheiate, precum și numărul de Task-uri finalizate până în prezent în întregul proiect.

Fals. Burndown Chart ne arată evoluția efortului și numărul de task-uri finalizate și de finalizat de-a lungul unui Sprint, perioada în care se realizează un Increment dintr-un proiect. Burndown Chart-ul nu arată evoluția efortului pentru întregul proiect.

15. O problemă importantă a proceselor de dezvoltare iterative este aceea că adesea trebuie modificat codul care a fost scris într-o iteracție anterioară, și uneori anumite porțiuni de cod trebuie chiar șterse, ceea ce reprezintă o pierdere/risipă.

~~Fals - este una hinc decât să avem soft prost~~  
Corect. Aceasta este o problema importantă a proceselor de dezvoltare iterative și este o risipă, în schimb, putem să refacem codul decât să pierdem timpul modificând același cod. Singurul cod care se pastrează este codul funcțional care face ceva util în cadrul softului.

16. În diagramele de UML de Use-Case relațiile <<extends>> și <<include>> sunt foarte asemănătoare și pot fi folosite interschimbabil încărcăt ambele sunt folosite pentru a da "factor comun" descrierea unei anumite funcționalități.

Fals. Relațiile <<extends>> și <<include>> sunt ca și cum se poate de diferențe. Prima reprezintă un caz exceptionál al unui use case, pe când cea de-a două factorizează un comportament comun al mai multor use case-uri.

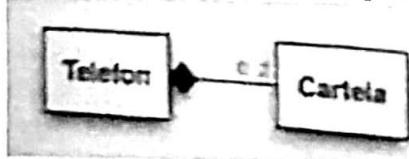
17. Diagramele UML de secvență (sequence diagrams) sunt folosite ca punct de plecare pentru sesiunile de CRC Cards în urma cărora se construiesc diagramele UML de clase (class diagrams).

Fals. Punctul de plecare pentru a crea un CRC card sunt diagramele use case care redau un scenariu identificând actorii și interacțiunile lor cu sistemul. În urma creării cardurilor CRC se realizează diagramele de clasa și apoi diagramele de secvență.

18. Într-o diagramă de secvență UML, toate obiectele care sunt instanțe ale aceleiași clase trebuie "comasate" într-un singur dreptunghi; adică nu este admis sau recomandat să reprezentăm fiecare obiect ca un dreptunghi separat, pentru că altfel s-ar încărca prea mult diagrama.

Fals. Toate instantele unei clase trebuie desenate în dreptunghiuri separate, întrucât fiecare obiect poate avea un comportament diferit chiar dacă sunt instante ale aceleiasi clase.

19. Relația din figura de mai jos modelează cazul unui telefon dual-sim (care poate ține simultan două cartele) cu cartele care pot fi oricând reutilizate și în alt telefon.



*ASOCIARE => reutilizare  
COMPOUNERE => nu reutilizare*

~~Adevarat~~. Avem clasa Telefon și clasa Cartela cu o relație de compunere între ele. Într-adevar, un obiect Cartela nu poate exista fără un obiect Telefon, dar putem avea instante ale clasei Cartela și în alte obiecte Telefon.

20. Un sistem care respectă criteriul de modularitate al Decompozabilității îl va respecta aproape sigur și pe cel al Compozabilității întrucât acesta din urmă se referă la posibilitatea compunerii unui sistem din module.

Fals. Dacă un sistem respectă criteriul de modularizare al decompozabilității nu înseamnă că îl va respecta și pe cel al compozabilității. Exemplu cu IKEA și LEGO.

21. Un avantaj principal al stilului arhitectural Stratificat (Layered Architecture) este asigurarea criteriului de modularitate al Protecției.

Adevarat. În cazul stilului arhitectural stratificat când se modifică interfața unui layer sau se adaugă noi facilități este afectat doar layer-ul adjacente, nu întregul sistem.

22. Ideea de bază în testarea whitebox este aceea de a ne asigura că cel puțin privită individual o funcție nu are nici un bug și nici o instrucțiune, întrucât toate instrucțiunile sunt verificate cu câte cel puțin un test.

Adevarat. Testarea de tip whitebox verifică toate instrucțiunile dintr-o funcție cu cel puțin un test. Astfel, funcția cel puțin privită individual nu are niciun bug.

23. Partițiile echivalente se folosesc la sisteme cu foarte multe funcții lungi și complexe, și reprezintă grupuri de funcții care sunt asemănătoare din punct de vedere al testării. Astfel, din fiecare grup (partiție) se vor alege una sau mai multe funcții pentru care se vor scrie suite de teste blackbox.

Fals. Partițiile echivalente se folosesc la sistemele cu domeniul datelor de intrare foarte mare și reprezintă clase de input-uri asemănătoare din punct de vedere al testării. Astfel, pentru fiecare partitie se aleg cazuri de teste pentru valorile corespunzătoare marginilor și mijlocului.

**24. Legile evoluției software-ului enunțate de către Lehman ne spun că nu există nici o modalitate de a încetini declinul calității software-ului.**

Fals. Legile scaderii calitatii a lui Lehman spun ca putem incetini procesul de deteriorare al software-ului daca il adaptam in mod continuu la schimbarile inconjuratoare.

**25. Printr-o proiectare riguroasă, implementarea software-ului poate fi transformată într-o activitate aproape perfect partaționabilă.**

Fals, o proiectare riguroasa ajuta intr-adevar la partitionarea implementarii softului, dar nu garanteaza o partitionare perfecta. Asta depinde de natura produsul, daca prin natura lui raportat la tehnologiile existente, el poate fi sau nu partitionat cat mai bine.

**26. Procesul de dezvoltare Extreme Programming spune că majoritatea efortului trebuie investit în programare și mai puțin în alte activități conexe cum ar fi captarea cerințelor sau testare, întrucât până la urmă produsul software propriu-zis este creat prin programare.**

Fals, Extreme Programming nu se refera la programarea in sensul de a scrie cod si atat. Este o metodologie bazata pe Agile care abordeaza extrem dezvoltarea iterativa. Este scrisa o noua versiune foarte des, incrementii sunt livrati clientului la fiecare doua trei saptamani, iar constructia e aprobată doar daca testele sunt trecute cu succes.

**27. Deși în procesele de dezvoltare iterative și incrementate trebuie uneori să schimbă sau chiar să rescrisce complet fragmente semnificative de cod ce au fost scrise în iterările anterioare, acestea nu reprezintă o pierdere întrucât schimbările sunt inevitabile.**

~~True~~  
Fals. Aceasta este o problema importanta a proceselor de dezvoltare iterative si este o risipa, in schimb, putem sa refacem codul decat sa pierdem timpul modificand acelasi cod. Singurul cod care se pastreaza este codul functional care face ceva util in cadrul softului.

**28. Tehnica de analiza cerințelor folosind Use-Case-uri este specifică procesului de dezvoltare Waterfall pentru că aici cerințele trebuie analizate riguros la începutul proiectului.**

Fals. Intr-adevar, procesul de tip Waterfall se foloseste atunci cand cerintele sunt foarte bine cunoscute de la inceput, dar asta nu inseamna ca folosirea use case-urilor este specifica doar acestui tip de proces.

**29. Doi sau mai mulți actori nu pot fi asociați (adică nu pot interacționa) cu același use-case pentru că aceasta ar însemna că sunt redundanți.**

Fals. Doi actori pot avea acelasi use case printre altele atata timp cat interactionarea lor cu sistemul este diferita in ansamblu. (Nu interactioneaza cu sistemul pentru aceeasi functionalitate a sistemului).

**30. Într-o diagramă de secvență se recomandă ca dacă apar mai multe instanțe ale aceleiași clase acestea să fie reprezentate într-un singur element pentru a nu se încărca excesiv diagrama.**

Fals. Toate instantele unei clase trebuie desenate în dreptunghiuri separate, intrucât fiecare obiect poate avea un comportament diferit chiar dacă sunt instante ale aceleiasi clase.

**31. Pentru a evita problema proliferării claselor prin modelare ca și clase a entităților externe sistemului trebuie să ținem cont că sunt clase doar acele entități care apelează alte entități (clase) din sistem.**

Fals. Clasele sunt aceleia care sunt apelate (primesc mesaje), nu cele care apelează (transmit mesaje).

**32. Un corp de mobilă modular care vine împachetat pe bucăți și care trebuie să îl asamblăm/compunem (gen IKEA) a fost proiectat urmărind în special criteriul de modularitate al compozabilității.**

Fals. Un corp de mobila modular de la Ikea a fost proiectat în astă fel încât să se poată realiza din acele piese numai un singur corp și numai într-un anumit fel. Acest lucru este în contradicție cu principiul modular al compozibilității care spune că putem compune un sistem modular în mod liber și cum ne dorim.

**33. Importanța criteriilor și regulilor de modularitate este cu atât mai mare cu cât anvergura sistemului software proiectat este mai mare, cu alte cuvinte: importanța modularizării este proporțională cu dimensiunea sistemului software.**

Adevarat. Într-un soft mic aproape că nu avem nevoie de modularizare pentru că nu avem în ce bucati să spargem. Pe când într-un soft mare e important să spargem pe bucati pentru structura frumoasă și pentru a realizare o comunicare între componente ușor de urmarit și componente ușor de reutilizat.

**34. Dacă dorim să verificăm cât mai timpuriu principalele puncte de control și decizie din sistem vom alege testarea de integrare de tip Bottom-Up.**

Fals. Testarea de integrare Bottom-Up verifică timpuriu procesarea de date de nivel scăzut. Pentru a verifica că mai timpuriu principalele puncte de control și decizie din sistem vom alege testarea de integrare de tip Top-Down.

**35. Valoarea complexității ciclomatrice a unei funcții nu este influențată de numărul de instrucțiuni de ciclare (for/while) întrucât singurele instrucțiuni care incrementează valoarea complexității ciclomatrice sunt cele de decizie de tip if-else.**

Fals. Valoarea complexității ciclomatrice este influențată atât de deciziile de tip if/else, cât și de numărul de instrucțiuni de ciclare (for/while), deoarece și aceste instrucțiuni contin câte o decizie simplă.

**36. Curba defectelor poate fi făcută să tindă în timp spre zero dacă cerințele sunt bine înțelese de la început, și dacă se aplică sistematic metode de testare eficiente.**

Fals. Chiar daca sunt bine intelese cerintele de la inceput si se aplica metode de testare eficiente, in timp, din cauza schimbarilor repeatate tot vor aparea erori in sistem, deci curba defectelor nu va tinde in timp spre 0.

**37. Utilizarea eficientă a instrumentelor CASE dedicate programării poate reduce semnificativ costurile totale ale proiectului având în vedere că într-un proiect software o parte semnificativă a costurilor sunt legate de activitatea de codare.**

Fals. Utilizarea eficienta CASE reduce costurile, dar ele se folosesc nu doar pentru partea de codare, ci si pentru alte etape ale procesului precum specificatii, design, testare, debugging. De-asemenea, o mare parte din costuri nu se duc pe partea de codare, ci pe partea de testare si planificare.

**38. În procesul de dezvoltare Waterfall se face doar un singur tip de activitate la un moment dat, spre deosebire de cele iterative unde într-o iteratie trebuie efectuate toate tipurile de activități.**

Adevarat. Procesul de dezvoltare Waterfall imparte proiectul in mai multe activitati ce sunt rezolvate pe rand. De exemplu, se realizeaza analiza si definirea cerintelor, dupa ce se incheie aceasta faza se realizeaza design-ul de sistem si software si asa mai departe.

**39. În procesul Rational Unified Process (RUP), deși este permisă efectuarea mai multor tipuri de activități în interiorul unei faze, se recomandă ca numărul de activități desfășurate în paralel să fie limitat.**

Fals. Intr-adevar, in cazul procesului Rational Unified Process putem efectua mai multe tipuri de activitati simultan in interiorul unei faze, dar nu exista nicio limitare cu privire la numarul de procese desfasurate in paralel. De verificat

**40. Use-Case-urile de tip Fish Level se folosesc atunci când vrem să detaliem fiecare acțiune principală din cadrul unui use-case de tip Sea Level.**

Fals. Use Case-urile de tip Sea Level ne arata cum userul interactioneaza cu sistemul, interactiunile sunt majore, iar scopul este bine precizat. Use Case-urile de tip Fish Level sunt acele use case-uri care se dau factor comun din use case-urile Sea-Level (cu <<include>>).

**41. Știm că am descoperit toate Use-Case-urile dintr-un sistem atunci când toate cerințele funcționale sunt acoperite de use-case-uri Sea Level, fiecare dintre acestea trebuind să fie conectate cu cel puțin un actor.**

Adevarat. Use case-urile de tip Sea Level prin definitie reprezinta interacciuni majore ale user-ilor cu sistemul cu un scop precis. Astfel daca toate cerintele funktionale sunt acoperite de Use Case-uri de tip Sea Level am descoperit toate Use Case-urile din

sistem. Si da, trebuie ca fiecare use case sea level sa fie legat la un user pentru ca altfel el nu are sens, un use case presupune existenta cel putin a unui user care sa-l foloseasca.

42. În tehnica CRC clasele sunt identificate pornind de la substantivele din descrierea use-case-urilor, iar responsabilitatea se identifică dintre verbele folosite în cadrul descrierii scenariilor.

Adevarat. In tehnica CRC clasele sunt identificate pornind de la substantive, intrucat programarea orientata pe obiecte cere ca si clasele sa poata defini entitati ale sistemului, deci ele nu pot fi decat substantive, iar responsabilitatile claselor reprezinta functionalitatea lor redată prin metode, iar aceasta functionalitate este data de verbele din scenariul unei diagrame use case.

43. Relația de compoziție este un caz special de agregare (relație parte-întreg) în care indicăm ca distrugerea clasei “întreg”, atrage după sine și distrugerea obiectelor “parte” conținute de către clasa “întreg”.

Adevarat. Relatia de compositie este o forma de agregare in care componentelete nu pot exista una fara cealalta.

44. Între specificatorul de acces “protected” și criteriul de modularitate al protecției există o legătură strânsă.

Fals. Specificatorul de acces „protected” face ca atributele si metodele unei clase sa fie vazute doar de clasele care mostenesc clasa respectiva. Criteriul de modularitate al protectiei ne spune ca daca apare o eroare de executie, aceasta se limiteaza la doar cateva module. Criteriul se poate asigura si cu specificatorul private, cu pachete etc.

45. În stilul arhitectural Repository diferentele componente care procesează date sunt total independente unele de altele, cu excepția faptului că folosesc același model de date.

Adevarat. Stilul arhitectural Repository este un mod eficient de a distribui mari cantitati de date in care producatorii si consumatorii de date sunt independenti, dar marele compromis este faptul ca folosesc acelasi model de date.

46. Principalul motiv pentru care testarea de integrare de tip Big Bang nu este recomandată este acela ca spre deosebire alte tehnici de testare de integrare aceasta identifică mai puține defecte.

Fals. Testarea de integrare de tip Big Bang combina toate componentelete in avans si testeaza intreg programul. In acest fel se creeaza un haos cu multe erori care la prima vedere nu se leaga. Corectarea acestora se realizeaza greu, dupa aceasta rezulta alte erori si astfel testarea pare sa intre intr-o buclu infinita.

47. Testarea ciclurilor pentru o funcție cu două cicluri imbricate(nested) implică scrierea a două cazuri de test; unul să parcurgă instrucțiunile din ciclul interior, și un al doilea care să testeze instrucțiunile din ciclul exterior.

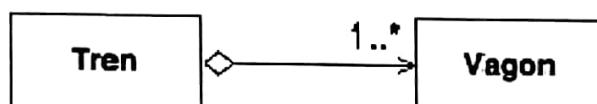
Adevarat. Testarea functiilor cu 2 cicluri imbricate incepe cu testarea buclei din interior care se face ca testarea unei bucle simple pastrand iteratorul buclei exterioare la valoarea minima. Dupa aceasta se trece la testarea celei de a doua bucle in maniera testarii unei bucle simple.

49. Precizati ce stil arhitectural s-ar asocia cel mai bine urmatoarele imagini:

- O ceapa - Layered Architecture intrucat aceasta are o structura stratificata, iar in mijloc se afla lastarul care este cea mai importanta parte a acesteia.
- O linie de productie - conducte si filtre. Conductele reprezinta benzile rulante care ajuta produsele sa se deplaseze prin fabrica, iarfiltrele sunt reprezentate de diferite masinarii care transforma simplele materii prime in produsul finit, pas cu pas.
- Un mediu de dezvoltare integrat (IDE), gen Eclipse, Visual Studio, JBuilder sau un instrument CASE complex - Repository deoarece un mediu de dezvoltare are mai multe unele: compilator, debugger etc, care sunt independente intre ele si care creeaza si folosesc acelasi tip de date.

#### Fundamente de inginerie software

1. Cum se reprezinta intr-o diagrama UML de clasa relatia intre o clasa "Vagon" si o clasa "Tren" considerand ca un tren este compus din unul sau mai multe vagoane si ca un obiect "Vagon" poate fi folosit in relatie cu diferite obiecte "Tren". Precizati cum se numeste acest tip de relatie descris si care au fost indiciile care v-au ajutat sa il identificati.



Raspuns:

Relatia descrisa se numeste **agregare**, si se reprezinta in UML ca in figura de mai jos:  
Relatia intre "Vagon" si "Tren" este in mod clar o relatie de tip "HAS-A" (intreg-partea), adica un obiect "Tren" este compus din obiecte "Vagon". Astfel, am oscilat intre o relatie de agregare si una de componitie. Indiciul din enunt care a facut diferenta este acela ca obiectul "Vagon" pot fi folosite in relatie cu diferite obiecte "Tren", ceea ce inseamna ca viata obiectelor "Vagon" este distincta de cea a obiectelor "Tren". Astfel, relatia este una de agregare.

2. Daca intr-un sistem ar trebui sa favorizati *performanta de timp*, ati opta pentru stilul arhitectural *Layered* (arhitectura stratificata)? Dar daca ar trebui sa favorizati *securitatea*? Justificati succint raspunsul.

Raspuns:

Daca trebuie favorizata performanta de timp, stilul architectural Layered, NU este recomandat. Motivul este acela ca apelarea unui serviciu la nivelul cel mai din exterior (accesibil clientului aplicatiei) implica cel mai adesea o dubla parcurgere a tuturor nivelurilor / straturilor, adica atat pentru transmiterea serviciului, cat si pentru receptarea rezultatului/rezultatelor.

Daca insa trebuie favorizata securitatea stilul architectural Layered, este extrem de indicat! Motivul este acela ca fiecare layer poate oferi un nivel distinct de securitate, ce poate fi

verificat atunci cand acel nivel este accesat. In plus, faptul ca fiecare nivel comunica doar cu nivelul imediat inferior (ca si client) respectiv cu nivelul imediat superior (ca si server) face ca incapsualarea datelor, si deci securitatea lor sa fie mai buna.

3. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *La testare blackbox avem nevoie si de cod pentru a verifica daca datele de test acopera toate caile din program?*

**Nota:** raspunsurile “adevarat/fals” neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este fundamental FALSA. Motivul este acela ca testarea blackbox, prin insasi definitia sa, nu implica cunoasterea codului. In testarea blackbox, cazurile de test se scriu strict pe baza “contractului” modului testat, adica a (specificatiilor ?) datelor de intrare, respectiv a celor de iesire.

4. Care este diferența intre urmatoarele tipuri de relatii intre clase: *asociere, agregare, compozitie?*

*Raspuns:*

**Asocierea** reprezinta forma cea mai slaba (si mai generala) in care putem exprima relatia dintre doua clase. Atunci cand spunem ca o relatie este de “asociere” tot ce stim este ca intre cele doua clase exista o relatie.

Atunci cand afirmam ca o relatie intre doua clase este una de **agregare** sau **compozitia**, inseamna ca din descriere cerintelor putem spune ca intre cele doua clase implicate exista o relatie de tip “HAS-A”, o relatie de tip “parte-intreg” (*containment*). Mai departe, distinctia intre agregare si compozitie este de data masura in care obiectele “parte” pot fi *reutilizate* de diferite obiecte “intreg”: daca obiectele parte sunt create si folosite exclusiv de catre un singur obiect, atunci relatia este una de **compozitie**; in caz contrar, daca obiectele “parte” pot fi folosite *shared* de catre mai multe obiecte “intreg” atunci relatia este una de **agregare**. Pe scurt: compozitia este o forma de relatie mai stransa decat agregarea,

5. Enuntati legea lui Brooks, referitoare la extinderea echipei de dezvoltatori in timpul proiectului. De asemenea precizati, argumentand succint, valoarea de adevar a urmatoarei afirmatii: *Intr-un sistem cu o modularitate foarte buna legea lui Brooks nu se aplica pentru ca activitatea de construire a sistemului este o activitate perfect partitionabila.* (**Nota:** precizarea valorii de adevar a afirmatiei, neinsotita de argumentatie nu se puncteaza)

*Raspuns:*

Legea lui Brooks afirma ca adaugarea de programatori (ingineri software) la un proiect aflat in intarziere, va face ca proiectul sa intarzie si mai mult. Motivul este acela, ca un realizarea unui proiect software nu este o activitate perfect partitionabila, ci dimpotriva una ce implica foarte multe interactiuni. Iar adaugarea unor noi programatori ar implica o repartitionare (uneiori imposibil de realizat) a task-urilor din proiect. In plus fiecare om adus in plus, aduce dupa sine relatii de comunicare mai complexe.

In sisteme cu modularitate foarte buna, creste intr-adevar gradul de partitionare, si deci capacitatea de diviziune a task-urilor intre membrii echipei. Totusi, legea lui Brooks ramane valabila, din 2 motive: (i) surplusul de comunicare este semnificativ indiferent de modularitate si (ii) oricat de modular ar fi proiectat un sistem, totusi nu se poate ca sistemul sa devina unul perfect partitionabil.

6. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *Testele whitebox nu pot garanta ca toate erorile dintr-o functie vor fi detectate.*

*dar daca sunt dublate de teste blackbox se poate garanta ca vor fi gasite toate erorile din respectiva functie.*

**Nota:** raspunsurile “adevarat/fals” neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este fundamental FALSA. Nici o forma de testare, si nici o combinatie de tehnici de testare nu pot vreodata garanta absenta bug-urilor. Asa cum spunea Dijkstra, testarea poate evidenta doar prezenta bug-urilor, dar nu poate niciodata garanta absenta lor. Desigur, utilizarea concertata a mai multor tehnici de testare contribuie la reducerea numarului de bug-uri, dar ceea ce face enuntul de mai sus, complet fals este cuvantul “garanta”.

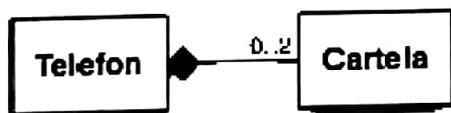
7. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *Testarea de regresie este bazata pe ideea ca atunci cand se opereaza schimbari intr-un modul, imediat dupa modificarile efectuate de testare trebuie concentrate exclusiv asupra modulului modificat.*

**Nota:** raspunsurile “adevarat/fals” neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este FALSA. Ideea de baza in testarea de regresie este reluarea unui subset de teste care au fost rulate anterior. Desigur, aceste teste se refera la modulul modificat, dar in nici un caz **exclusiv** asupra sa. In testarea de regresie, pe langa testele ce vizeaza modulul modificat se mai reiau doua clase de teste: (i) un set de teste reprezentative prin care se retesteaza principalele functionalitati ale sistemului; (ii) un set de teste ce vizeaza modulele direct conectate cu modul modificat, avand in vedere ca impactul schimbarii ar putea sa le fi influentat in primul rand pe acestea.

8. Precizati, si argumentati succint daca relatia descrisa de diagrama UML de mai jos modeleaza sau nu cazul unui telefon dual-sim (care poate tine simultan doua cartele) cu cartele care pot fi oricand reutilizate si in alt telefon.



*Raspuns:*

Relatia descrisa de diagrama UML NU modeleaza intrutotul ceea ce sustine enuntul de mai sus! Intr-adevar relatia de mai sus, indica faptul ca un obiect telefon are in compositia sa un numar de cartele ce poate varia intre niciuna si doua cartele... dar partea care nu este bine interpretata se refera la “reutilizarea oricand in alt telefon”. Relatia din figura este una de compositie, marcata prin rombul negru, si deci obiectele “Cartela” sunt construite specific pentru un anumit obiect “Telefon”, putand fi folosite doar de catre acel obiect.

9. Considerand urmatoarele doua procese de dezvoltare: *Waterfall* si respectiv *Extreme Programming*, precizati si argumentati succint pe care l-ati alege daca ati fi *pe rand* in urmatoarele cazuri:

*Cazul 1:* trebuie sa reimplementati de la zero, un sistem complex (cca. 1 milion de linii de cod) pentru gestiunea personalului si a salariilor, care sa inlocuiasca sistemul existent in prezent, fara nici o modificarile de cerinte.

*Cazul 2:* aveți de construit un sistem pentru plata taxelor si impozitelor pentru tara imaginara Ainamor cu o legislatie haotica si in permanenta modificarile.

**Nota:** cele 2 cazuri sunt complet independente.

*Raspuns:*

Pentru “Cazul 1” as alege procesul de dezvoltare Waterfall din urmatoarele motive: (i) e vorba de un sistem de mari dimensiuni, ce implica deci echipe de mari dimensiuni, ceea ce

face ca un proces mai formal sa fie dezirabil; in plus (ii) cerintele sistemului sunt f. bine cunoscute, si clar fixate, deci nu exista riscul de a trebui sa reluam toate fazele procesului (principalul dezavantaj la Waterfall) datorita unei schimbari de cerinte.

Pentru "Cazul 2" as alege in mod evident Extreme Programming datorita cerintelor in permanenta schimbare. Din acest motiv avem nevoie de un proces care sa se poata adapta bine, si mai ales rapid schimbarilor de cerinte, iar Waterfall nu corespunde acestor cerinte. Extreme Programming, are avantajul ca este un proces iterativ si incremental, si in plus are durata iteratiilor scurte.

10. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *Procesul de dezvoltare in cascada (waterfall) este recomandat in majoritatea proiectelor intrucat, datorita structurii sale rigide, poate constrange clientul sa descopere, inca de la inceput, toate posibilele cauze de schimbare din sistem.*

**Nota:** raspunsurile "adevarat/fals" neinsotite de frazele explicative nu se puncteaza!

*Raspuns:*

Afirmatia este FALSA. Procesul de dezvoltare Waterfall este recomandat in foarte putine cazuri, tocmai datorita structurii sale rigide, nebazata pe iteratii si incremente, ce impiedica o adaptare rapida la schimbari de cerinte. Cu atat mai putin este recomandat procesul Waterfall atunci cand vine vorba de un sistem unde cerintele nu sunt bine intelese. Structura rigida a lui Waterfall NU poate sub nici o forma "constrange" (determina) clientul sa descopere de la inceput toate posibilele cauza de schimbare... Aceasta este o utopie. Chiar si in sisteme unde cerintele sunt clar specificate de la inceput, pot interveni in timp schimbari ce nu pot fi sub nici o forma prevazute. De aceea, trebuie favorizate procesele de dezvoltare (iterative si incrementale) care pot face fata schimbarilor neasteptate.

1. 50% din timpul si costurile totale pentru un proiect ar trebui alocat testarii, dar in multe cazuri in realitate, procentajul este mai mic.

Raspuns: Afirmatia este falsa. Conform regulilor lui Brooks, testarea ocupa intr-adevar 50% atat dpdv al timpului cat si al costului, iar in realitate, acest lucru este respectat. Prin testare se descopera eventuale erori in program, iar testarea insuficienta ar putea avea consecinte dezastruoase datorita anumitor erori nedepistate.

2. Forma reala a curbei ratei defectarii SW-ului este datorata faptului ca cerintele nu sunt intelese clar de la bun inceput, ceea ce duce la necesitatea de schimbari continue asupra sistemului.

Raspuns: Afirmatia este falsa. Schimbarile continue asupra sistemului constituie intr-adevar principalul motiv pentru care curba are acea forma, insa acestea pot aparea chiar daca cerintele au fost intelese perfect initial. De exemplu, s-ar putea ca pe parcurs una sau mai multe cerinte sa se schimbe, ceea ce evident inseamna ca va trebui ca sistemul sa fie modicat corespunzator.

3. Conceptul de time unboxing spune ca sarcinile alocate pt o anumita iteratie trebuie terminate in cadrul iteratiei in cauza

Raspuns: Afirmatia este falsa. In cazul in care sarcinile asignate unei anumite iteratii nu sunt finalizate la timp, ceea ce a ramas va fi inclus in iteratia urmatoare. Intervalul de timp pentru iteratii trebuie sa ramana intotdeauna fix, deoarece altfel echipa de programatori ar avea dificultati in a-si da seama de motivul pentru care nu a reusit sa finalizeze tot ce si-a propus in cadrul iteratiei in cauza.

4. In procesele de dezvoltare agile nu este necesar sa intelegem de la bun inceput cerintele sistemului.

Raspuns: Afirmatia este falsa. Chiar daca metodele agile se caracterizeaza prin adaptabilitatea rapida la modificari si prin faptul ca schimbarile sunt mereu bine-venite, acest lucru nu inseamna ca nu trebuie sa intelegem de la bun inceput ceea ce se cere. Daca pe parcurs programatorii si-ar da seama ca anumite cerinte au fost intelese gresit, acest lucru ar necesita modificarile care ar constitui o risipa de bani si de timp.

5. Daca avem la dispozitie diagrame UML foarte explicite pt un sistem, putem deduce diagramele de secventa pt acel sistem, adica toate interactiunile posibile dintre obiecte

Raspuns: Afirmatia este falsa. Diagramele UML de clasa descriu sistemul dpdv structural - prezinta clasele de obiecte impreuna cu atributiile si operatiile lor, precum si relatiile existente intre obiecte. Diagramele de secventa, pe de alta parte, descriu comportamentul sistemului. Aceste diagrame nu se pot deduce decat cu ajutorul codului sursa, iar acesta nu apare pe diagrama de clasa.

6. Un sistem nu poate fi actor pentru un alt sistem pentru ca acest lucru ar implica accesul la informatiile din sistemul mare, ceea ce incalca incapsularea.

Raspuns: Afirmatia este falsa. Actorul este o entitate care interactioneaza cu sistemul, iar aceasta entitate poate fi orice. Un om (actor) care doreste sa retraga o suma de bani

de la un bancomat (sistem) nu stie nimic despre detaliiile de implementare ale aparatului. Prin urmare, acest lucru se aplica si in cazul in care actorul nostru este chiar un alt sistem.

7. Pentru a evita proliferarea claselor, clasa Family trebuie implementata ca (in cursul 4, slide-ul 4, varianta din dreapta.)

Raspuns: Afirmatia este falsa. Modul in care trebuie implementate clasele depinde de ceea ce ne cere sistemul. De exemplu, daca o functionalitate ceruta de sistem ar fi "schimbaScutece()", atunci am opta pentru cea de-a doua varianta, deoarece aceasta operatie ar putea fi realizata de orice persoana din familie (mama, tata, frate, etc.). Daca insa sistemul are ere functionalitatea "naste()", cum mama este singura care poate face asta, varianta potrivita ar fi, evident, cea din stanga.

8. Folosirea constantelor globale incalca criteriul de modularitate al continuitatii.

Raspuns: Afirmatia este adevarata. Daca dorim sa dam variabilei globale alta valoare, acest lucru ar inseamna ca ar trebui sa modificam valoarea peste tot pe unde aceasta apare, ceea ce ar putea contraveni criteriului de continuitate in cazul in care variabila noastra ar aparea in multe locuri.

9. Stilul arhitectural pipes and filters este avantajos dpdv al timpului

Raspuns: Afirmatia este fundamental falsa. Trecerea prin fiecare filtru necesita parsarea si analizarea intregului flux de date, ceea ce este consumator de timp.

10. Un program care nu are cicluri (for, while) are numarul ciclomatic=1

Raspuns: Afirmatia este falsa. Numarul ciclomatic ne arata numarul de "cai" de executie posibile pentru un anumit program. Ca exemplu vom lua o bucată de cod constând dintr-o secvență if-then-else. Acest program nu are cicluri for sau while, însă există două posibilități de executie: ramura de if, respectiv cea de else, adică numarul ciclomatic =2.

11. Principalul dezavantaj al testarii top down integration este crearea de drivere si stub-uri

Raspuns: Afirmatia este parcial adevarata. Top-down integration presupune testarea sistemului de sus in jos, incepand cu modulul principal, si coborand in ierarhie spre modulele de nivel inferior. Stuburile sunt niste inlocuitori pentru modulele care nu au fost inca testate sau implementate, iar crearea lor, necesara pentru acest tip de testare, este consumatoare de timp, deci constituie un dezavantaj. Driverele sunt niste programe care preiau datele de intrare pentru un test, le proceseaza, si tiparesc rezultatele testului, insa ele trebuie create indiferent de procesul de testare ales, deci nu as spune ca acestea constituie un dezavantaj specific testarii top-down integration.

FALS Dunderde niet negatieke dender  
de tip brotken y

34. Daca dorim sa verificam cat mai timpuriu principalele puncte de control si decizie din sistem vom alege testarea de integrare de tip Bottom-Up.

Fals. Testarea de integrare Bottom-Up verifica timpuriu procesarea de date de nivel scazut. Pentru a verifica cat mai timpuriu principalele puncte de control si decizie din sistem vom alege testarea de integrare de tip Top-Down.

35. Valoarea complexitatii ciclomatice a unei functii nu este influentata de numarul de instructiuni de ciclare(for/while) intrucat singurele instructiuni care incrementeaza valoarea complexitatii ciclomatice sunt cele de divizie de tip if-else.

Fals. Valoarea complexitatii ciclomatice este influentata atat de deciziile if/else, cat si de numarul de instructiuni de ciclare (for/while), deoarece si aceste instructiuni contin cate o decenzie simpla.

36. Curba defectelor poate fi facuta sa tinda in timp spre zero daca cerintele sunt bine intelese de la inceput, si daca se aplica sistematic metode de testare eficiente.

Fals. Chiar daca sunt bine intelese cerintele de la inceput si se aplica metode de testare eficiente, in timp, din cauza schimbarilor repetate tot vor aparea erori in sistem, deci curba defectelor nu va tinde in timp spre 0.

37. Utilizarea eficienta a instrumentelor CASE dedicate programarii poate reduce semnificativ costurile totale ale proiectului avand in vedere ca intr-un proiect software o parte semnificativa a costurilor sunt legate de activitatea de codare.

Fals. Utilizarea CASE reduce costurile, dar ele se folosesc nu doar pentru partea de codare, ci si pentru alte etape ale procesului precum specificatii, design, testare, debugging. De- asemenea, o mare parte din costuri nu se duc pe partea de codare, ci pe partea de testare si planificare.

38. In procesul de dezvoltare Waterfall se face doar un singur tip de activitate la un moment dat, spre deosebire de cele iterative unde intr-o iteratie trebuie efectuate toate tipurile de activitate.

Adevarat. Procesul de dezvoltare Waterfall imparte proiectul in mai multe activitati ce sunt rezolvate pe rand. De exemplu, se realizeaza analiza si definirea cerintelor, dupa ce se incheie aceasta faza se realizeaza design-ul de sistem si software si asa mai departe.

39. In procesul Rational Unified Process (RUP), desi este permisa efectuarea mai multor tipuri de activitati in interiorul unei faze, se recomanda ca numarul de activitati desfasurate in paralele sa fie limitat.

Fals. Intr-adevar, in cazul procesului Rational Unified Process putem efectua mai multe tipuri de activitati simultan in interiorul unei faze, dar nu exista nicio limitare cu privire la numarul de procese desfasurate in paralel.

40. Use Case-urile de tip Fish Level se folosesc atunci cand vrem sa detaliem fiecare actiune principala din cadrul unui use-case de tip Sea Level.

Fals. Use Case-urile de tip Sea Level ne arata cum userul interactioneaza cu sistemul, interactiunile sunt majore, iar scopul este bine precizat. Use Case-urile de tip Fish Level sunt acele use case-uri care se dau factor comun din use case-urile Sea Level(cu <<include>>)

41. Stim ca am descoperit toate Use Case-urile dintr-un sistem atunci cand toate cerintele functionale sunt acoperite de use case-uri Sea Level, fiecare dintre acestea trebuind sa fie conectate cu cel putin un actor.

Adevarat. Use Case-urile de tip Sea Level prin definite reprezinta interacțiuni majore ale utilizatorilor sistemului cu un scop precis. Astfel daca toate cerintele functionale sunt acoperite de Use Case-uri de tip Sea Level am descoperit toate Use Case-urile din sistem. Si da, trebuie ca fiecare use case sea level sa fie legat la un user pentru ca altfel nu are sens, un use case presupune existenta cel putin a unui user care sa-l foloseasca.

42. In tehnica CRC clasele sunt identificate pornind de la substantivale din descrierea use case-urilor, iar responsabilitatea se identifica dintre verbele folosite in cadrul descrierii scenarior.

Adevarat. In tehnica CRC clasele sunt identificate pornind de la substantivale, intrucat programarea orientata pe obiecte cere ca si clasele sa poata defini entitati ale sistemului, deci ele nu pot fi decat substantivale, iar responsabilitatile claselor reprezinta functionalitatea lor redată prin metode, iar aceasta functionalitate este data de verbele din scenariul unei diagrame use case.

43. Relatia de compositie este un caz special de agregare (relatie parte-intreg) in care indicam ca distrugerea clasei "intreg", atrage dupa sine si distrugerea obiectelor "parte" continute de catre clasa "intreg".

Adevarat. Relatia de compositie este o forma de agregare in care componentele nu pot exista una fara cealalta.

44. Intre specificatorul de acces "protected" si criteriul de modularitate al protectiei exista o legatura stransa.

Fals. Specificatorul de acces "protected" face ca atributele si metodele unei clase sa fie vazute doar de clasele care mostenesc clasa respectiva. Criteriul de modularitate al protectiei ne spune ca daca apare o eroare de executie, aceasta se limiteaza la doar cateva module. Criteriul se poate asigura si cu specificatorul private, cu pachete etc.

45. In stilul arhitectural Repository diferentele componente care proceseaza date sunt total independente unele de altele, cu exceptia faptului ca folosesc acelasi model de date.

Adevarat. Stilul arhitectural Repository este un mod eficient de a distribui mari cantitati de date in care producatorii si consumatorii de date sunt independenti, dar marele compromis este faptul ca folosesc acelasi model de date.

46. Principalul motiv pentru care testarea de integrare de tip Big Bang nu este recomandata este acela ca spre deosebire de alte tehnici de testare de integrare aceasta identifica mai putine defecte.

Fals. Testarea de integrare de tip Big Bang combina oate componente in avans si testeaza intreg programul. In acest fel se creeaza un haos cu multe erori care la prima vedere nu se leaga. Corectarea acestora se realizeaza greu, dupa aceasta rezulta alte erori si astfel testarea pare sa intre intr-o bucla infinita.

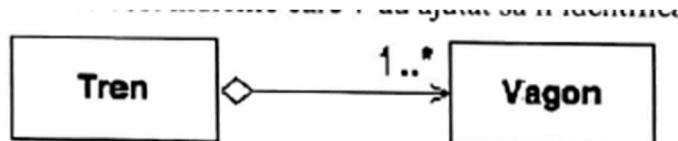
47. Testarea ciclurilor pentru o functie cu doua cicluri imbicate (nested) implica scrierea a doua cazuri de test; unul sa parcurga instructiunile din ciclul interior, si un al doilea care sa testeze instructiunile din ciclul exterior.

Adevarat. Testarea functiilor cu 2 cicluri imbicate incepe cu testarea buclei din interior care face ca testarea unei bucle simple pastrand iteratorul buclei exterioare la valoarea minima. Dupa aceasta se trece la testarea celei de a doua bucle in maniera testarii unei bucle simple.

48. Precizati ce stil arhitectural s-ar asocia cel mai bine urmatoarele imagini:

- O ceapa - Layered Architecture intrucat aceasta are o structura stratificata, iar in mijloc se afla lastarul care este cea mai importanta parte a acesteia
- O linie de productie - conducte si filtre. Conductele reprezinta benzile rulante care ajuta produsele sa se deplaseze prin fabrica, iar filtrele sunt reprezentate de diferite masinarii care transforma simplele materii prime in produsul finit, pas cu pas.
- Un mediu de dezvoltare integrat(IDE), gen Eclipse, Visual Studio, JBuilder sau un instrument CASE complex - Repository, deoarece e un mediu de dezvoltare care are mai multe unelte: compilator, debugger, etc. care sunt independente intre ele si care creeaza si folosesc acelasi tip de date.

49. Cum se reprezinta intr-o diagrama UML de clasa relatia intre o clasa "vagon" si o clasa "tren" considerand ca un tren este compus din unul sau mai multe vagoane si ca un obiect "vagon" poate fi refolosit in relatia cu diferite obiecte "tren". Precizati cum se numeste acest tip de relatie descris si care au fost indiciile care v-au ajutat sa il identificati.



Relatia descisa se numeste agregare, si se reprezinta in UML ca in figura de mai jos: Relatia intre "Vagon" si "tren" este in mod clar o relatia de tip "HAS-A" (intre-parte), adica un obiect "tren" este compus din obiecte "vagon". Astfel, am oscilat intre o relatia de agregare si una de compositie. Indiciul din enunt care a facut diferența este acela ca obiectul "Vagon" poate fi refolosit in relatia cu diferite obiecte "tren", ceea ce inseamna ca viata obiectelor "vagon" este distincta de cea a obiectelor "vagon". Astfel, relatia este una de agregare.

50. Daca intr-un sistem ar trebui sa favorizati performanta de timp, ati opta pentru stilul arhitectural Layered (arhitectura stratificata)? Dar daca ar trebui sa favorizati securitatea?

Daca trebuie favorizata performanta de timp, stilul arhitectural Layered, NU este recomandat. Motivul este acela ca apelarea unui serviciu la nivelul cel mai din exterior implica cel mai adesea o dubla parcurgere a tuturor nivelurilor/straturilor, adica atat pentru transimterea serviciului, cat si pentru receptarea rezultatului/ rezultatelor.

Daca insa trebuie favorizata securitatea, stilul Layered, este extrem de indicat. Motivul este acela ca fiecare layer poate oferi un nivel distinct de securitate, ce poate fi verificat atunci cand acel nivel este accesat. In plus, faptul ca fiecare nivel comunica doar cu nivelul imediat inferior respectiv superior face ca incapsularea datelor, si deci securitatea sa fie mai buna.

51. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint raspunsul: La testare blackbox avem nevoie si de cod pentru a verifica daca datele de test acopera toate calele din program ?

FALS. Motivul este ca acela ca testarea blackbox, prin insasi definitia sa, nu implica cunoasterea codului. In acest black box, cazurile de test se scriu strict pe baza "contractului" modului testat, adica a datelor de intrare, respectiv a celor de iesire.

52. Care este diferența intre urmatoarele tipuri de relatii intre clase: asociere, agregare, componitie ?

Asocierea reprezinta forma cea mai slaba in care putem exprima relatia dintre doua clase. Atunci cand spunem ca o relatie este de "asociere" tot ce stim este ca intre cele doua clase exista o relatie.

Atunci cand afirmam ca o relatie intre doua clase este una de agregare sau componitie, inseamna ca din descrierea cerintelor putem spune ca intre cele doua clase implicate exista o relatie de tip "HAS-A", o relatie de tip "parte-intreg". Mai departe, distinctia intre agregare si componitie este data de masura in care obiectele "parte" pot fi reutilizate de diferite obiecte "intreg": daca obiectele parte sunt create si folosite exclusiv de catre un singur obiect, atunci relatia este una de componitie; in caz contrar, daca obiectele "parte" pot fi folosite shared de catre mai multe obiecte "intreg" atunci relatia este una de agregare. Pe scurt: componitia este o forma de relatie mai stransa decat agregarea.

53. Enuntati legea lui Brooks, referitoare la extinderea echipelor de dezvoltatori in timpul proiectului. De asemenea precizati, argumentand succint, valoarea de adevar a urmatoarei afirmatii: Intr-un sistem cu o modularitate foarte buna legea lui Brooks nu se aplica pentru ca activitatea de construire a sistemului este o activitate perfect partitionabila.

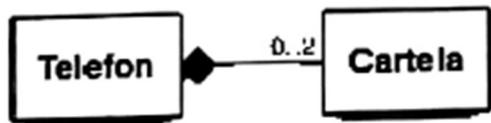
Legea lui Brooks afirma ca adaugarea de programatori la un proiect aflat in intarziere, va face proiectul sa intarzie si mai mult. Motivul este acela, ca in realizarea unui proiect software nu este o activitate perfect partitionabila, ci dimpotriva una ce implica foarte multe interactiuni. Iar adaugarea unor noi programatori ar implica o repartitionare a taskurilor din proiect. In plus fiecare om adus in plus, aduce dupa sine relatii de comunicare mai complexe.

In sisteme cu modularitate foarte buna, creste intr-adefar gradul de partitionare, si deci capacitatea de diviziune a task-urilor intre membrii echipei. Totusi, legea lui Brooks ramane valabila, din 2 motive: 1. Surplusul de comunicare este semnificativ indiferent de modularitate si 2. Oricat de modular ar fi proiectat un sistem, totusi nu se poate ca sistemul sa devina unul perfect partitionabil.

54. Testele whitebox nu pot garanta ca toate erorile dintr-o functie vor fi detectate, dar daca sunt dublate de teste blackbox se poate garanta ca vor fi gasite toate erorile din respectiva functie.

FALS. Ideea de baza in testarea de regresie este reluarea unui subset de teste care au fost rulate anterior. Desigur, aceste teste se refereaza la modulul modificat, dar nici un caz exclusiv asupra sa. In testarea de regresie, pe langa testele ce vizeaza modulul modificat se mai reiau doua clase de teste: 1. Un set de teste reprezentative prin care se retesteaza principalele functionalitati ale sistemului 2. Un set de teste ce vizeaza modulele direct conectate cu modul modificat, avand in vedere ca impactul schimbarii ar putea sa le fie influentat in primul rand pe acestea.

55. Modeleaza sau nu cazul unui telefon dual-sim cu cartele care pot fi reutilizate oricand in alt telefon



FALS. Diagrama UML nu modeleaza intrutotul ceea ce sustine enuntul de mai sus. Intr-adefar relatia de mai sus, indica faptul ca un obiect telefon are in componitia sa un numar de cartele ce poate varia intre niciuna si doua cartele, dar partea care nu este bine interpretata se refera la reutilizare. Relatia din figura este una de compositie, marcată prin rombul negru, deci obiectele "Cartela" sunt construite specific pentru un anumit obiect "Telefon", putand fi folosite doar de catre acel obiect.

56. Waterfall si Extreme Programming, precizati si argumentati succint pe care l-ati alege daca ati fi pe rand in urmatoarele cazuri:

- I. Trebuie sa reimplementati de la zero, un sistem complex pentru gestiunea personalului si a salariilor, care sa inlocuiasca sistemul existent in prezent, daca nici o modificare de cerinte.
- II. Aveti de construit un sistem pentru plata taxelor si impozitelor pentru tara imaginara Ainand cu o legislatie haotica si in permanenta modificare

Pentru I. se alege procesul de dezvoltare Waterfall, deoarece e vorba de un sistem de mari dimensiuni, ce implica deci echipe de mari dimensiuni, ceea ce face ca un proces mai formal sa fie dezirabil; in plus cerintele sunt foarte bine cunoscute deci nu exista riscul sa relua toate fazele procesului.

Pentru II. as alege in mod evident Extreme Programming datorita cerintelor in permanenta schimbare. Din acest motiv avem nevoie de un proces care sa se poata adapta bine, si mai ales rapid schimbarilor de cerinte, iar Waterfall nu corespunde acestor cerinte.

Extreme Programming, are avantajul ca este un proces iterativ si incremental, si in plus are duarata iteratiilor scurte.

57. Procesul de dezvoltare waterfall este recomandat in majoritatea proiectelor intrucat, datorita structurii sale rigide, poate constrange clientul sa descopere, inca de la inceput, toate posibilele cauze de schimbare din sistem.

FALS. Procesul de dezvoltare Waterfall este recomandat in foarte putine cazuri, tocmai datorita structurii sale rigide, nebazata pe iteratii si incremente, ce impiedica o adaptare rapida la schimbari de cerinte. Cu atat mai putin este recomandat procesul Waterfall atunci cand vine vorba de un sistem unde cerintele nu sunt bine intelese.

58. 50% din timpul si costurile totale pentru un proiect ar trebui alocat testarii, dar in multe cazuri in realitate, procentajul este mai mic.

FALS. Conform regulilor lui Brooks, testarea ocupa intr-adevar 50% atat dpdv al timpului cat si al costului, iar in realitate, acest lucru este respectat. Prin testare se descopera eventuale erori in program, iar testarea insuficienta ar putea avea consecinte dezastruoase datorita anumitor erori nedepistate.

59. Forma reala a curbei ratei defectarii SW-ului este datorata faptului ca cerintele nu sunt intelese clar de la bun inceput, ceea ce duce la necesitatea de schimbari continue asupra sistemului.

FALS. Schimbarile continue asupra sistemului constituie intr-adevar principalul motiv pentru care curba are acea forma, insa acestea pot aparea chiar daca cerintele au fost intelese perfect initial. De exemplu, s-ar putea ca pe parcurs una sau mai multe cerinte sa se schimbe, ceea ce evident inseamna ca va trebui ca sistemul sa fie modificat corespunzator.

60. Conceptul de time unboxing spune ca sarcinile alocate pt o anumita iteratie trebuie terminate in cadrul iteratiei in cauza

FALS. In cazul in care sarcinile asignate unei anumite iteratii nu sunt finalizate la timp, ceea ce a rarnas va fi inclus in iteratia urmatoare, Intervalul de timp pentru iteratii trebuie sa ramana intotdeauna fix, deoarece altfel echipa de programatori ar avea dificultati in a si da seama de motivul pentru care nu a reusit sa finalizeze tot ce si-a propus in cadrul iteratiei in cauza.

61. In procesele de dezvoltare agile nu este necesar sa intelegem de la bun inceput cerintele sistemului

FALS. Chiar daca metodele agile se caracterizeaza prin adaptabilitatea rapida la modificari si prin faptul ca schimbarile sunt mereu bine-venite, acest lucru nu inseamna ca nu trebuie sa intelegem de la bun inceput ceea ce se cere. Daca pe parcurs programatori si-ar da seama ca anumite cerinte au fost intelese gresit, acest lucru ar necesita modificari care ar constitui o risipa de bani si de timp.

62. Daca avem la dispozitie diagrame UML foarte explicite pt un sistem, putem deduce diagramele de secventa pt acel sistem, adica toate interactiunile posibile dintre obiecte

FALS. Diagramele UML de clasa descriu sistemul dpdv structural - prezinta clasele de obiecte existente intre obiecte. Diagramele de secventa, pe de alta parte, descriu comportamentul sistemului. impreuna cu atributele si operatiile lor, precum si relatiile Aceste diagrame nu se pot deduce decat cu ajutorul codului sursa, iar acesta nu apare pe diagrama de clasa

63. Un sistem nu poate fi actor pentru un alt sistem pentru ca acest lucru ar implica accesul la informatiile din sistemul mare, ceea ce incalca incapsularea.

FALS. Actorul este o entitate care interactioneaza cu sistemul, iar aceasta entitate poate fi orice, Un om (actor) care doreste sa retraga o suma de bani de la un bancomat (sistem) nu stie nimic despre detaliiile de implementare ale aparaturii. Prin urmare acest lucru se aplica si in cazul in care actorul nostru este chiar un alt sistem.

64. Pentru a evita proliferarea claselor, clasa Family trebuie implementata ca (in cursul 4, slide-ul 4, varianta din dreapta.)

FALS. Modul in care trebuie implementate clasele depinde de ceea ce ne cere sistemul. De exemplu, daca o functionalitate ceruta de sistem ar fi "schimbaScutece". atunci am opta pentru cea de-a doua varianta, deoarece aceasta operatie ar putea fi realizata de orice persoana din familie (mama, tata, frate, etc.). Daca insa sistemul ar cere functionalitatea "naste()", cum mama este singura care poate face asta varianta potrivita ar fi, evident, cea din stanga.

65. Folosirea constantelor globale incalca criteriul de modularitate al continuitatii

ADEVARAT. Daca dorim sa dam variabilei globale alta valoare, acest lucru ar inseamna ca ar trebui sa modificam valoarea peste tot pe unde aceasta apare, ceea ce ar putea contraveni criteriului de continuitate in cazul in care variabila noastra ar aparea in multe locuri

66. Stilul arhitectural pipes and filters este avantajos dpdv al timpului

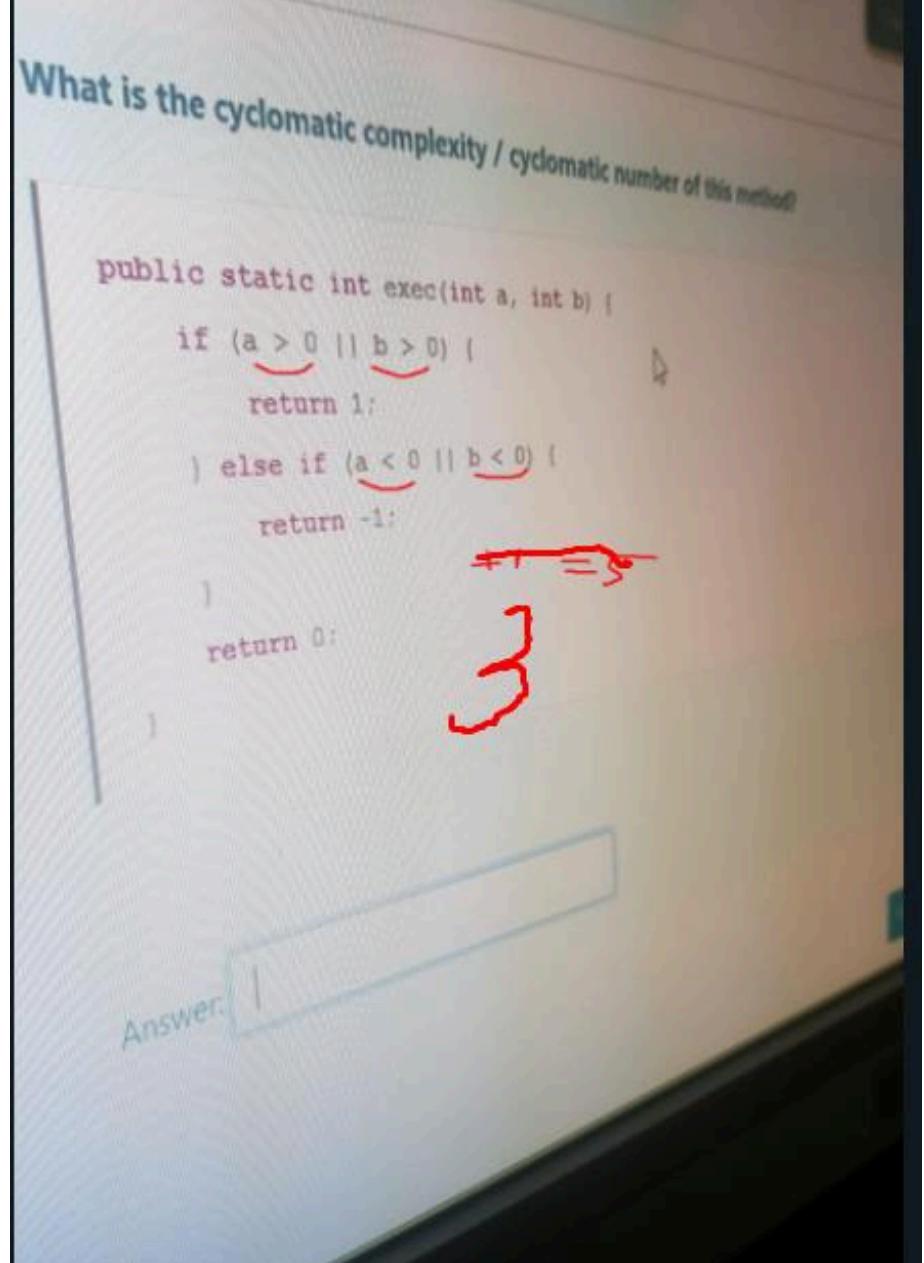
FALS, Trecerea prin fiecare filtru necesita parsarea si analizarea intregului flux de date, ceea ce este consumator de timp

67. Un program care nu are cicluri (for, while) are numarul ciclomatic 1

FALS. Numarul ciclomatic ne arata numarul de "cai" de executie posibile pentru un anumit program. Ca exemplu vom lua o bucată de cod constant dintr-o secvență if- then-else. Acest program nu are cicluri for sau while, insă există două posibilități de execuție: ramura de if, respectiv cea de else, adică numarul ciclomatic =2.

68. Principalul dezavantaj al testarii top down integration este crearea de drivere și stub-uri

FALS. Driverele și stub-urile se folosesc la testarea de tip Bottom-Up.



Question 1

Not yet  
answered

Marked out of  
1.00

Flag question

What is the cyclomatic complexity / cyclomatic number of this method?

```
public static void exec(int a, int b, boolean c, boolean d) {  
    if (c && d) {  
        if (a > 0 && b > 0) {  
            while (a > 0) {  
                System.out.println(a);  
                a--;  
            }  
            while (b > 0) {  
                System.out.println(b);  
                b--;  
            }  
        }  
    }  
}
```

8

6 + 1 →

$$4+1=5$$

**Question 3**

Not yet  
answered

Marked out of  
1.00

Flag question

You need to build a non-distributed system in which input data is transformed into output data by applying a series of well defined data transformations. A global data format exists but the data itself must not be stored globally. The system must be easy to extend when additional transformations must be added and the system components must be easily reusable and combinable one with another when used to build other similar systems. Which of the architectural styles listed below would you chose to satisfy the presented constraints?

Select one or more:

- Blackboard
- Pipes and filters
- Client / server
- Repository
- Layered architecture

**Which of the following statements related to the Throwaway Prototyping are incorrect?**

Select one or more:

- The maintainability of the prototype is favoured by this process ? ✓ F
- It supports better understanding and refinement of requirements A
- The prototype can be easily evolved to a full product ✓ F
- The involved design activities focus on aspects that are visible to the customer A
- It supports early evaluation and feedback from the customer A

← EXAM (Attempt 1) (page 6 of 10) - Opera

cv.upt.ro/mod/quiz/attempt.php

CV

## Software Engineering Fundamentals

Quiz navigation

Question 6 Not yet answered Marked out of 1.00 Flag question

Which of the following statements related to the Rational Unified Process are incorrect?

Select one or more:

- The product is built during four successive phases: Requirements, Analysis and design, Implementation and Testing ✓ F
- In a project phase, several workflows may be active simultaneously A
- A workflow may be active in more than one project phase A
- It is an iterative development process A
- Activities are separated into several workflows such as Inception, Elaboration, Construction and Transition F

Previous page Next page

2021-06-18 10-59-35

00:28:30 00:06:04

Waiting for google-analytics.com... You are screen sharing 10 30 Stop Share

Type here to search ENG 18/06/2021

← EXAM (Attempt 1) (page 5 of 10) - Opera  
cvuptr.o/mod/quiz/attempt.php

CV

## Software Engineering Fundamentals

Quiz navigation

Finish attempt ...  
Time left 0:17:37

Question 5  
Not yet answered  
Marked out of 1.00  
Flag question

Which of the following statements related to the UML relation shown in the attached figure are true?

```
graph LR; B((B)) -- "<-->|<--extend-->" --> A((A))
```

Select one or more:

- It is a relation between two use cases **A**
- It means that the behaviour represented by A is always inserted into the behaviour represented by B **F**
- It means that the behaviour represented by **B** inherits the behaviour represented by A **F**
- It means that the behaviour represented by A is inserted into the behaviour represented by B in some special conditions **A**
- It is a relation between two classes **F**

Previous page Next page

2021-06-18 10-59-35

00:17:26 00:17:08

Type here to search

You are recording 30 Stop Share

11:17 ENG 18/05/2021

49/82

Which of the following statements related to the layered architectural style are correct?

Select one or more:

- It favours the portability of the software product under development A
- In general, a layer is implemented based only on the services provided by the layer below it A
- It has a positive impact on reusability since layers can easily be combined with other layers using pipes in order to build other software products F
- In general, it is allowed for a layer to interact directly with any other layer from the developed software product F
- It may have a negative impact on the performance of the developed software product due to potential increased communication between layers A

← EXAM (Attempt 1) (page 10 of 10) - Opera

cv.upf.ro/mod/quiz/attempt.php

CV

## Software Engineering Fundamentals

Quiz navigation

Question 10

Not yet answered

Marked out of 1.00

Flag question

Time left 0:01:35

Which of the following statements related to the Waterfall development process are correct?

Select one or more:

- It interleaves specification, design, coding, etc. activities F
- It breaks down the project based on development activities ✓
- It breaks down the project by subsets of product functionalities F
- It has separate steps / phases for specification, design, coding, etc. ✓
- It enables a fast development of an initial version of the product F
- It offers the possibility of refining the product with the customer anytime during development F

Previous page

Finish attempt ...

2021-06-18 10:59:35

00:33:29 00:01:05

You are screen sharing Stop Share

Type here to search

11:33 18/06/2021

EXAM (Attempt 1) (page 9 of 10) - Opera  
cv.upt.ro/mod/quiz/attempt.php

Software Engineering Fundamentals

Quiz navigation

Finish attempt ...  
Time left 0:03:32

Question 9  
Answer saved  
Marked out of 1.00  
Flag question

Which of the following statements regarding the use case technique are true?

Select one or more:

Different persons can act as the same actor of a system ✓

Actors are parts of the system F

Actors interact with the system ✓

A person cannot act as more than one actor of a system F

If a person fulfils a combination of roles, each role and each combination of roles represent a separate actor F

A sea-level use case doesn't have to satisfy a goal of an actor F

Previous page      Next page

2021-06-18 10-59-35

00:03:32 00:03:02  
You are screen sharing 10 30 Stop Share  
Type here to search

Question 3

Not yet  
answered

Marked out of  
3.00

Flag question

**Which of the following statements about SCRUM are FALSE?**

Select one or more:

- a. It is suited for the development of large, safety critical software products F
- b. It is an iterative development process A
- c. It can be applied only when the requirements are stable F
- d. It is an agile development process A
- e. It makes use of the timeboxing concept A

[Previous page](#)

Engineering Fundamentals

Question 3  
Answer saved  
Marked out of 3.00  
 Flag question

In the general case, stubs are necessary in order to perform:

Select one or more:

a. Top-down integration testing ✓  
 b. Bottom-up integration testing X  
 c. Unit testing ✓  
 d. Validation testing  
 e. Beta-testing

[Previous page](#)

EXAM (Attempt 1) (page 2 of 10) - Opera  
cv.upf.edu/mod/quiz/attempt.php

CV  
Software Engineering Fundamentals

Quiz navigation

Question 2  
Not yet answered  
Marked out of 1.00  
Flag question

Considering the following Java method, which of the proposed test case suites satisfy exactly the basis path testing strategy?

```
public class SomeClass {  
    public static int someMethod(int x, int y, int z) {  
        if (x > 0) {  
            if (y > 0 || z > 0) {  
                return x + y + z;  
            } else {  
                return x;  
            }  
        }  
        return 0;  
    }  
}
```

Select one or more:

1. x = -6, y = 1, z = 1, ReturnedValue = 0  
2. x = 1, y = 5, z = 9, ReturnedValue = 15  
3. x = 2, y = 0, z = 1, ReturnedValue = 3  
4. x = 17, y = 0, z = -1, ReturnedValue=17

1. x = -6, y = 1, z = 1, ReturnedValue = 0  
2. x = 1, y = 5, z = 9, ReturnedValue = 15

1. x = -6, y = 1, z = 1, ReturnedValue = 0  
2. x = 17, y = 0, z = -1, ReturnedValue = 17

1. x = -6, y = 1, z = 1, ReturnedValue = 0  
2. x = 1, y = 5, z = 9, ReturnedValue = 15

You are screen sharing. Stop Share 11:24 18/06/2021

EXAM (Attempt 1) (page 1 of 10) - Opera  
cv.upLro/mod/quiz/attempt.php

CV

## Software Engineering Fundamentals

Quiz navigation

Finish attempt ...

Time left 0:11:30

Question 1 Not yet answered Marked out of 1.00 Flag question

Which of the following are integration testing approaches?

Select one or more:

- Beta Testing ✓
- Unit Testing ✓
- Top-Up
- Bottom-Up ✓
- Alpha Testing
- Bottom-Down
- Top-Down ✓
- Big-Bang ✓

Koliation Testing

Next page

You are screen sharing Stop Share

Type here to search

11:23 18/06/2021

Exams (Attempt 1) (page 3 of 4) - Google Chrome  
cv.upt.ro/mod/quiz/attempt.php?attempt=155499&cmid=105516&page=2

## Software Engineering Fundamentals

Question 3  
Not yet answered  
Marked out of 3.00  
 Flag question

In the UseCase technique, which of the following statements related to the actor concept are FALSE?

Select one or more:

- a. An actor exchanges information with the software product under analysis A
- b. A primary actor is an external entity with respect to the software product under analysis A
- c. A particular person cannot play the role of different actors F
- d. Two distinct persons can play the role of the same actor A
- e. A secondary actor is an internal entity with respect to the software product under analysis F



Type here to search

F1 F2 F3 F4 F5 F6

Which of the following statements about the top-down integration testing approach are false?

Select one or more:

- Modules are integrated by moving upwards through the control hierarchy starting from the main control module
- It enables checking major control points early in the testing process
- It requires drivers for both built/untested components
- It starts by using the main module as a test driver and stubs are substituted for all components directly subordinate to it
- It is an incremental integration testing approach

# Fundamentals

Question 7

Never saved

Marked out of

Aggregation

According to the Law of Demeter, a method of an object is always allowed to call / invoke methods of:

Select one or more:

- any object returned by a called method
- parameters received by the invoking method ✓
- attributes of the class containing the invoking method
- this / super ✓
- objects created in the invoking method ✓

Previous page



## Theory

Note: If you do not provide an argument when it is required, the answer is not taken into account. The same thing for an incorrect argument, for a generic/vague/unconvincing argument, for an answer containing mistakes etc.

1. (1.5p) For each of the following statements, specify if it is correct or incorrect. After that, in 2-3 phrases, provide an argument for your answer

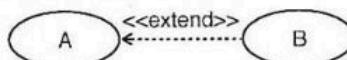
a. You have to develop a student management system for an university. The university is chaotic, with very radical, frequent and fast changes with respect to the student organization policies (e.g., selecting optional disciplines, mark organization, student rankings, etc.). In this context, the best idea is to use a waterfall development process to build the required system.

b. Your SCRUM team has not managed to implement all the tasks committed for the current sprint. According to SCRUM process, you have to allocate some extra time (e.g. one more week) to the current sprint.

---

2. (1.5p) For each of the following statements, specify if it is correct or incorrect. After that, in 2-3 phrases, provide an argument for your answer

a. It is possible for a person (from the real word) to have the role of more than one actor of a system used by that person (Note: the system under discussion has many actors)



b. An <>extend><> relation like in the figure means that the B use case inherits all the interactions from the A use case and, additionally, B adds some supplementary interaction steps to the inherited one

---

3. (1.5p) For each of the following statements, specify if it is correct or incorrect. After that, in 2-3 phrases, provide an argument for your answer

a. The CRC cards model of a system helps us to start building class diagrams and sequence diagrams for that system

b. The modularity criterion named continuity states that we have to continuously search for new requirements for a system due to a software evolution law

---

4. (1.5p) For each of the following statements, specify if it is correct or incorrect. After that, in 2-3 phrases, provide an argument for your answer

a. For the basis / independent path testing approach, we can compute directly (without having to identify all the basis paths) based on the tested function code the exact number of tests that have to be written. If correct give an example, if incorrect give a counterexample.

b. When doing black-box testing on a function having as an input a list of elements, there are some special test cases that have to be designed for that list argument

5. (1.5p) The presented code violates the Law of Demeter

a. Explain where does this flaw occur and why that piece of code is a violation of the Law of Demeter

b. Reimplement the code in order to eliminate the problem (Note: all the classes with their fields must continue to exist as they are now)

```
class Point {  
    private int x,y;  
    public int getX() {return x;}  
    public int getY() {return y;}  
    public void setXY(int a, int b) {x = a; y = b;}  
}  
class Figure {  
    private Point referencePoint;  
    public Figure(Point o) {referencePoint = o;}  
    public Point getPoint() {return referencePoint;}  
}  
class TranslateCommand {  
    public void execute(Figure f, int dx, int dy) {  
        int nx = f.getPoint().getX() + dx;  
        int ny = f.getPoint().getY() + dy;  
        f.getPoint().setXY(nx,ny);  
    }  
}
```

---

6. In a program, MyPersonalList class (the code in the left part) is completely implemented. In such an object we can i) add an Integer at the beginning of the list ii) add an Integer at the end of the list and iii) we can remove and get an element from a given position. On a FIFO object we must be able to only i) add an Integer ii) extract an element in the order in which the elements have been added (like in a queue)

a. (0.5p) Based on a piece of code written by you, explain why using inheritance to implement the FIFO objects is not an adequate solution

b. (1p) Reimplement completely the FIFO class (having the same functionality) without using inheritance but using MyPersonalList objects (Note: it is forbidden to modify MyPersonalList class)

```
class MyPersonalList {  
    public void addAtFirstPosition(Integer i)  
    {...}  
    public void addAtLastPosition(Integer i)  
    {...}  
    public Integer removePosition(int i)  
    {...}}
```

```
class FIFO extends MyPersonalList {  
    public void add(Integer i) {  
        addAtLastPosition(i);  
    }  
    public Integer extract() {  
        return removePosition(0);  
    }  
}
```

## Problems

1. (3p) In the given execution context and starting from the call marked with `/**/`, draw the sequence diagram (with all the details e.g., activation bars, return values, etc.) that contains all the interactions between the objects

```

interface X {
    public int t();
}

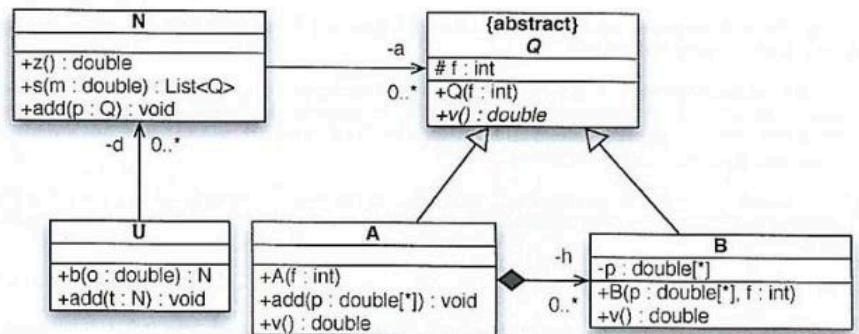
class D implements X {
    private X a, b;
    public D(X a, X b) {
        this.a = a;
        this.b = b;
    }
    public int t() {
        int i1 = a.t();
        int i2 = b.t();
        return i1 + i2;
    }
    public int k() {
        return b.t() + 1;
    }
    public int q(E x) {
        return x.w(this);
    }
}

class E implements X {
    public int t() {
        return 1;
    }
    public int w(X s) {
        return s.t();
    }
}

public class Test {
    public static void main(String[] args) {
        E e1 = new E();
        E e2 = new E();
        D d1 = new D(e1,e2);
        D d2 = new D(d1,e2);
        /*d2.q(e1);
    }
}

```

2. (6p) Implement in Java all the entities (completely) from the following class diagram, based on the diagram and based on the provided explanations.



- the constructors are used to initialize the corresponding fields
- the `add` methods are used to i) add the object referred by the parameter as an associated object or ii) to implement the composition (in these cases, if any other values are required you can consider them as being 1)

### Details for Q:

- the `v` method does not have an implementation

### Details for N:

- the `z` method returns the sum of the values returned by the `v` method when invoked on each object of type `Q` associated to the current object
- the `s` method returns a reference to a new object of type `List` (`ArrayList` or `LinkedList`); this list will contain references to all the objects of type `Q` associated to the current object and for which the value returned by their `v` method is greater than `m` (the method parameter)

### Details for U:

- the `b` method returns a reference to an object of type `N` associated to the current object; for this returned object the `z` method must return a value that is equal to the value of the `o` parameter; if such an object does not exist null is returned

### Details for A:

- the `v` method returns the value of `f` multiplied by the sum of the values returned by the `v` method invoked on each `B` object from the current object

### Details for B:

- the `v` method returns the value of `f` multiplied by the sum of all the elements from `p`

# Software Engineering Fundamentals

## Question 1

Not yet answered

Marked out of 3.00

 Flag question

In the general case, drivers are necessary in order to perform:

Select one or more:

- a. Bottom-up integration testing
- b. Validation testing
- c. Beta-testing
- d. Unit testing
- e. Top-down integration testing

[Next page](#)



Type here to search



acer





# Software Engineering Fundamentals

## Question 3

Not yet answered

Marked out of 3.00

Flag question

In the UseCase technique, which of the following statements related to the actor concept are FALSE?

Select one or more:

- a. An actor exchanges information with the software product under analysis
- b. A primary actor is an external entity with respect to the software product under analysis
- c. A particular person cannot play the role of different actors
- d. Two distinct persons can play the role of the same actor
- e. A secondary actor is an internal entity with respect to the software product under analysis



Type here to search



F1

F2

F3

F4

F5

F6

# Software Engineering Fundamentals

## Question 4

Not yet answered

Marked out of 3.00

 Flag question

Which of the following development process models / processes are NOT iterative or CANNOT be used to build full-fledged software products?

Select one or more:

- a. Agile Methods
- b. RUP
- c. Throwaway Prototyping
- d. Waterfall
- e. Incremental Delivery

[Previous page](#)

[Finish attempt ...](#)

 Type here to search



Question 1

Not yet  
answered

Marked out of  
3.00

\* Flag question

What is the cyclomatic complexity / cyclomatic number of this method?

```
public static void exec(int a, int b, boolean c, boolean d) {
    if (c && d) {
        if (a > 0 && b > 0) {
            while (a > 0) {
                System.out.println(a);
                a--;
            }
            while (b > 0) {
                System.out.println(b);
                b--;
            }
        }
    }
}
```

Answer:

**Question 3**

Not yet  
answered

Marked out of  
3.00

Flag question

**Which of the following statements about SCRUM are FALSE?**

Select one or more:

- a. It is suited for the development of large, safety critical software products
- b. It is an iterative development process
- c. It can be applied only when the requirements are stable
- d. It is an agile development process
- e. It makes use of the timeboxing concept

[Previous page](#)

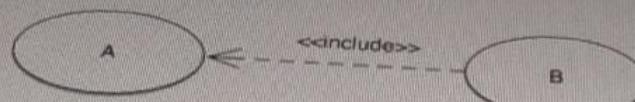
Question 2

Not yet  
answered

Marked out of  
3.00

Flag question

Which of the following statements related to the UML relation shown in the attached figure are TRUE?



Select one or more:

- a. It means that the behaviour represented by A is always inserted into the behaviour represented by B
- b. It means that the behaviour represented by A is inserted into the behaviour represented by B in some special conditions
- c. It means that the behaviour represented by B is always inserted into the behaviour represented by A
- d. It is a relation between two use cases
- e. It is a relation between two classes

Question 3

Answer saved

Marked out of  
3.00

Flag  
question

In the general case, stubs are necessary in order to perform:

Select one or more:

- a. Top-down integration testing
- b. Bottom-up integration testing
- c. Unit testing
- d. Validation testing
- e. Beta-testing

# Software Engineering Fundamentals

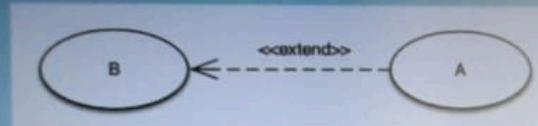
Question 1

Answer saved

Marked out of  
2.00

Flag question

Which of the following statements related to the UML relation shown in the attached figure are TRUE?



Select one or more:

- a. It means that the behaviour represented by A is always inserted into the behaviour represented by B.
- b. It is a relation between two use cases.
- c. It means that the behaviour represented by A inherits the behaviour represented by B.
- d. It means that the behaviour represented by A is inserted into the behaviour represented by B in some special conditions.
- e. It is a relation between two classes.

What is the cyclomatic complexity / cyclomatic number of this method?

```
public static int exec(int a, int b) {  
    if (a > 0 || b > 0) {  
        return 1;  
    } else if (a < 0 || b < 0) {  
        return -1;  
    }  
    return 0;  
}
```

Answer:



# Software Engineering Fundamentals

## Question 1

Not yet answered

Marked out of 3.00

Flag question

In the general case, drivers are necessary in order to perform:

Select one or more:

- a. Bottom-up integration testing ✓
- b. Validation testing
- c. Beta-testing
- d. Unit testing ✓
- e. Top-down integration testing

Next page



Type here to search



acer



# Software Engineering Fundamentals

## Question 3

Not yet answered

Marked out of 3.00

Flag question

In the UseCase technique, which of the following statements related to the actor concept are FALSE?

Select one or more:

- a. An actor exchanges information with the software product under analysis A
- b. A primary actor is an external entity with respect to the software product under analysis A
- c. A particular person cannot play the role of different actors F
- d. Two distinct persons can play the role of the same actor A
- e. A secondary actor is an internal entity with respect to the software product under analysis R

Type here to search



F1

F2

F3

F4

F5

F6

# Software Engineering Fundamentals

## Question 4

Not yet answered

Marked out of 3.00

Flag question

Which of the following development process models / processes are NOT iterative or CANNOT be used to build full-fledged software products?

Select one or more:

- a. Agile Methods
- b. RUP
- c. Throwaway Prototyping ✓
- d. Waterfall ✓
- e. Incremental Delivery

Previous page

Finish attempt ...

Type here to search



Question 1

Not yet  
answered

Marked out of  
3.00

\* Flag question

What is the cyclomatic complexity / cyclomatic number of this method?

```
public static void exec(int a, int b, boolean c, boolean d) {
    if (c && d) {
        if (a > 0 && b > 0) {
            while (a > 0) {
                System.out.println(a);
                a--;
            }
            while (b > 0) {
                System.out.println(b);
                b--;
            }
        }
    }
}
```

Answer:

Question 3

Not yet  
answered

Marked out of  
3.00

Flag question

**Which of the following statements about SCRUM are FALSE?**

Select one or more:

- a. It is suited for the development of large, safety critical software products F
- b. It is an iterative development process A
- c. It can be applied only when the requirements are stable F
- d. It is an agile development process A
- e. It makes use of the timeboxing concept A

[Previous page](#)

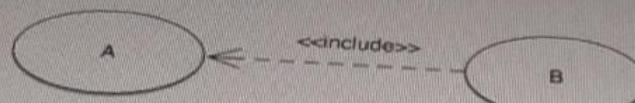
Question 2

Not yet  
answered

Marked out of  
3.00

Flag question

Which of the following statements related to the UML relation shown in the attached figure are TRUE?



Select one or more:

- a. It means that the behaviour represented by A is always inserted into the behaviour represented by B F
- b. It means that the behaviour represented by A is inserted into the behaviour represented by B in some special conditions F
- c. It means that the behaviour represented by B is always inserted into the behaviour represented by A A
- d. It is a relation between two use cases A
- e. It is a relation between two classes F

## Question 3

Answer saved

Marked out of  
3.00

Flag  
question

In the general case, stubs are necessary in order to perform:

Select one or more:

- a. Top-down integration testing VA
- b. Bottom-up integration testing F
- c. Unit testing VA
- d. Validation testing F
- e. Beta-testing F

# Software Engineering Fundamentals

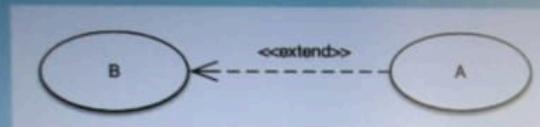
Question 1

Answer saved

Marked out of  
2.00

Flag question

Which of the following statements related to the UML relation shown in the attached figure are TRUE?



Select one or more:

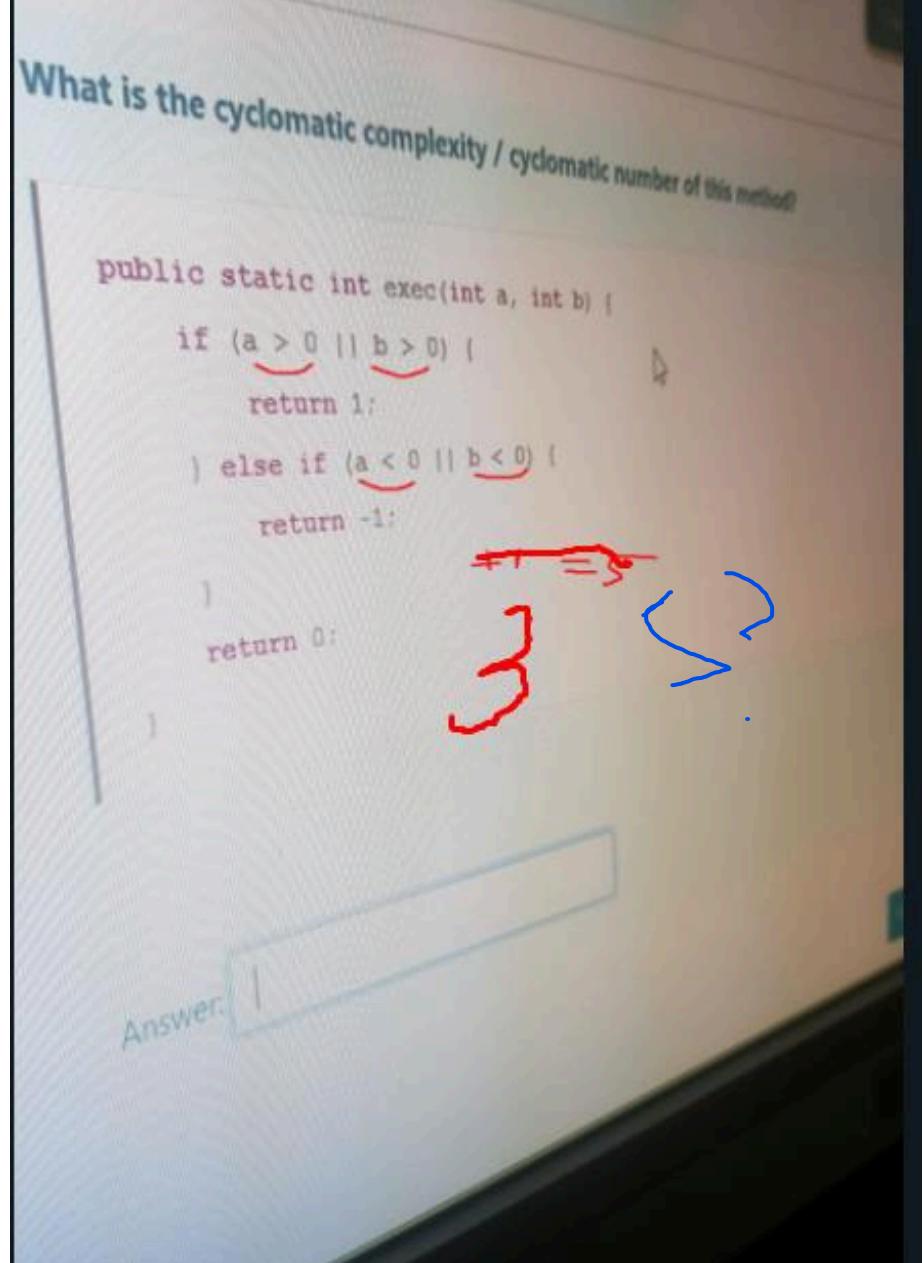
- a. It means that the behaviour represented by A is always inserted into the behaviour represented by B F
- b. It is a relation between two use cases A
- c. It means that the behaviour represented by A inherits the behaviour represented by B F
- d. It means that the behaviour represented by A is inserted into the behaviour represented by B in some special conditions A
- e. It is a relation between two classes F

What is the cyclomatic complexity / cyclomatic number of this method?

```
public static int exec(int a, int b) {  
    if (a > 0 || b > 0) {  
        return 1;  
    } else if (a < 0 || b < 0) {  
        return -1;  
    }  
    return 0;  
}
```

5?

Answer:



Question 1

Not yet  
answered

Marked out of  
1.00

Flag question

What is the cyclomatic complexity / cyclomatic number of this method?

```
public static void exec(int a, int b, boolean c, boolean d) {  
    if (c && d) {  
        if (a > 0 && b > 0) {  
            while (a > 0) {  
                System.out.println(a);  
                a--;  
            }  
            while (b > 0) {  
                System.out.println(b);  
                b--;  
            }  
        }  
    }  
}
```

F

6 + 1 = 7

4 + 1 = 5

**Question 3**

Not yet  
answered

Marked out of  
1.00

Flag question

You need to build a non-distributed system in which input data is transformed into output data by applying a series of well defined data transformations. A global data format exists but the data itself must not be stored globally. The system must be easy to extend when additional transformations must be added and the system components must be easily reusable and combinable one with another when used to build other similar systems. Which of the architectural styles listed below would you chose to satisfy the presented constraints?

Select one or more:

- Blackboard
- Pipes and filters
- Client / server
- Repository
- Layered architecture

**Which of the following statements related to the Throwaway Prototyping are incorrect?**

Select one or more:

- The maintainability of the prototype is favoured by this process ? ✓ F
- It supports better understanding and refinement of requirements A
- The prototype can be easily evolved to a full product ✓ F
- The involved design activities focus on aspects that are visible to the customer A
- It supports early evaluation and feedback from the customer A

← EXAM (Attempt 1) (page 6 of 10) - Opera

cv.upt.ro/mod/quiz/attempt.php

CV

## Software Engineering Fundamentals

Quiz navigation

Question 6 Not yet answered Marked out of 1.00 Flag question

Which of the following statements related to the Rational Unified Process are incorrect?

Select one or more:

- The product is built during four successive phases: Requirements, Analysis and design, Implementation and Testing ✓ F
- In a project phase, several workflows may be active simultaneously A
- A workflow may be active in more than one project phase A
- It is an iterative development process A
- Activities are separated into several workflows such as Inception, Elaboration, Construction and Transition F

Previous page Next page

2021-06-18 10-59-35

00:28:30 00:06:04

Waiting for google-analytics.com... You are screen sharing 10 30 Stop Share

Type here to search ENG 18/06/2021

← EXAM (Attempt 1) (page 5 of 10) - Opera  
cvuptr.o/mod/quiz/attempt.php

CV

## Software Engineering Fundamentals

Quiz navigation

Finish attempt ...  
Time left 0:17:37

Question 5  
Not yet answered  
Marked out of 1.00  
Flag question

Which of the following statements related to the UML relation shown in the attached figure are true?

```
graph LR; B((B)) -- "<-->|<--extend-->" --> A((A))
```

Select one or more:

- It is a relation between two use cases **A**
- It means that the behaviour represented by A is always inserted into the behaviour represented by B **F**
- It means that the behaviour represented by **B** inherits the behaviour represented by A **F**
- It means that the behaviour represented by A is inserted into the behaviour represented by B in some special conditions **A**
- It is a relation between two classes **F**

Previous page Next page

2021-06-18 10-59-35

00:17:26 00:17:08

Type here to search

You are recording 30 Stop Share

11:17 ENG 18/05/2021

49/82

Which of the following statements related to the layered architectural style are correct?

Select one or more:

- It favours the portability of the software product under development A
- In general, a layer is implemented based only on the services provided by the layer below it A
- It has a positive impact on reusability since layers can easily be combined with other layers using pipes in order to build other software products F
- In general, it is allowed for a layer to interact directly with any other layer from the developed software product F
- It may have a negative impact on the performance of the developed software product due to potential increased communication between layers A

← EXAM (Attempt 1) (page 10 of 10) - Opera

cv.upf.ro/mod/quiz/attempt.php

CV

## Software Engineering Fundamentals

Quiz navigation

Question 10

Not yet answered

Marked out of 1.00

Flag question

Time left 0:01:35

Which of the following statements related to the Waterfall development process are correct?

Select one or more:

- It interleaves specification, design, coding, etc. activities F
- It breaks down the project based on development activities ✓
- It breaks down the project by subsets of product functionalities F
- It has separate steps / phases for specification, design, coding, etc. ✓
- It enables a fast development of an initial version of the product F
- It offers the possibility of refining the product with the customer anytime during development F

Previous page

Finish attempt ...

2021-06-18 10:59:35

00:33:29 00:01:05

You are screen sharing Stop Share

Type here to search

11:33 18/06/2021

EXAM (Attempt 1) (page 9 of 10) - Opera  
cv.upt.ro/mod/quiz/attempt.php

CV

## Software Engineering Fundamentals

Quiz navigation

Finish attempt ...  
Time left 0:03:32

Question 9  
Answer saved  
Marked out of 1.00  
Flag question

Which of the following statements regarding the use case technique are true?

Select one or more:

- Different persons can act as the same actor of a system ✓
- Actors are parts of the system F
- Actors interact with the system ✓
- A person cannot act as more than one actor of a system F
- If a person fulfils a combination of roles, each role and each combination of roles represent a separate actor F
- A sea-level use case doesn't have to satisfy a goal of an actor F

Previous page      Next page

2021-06-18 10-59-35

00:03:32 00:03:02  
Type here to search You are screen sharing Stop Share  
10 30 Stop Share  
11:31 ENG 18/06/2021

Question 3

Not yet  
answered

Marked out of  
3.00

Flag question

**Which of the following statements about SCRUM are FALSE?**

Select one or more:

- a. It is suited for the development of large, safety critical software products F
- b. It is an iterative development process A
- c. It can be applied only when the requirements are stable F
- d. It is an agile development process A
- e. It makes use of the timeboxing concept A

[Previous page](#)

Engineering Fundamentals

Question 3  
Answer saved  
Marked out of 3.00  
 Flag question

In the general case, stubs are necessary in order to perform:

Select one or more:

a. Top-down integration testing ✓  
 b. Bottom-up integration testing X  
 c. Unit testing ✓  
 d. Validation testing  
 e. Beta-testing

[Previous page](#)

EXAM (Attempt 1) (page 2 of 10) - Opera  
cv.upf.edu/mod/quiz/attempt.php

CV  
Software Engineering Fundamentals

Quiz navigation

Question 2  
Not yet answered  
Marked out of 1.00  
Flag question

Considering the following Java method, which of the proposed test case suites satisfy exactly the basis path testing strategy?

```
public class SomeClass {  
    public static int someMethod(int x, int y, int z) {  
        if (x > 0) {  
            if (y > 0 || z > 0) {  
                return x + y + z;  
            } else {  
                return x;  
            }  
        }  
        return 0;  
    }  
}
```

Select one or more:

1. x = -6, y = 1, z = 1, ReturnedValue = 0  
2. x = 1, y = 5, z = 9, ReturnedValue = 15  
3. x = 2, y = 0, z = 1, ReturnedValue = 3  
4. x = 17, y = 0, z = -1, ReturnedValue=17

1. x = -6, y = 1, z = 1, ReturnedValue = 0  
2. x = 1, y = 5, z = 9, ReturnedValue = 15

1. x = -6, y = 1, z = 1, ReturnedValue = 0  
2. x = 17, y = 0, z = -1, ReturnedValue = 17

1. x = -6, y = 1, z = 1, ReturnedValue = 0  
2. x = 1, y = 5, z = 9, ReturnedValue = 15

You are screen sharing. Stop Share 11:24 18/06/2021

EXAM (Attempt 1) (page 1 of 10) - Opera  
cv.upLro/mod/quiz/attempt.php

CV

## Software Engineering Fundamentals

Quiz navigation

Finish attempt ...

Time left 0:11:30

Question 1 Not yet answered Marked out of 1.00 Flag question

Which of the following are integration testing approaches?

Select one or more:

- Beta Testing ✓
- Unit Testing ✓
- Top-Up
- Bottom-Up ✓
- Alpha Testing
- Bottom-Down
- Top-Down ✓
- Big-Bang ✓

Koliation Testing

Next page

You are screen sharing Stop Share

Type here to search

11:23 18/06/2021

Exams (Attempt 1) (page 3 of 4) - Google Chrome  
cv.upt.ro/mod/quiz/attempt.php?attempt=155499&cmid=105516&page=2

## Software Engineering Fundamentals

Question 3  
Not yet answered  
Marked out of 3.00  
 Flag question

In the UseCase technique, which of the following statements related to the actor concept are FALSE?

Select one or more:

- a. An actor exchanges information with the software product under analysis A
- b. A primary actor is an external entity with respect to the software product under analysis A
- c. A particular person cannot play the role of different actors F
- d. Two distinct persons can play the role of the same actor A
- e. A secondary actor is an internal entity with respect to the software product under analysis F



Type here to search

F1 F2 F3 F4 F5 F6

Which of the following statements about the top-down integration testing approach are false?

Select one or more:

- Modules are integrated by moving upwards through the control hierarchy starting from the main control module
- It enables checking major control points early in the testing process
- It requires drivers for both built/untested components
- It starts by using the main module as a test driver and stubs are substituted for all components directly subordinate to it
- It is an incremental integration testing approach

# Fundamentals

Question 7

Never saved

Marked out of

Aggregation

According to the Law of Demeter, a method of an object is always allowed to call / invoke methods of:

Select one or more:

- any object returned by a called method
- parameters received by the invoking method ✓
- attributes of the class containing the invoking method
- this / super ✓
- objects created in the invoking method ✓

Previous page



## Problems

1. (3p) In the given execution context and starting from the call marked with `/**/`, draw the sequence diagram (with all the details e.g., activation bars, return values, etc.) that contains all the interactions between the objects

```

interface X {
    public int t();
}

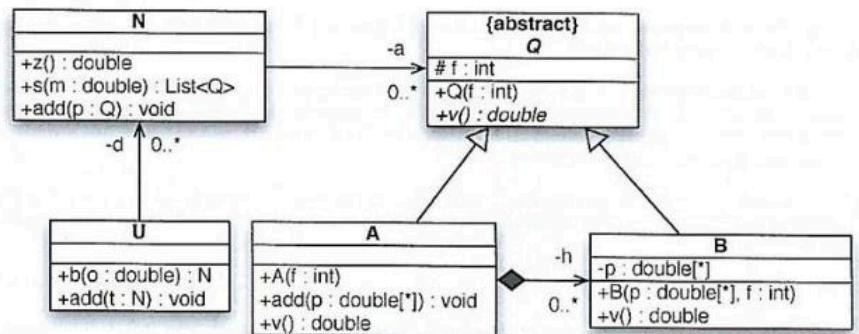
class D implements X {
    private X a, b;
    public D(X a, X b) {
        this.a = a;
        this.b = b;
    }
    public int t() {
        int i1 = a.t();
        int i2 = b.t();
        return i1 + i2;
    }
    public int k() {
        return b.t() + 1;
    }
    public int q(E x) {
        return x.w(this);
    }
}

class E implements X {
    public int t() {
        return 1;
    }
    public int w(X s) {
        return s.t();
    }
}

public class Test {
    public static void main(String[] args) {
        E e1 = new E();
        E e2 = new E();
        D d1 = new D(e1,e2);
        D d2 = new D(d1,e2);
        /*d2.q(e1);
    }
}

```

2. (6p) Implement in Java all the entities (completely) from the following class diagram, based on the diagram and based on the provided explanations.



- the constructors are used to initialize the corresponding fields
- the `add` methods are used to i) add the object referred by the parameter as an associated object or ii) to implement the composition (in these cases, if any other values are required you can consider them as being 1)

### Details for Q:

- the `v` method does not have an implementation

### Details for N:

- the `z` method returns the sum of the values returned by the `v` method when invoked on each object of type `Q` associated to the current object
- the `s` method returns a reference to a new object of type `List` (`ArrayList` or `LinkedList`); this list will contain references to all the objects of type `Q` associated to the current object and for which the value returned by their `v` method is greater than `m` (the method parameter)

### Details for U:

- the `b` method returns a reference to an object of type `N` associated to the current object; for this returned object the `z` method must return a value that is equal to the value of the `o` parameter; if such an object does not exist null is returned

### Details for A:

- the `v` method returns the value of `f` multiplied by the sum of the values returned by the `v` method invoked on each `B` object from the current object

### Details for B:

- the `v` method returns the value of `f` multiplied by the sum of all the elements from `p`

## Theory

Note: If you do not provide an argument when it is required, the answer is not taken into account. The same thing for an incorrect argument, for a generic/vague/unconvincing argument, for an answer containing mistakes etc.

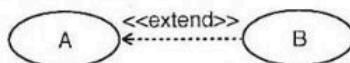
1. (1.5p) For each of the following statements, specify if it is correct or incorrect. After that, in 2-3 phrases, provide an argument for your answer

F a. You have to develop a student management system for an university. The university is chaotic, with very radical, frequent and fast changes with respect to the student organization policies (e.g., selecting optional disciplines, mark organization, student rankings, etc.). In this context, the best idea is to use a waterfall development process to build the required system.

F b. Your SCRUM team has not managed to implement all the tasks committed for the current sprint. According to SCRUM process, you have to allocate some extra time (e.g. one more week) to the current sprint.

2. (1.5p) For each of the following statements, specify if it is correct or incorrect. After that, in 2-3 phrases, provide an argument for your answer

A a. It is possible for a person (from the real word) to have the role of more than one actor of a system used by that person (Note: the system under discussion has many actors)



F only adds

b. An <>extend><> relation like in the figure means that the B use case inherits all the interactions from the A use case and, additionally, B adds some supplementary interaction steps to the inherited one

3. (1.5p) For each of the following statements, specify if it is correct or incorrect. After that, in 2-3 phrases, provide an argument for your answer

A a. The CRC cards model of a system helps us to start building class diagrams and sequence diagrams for that system

? b. The modularity criterion named continuity states that we have to continuously search for new requirements for a system due to a software evolution law

4. (1.5p) For each of the following statements, specify if it is correct or incorrect. After that, in 2-3 phrases, provide an argument for your answer

F a. For the basis / independent path testing approach, we can compute directly (without having to identify all the basis paths) based on the tested function code the exact number of tests that have to be written. If correct give an example, if incorrect give a counterexample.

A b. When doing black-box testing on a function having as an input a list of elements, there are some special test cases that have to be designed for that list argument

5. (1.5p) The presented code violates the Law of Demeter

a. Explain where does this flaw occur and why that piece of code is a violation of the Law of Demeter

b. Reimplement the code in order to eliminate the problem (Note: all the classes with their fields must continue to exist as they are now)

```
translate(int dx,int dy)
{
    refPoint.setXY(refPoint.get(X) +dx
    ,refPoint.get(Y) +dy)
}
```

```
class Point {
    private int x,y;
    public int getX() {return x;}
    public int getY() {return y;}
    public void setXY(int a, int b) {x = a; y = b;}
}

class Figure {
    private Point referencePoint;
    public Figure(Point o) {referencePoint = o;}
    public Point getPoint() {return referencePoint;}
}

class TranslateCommand {
    public void execute(Figure f, int dx, int dy) {
        int nx = f.getPoint().getX() + dx;
        int ny = f.getPoint().getY() + dy;
        f.getPoint().setXY(nx,ny);
    } f.translate(dx,dy)✓
}
```

6. In a program, MyPersonalList class (the code in the left part) is completely implemented. In such an object we can i) add an Integer at the beginning of the list ii) add an Integer at the end of the list and iii) we can remove and get an element from a given position. On a FIFO object we must be able to only i) add an Integer ii) extract an element in the order in which the elements have been added (like in a queue)

a. (0.5p) Based on a piece of code written by you, explain why using inheritance to implement the FIFO objects is not an adequate solution

b. (1p) Reimplement completely the FIFO class (having the same functionality) without using inheritance but using MyPersonalList objects (Note: it is forbidden to modify MyPersonalList class)

```
class MyPersonalList {
    public void addAtFirstPosition(Integer i)
    {...}
    public void addAtLastPosition(Integer i)
    {...}
    public Integer removePosition(int i)
    {...}
}
```

```
class FIFO extends MyPersonalList {
    public void add(Integer i) {
        addAtLastPosition(i);
    }
    public Integer extract() {
        return removePosition(0);
    }
}
```

The screenshot shows a slide from a presentation titled "Software Engineering Fundamentals". The slide has a blue header bar with the title and some navigation icons. Below the header, there's a section titled "Navigate in test" with a grid of small squares. A blue button labeled "Revizuire și creare Raport Recenzie" is visible. To the right, there's a box containing the number "5" and some text in Romanian: "Revizuire și creare Raport Recenzie", "Nu a prezent capitolul final", "Mersul din 1.00", and "Părere în cadrul grupului". At the bottom left, a blue button says "Pagina precedenta". On the right side of the slide, there's a question in English: "Which of the following statements related to the Repository/Blackboard style are correct?". Below the question, there's a list of five statements with checkboxes:

- It may be expensive to migrate to a new data model of the data store
- In the case of the Blackboard variation, a sub-system must query the data store to detect when data of interest changed
- The data store is accessible to all sub-systems from the developed software product
- In the case of the Repository variation, a sub-system is informed by the data store when data of interest changed
- In general, it is allowed for a sub-system to interact directly with any other sub-system from the developed software product

<https://cv.upr.edu/mod/quiz/attempt.php?attempt=693225&cmid=246305&page=9>

## Engineering Fundamentals

3 întrebare

Nu a printat  
răspuns închisMarcat din 1,00  
Iată întrebarea cu  
tag

Timp rămas 0:17:22

Considering the following Java code, which of the presented expressions will be FALSE when evaluated at the position indicated in the code by /\*HERE\*/?

```
class A {}  
class B extends A {}  
class Main {  
    public static void main(String argv[]) {  
        A a = new B();  
        B b = new B();  
        Object o = new A();  
        System.out.println(/*HERE*/);  
    }  
}
```

Selectați unul sau mai multe:

- B.class.isInstance(a)
- o.getClass().equals(Object.class)
- A.class.equals(a.getClass())
- A.class.isInstance(b)

attempt=6932256&amp;mid=2463056&amp;page=1#



**BTC**  
Embedded Systems Romania



1. Întrebare  
Nu a primit  
răspuns încă  
Marcat din 1,00  
Căutare cu  
fag

Considering the following Java method, which of the proposed test case suites satisfy exactly the basis path testing strategy?

```
public class SomeClass {  
    public static int someMethod(int x, int y, int z) {  
        if (x > 0) {  
            if (y > 0 || z > 0) {  
                return x + y + z;  
            } else {  
                return x;  
            }  
        }  
        return 0;  
    }  
}
```

Selectați unul sau mai multe:

- 1. x = -1, y = 10, z = 0, ReturnedValue = 0  
2. x = 1, y = 10, z = 0, ReturnedValue = 11  
3. x = 1, y = 0, z = 10, ReturnedValue = 11  
4. x = 1, y = 0, z = 0, ReturnedValue = 1
- 1. x = -6, y = 1, z = 1, ReturnedValue = 0  
2. x = 1, y = 5, z = 9, ReturnedValue = 15  
3. x = 17, y = 0, z = -1, ReturnedValue = 17
- 1. x = -6, y = 1, z = 1, ReturnedValue = 0  
2. x = 1, y = 5, z = 9, ReturnedValue = 15  
3. x = 2, y = 0, z = 1, ReturnedValue = 3  
4. x = 17, y = 0, z = -1, ReturnedValue = 17
- 1. x = -1, y = 10, z = 0, ReturnedValue = 0  
2. x = 1, y = 10, z = 0, ReturnedValue = 11



**BTC**  
Embedded Systems Platform

