

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.

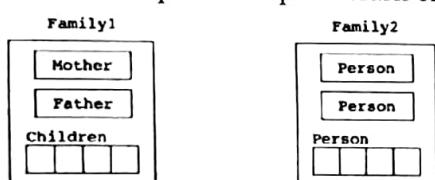
3. **Conceptul de time boxing spune că sarcinile alocate pentru o anumita iteratie trebuie terminate în cadrul iteratiei în cauză.**

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 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 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 imita 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 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.

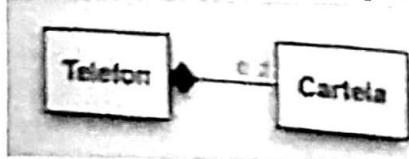
17. Diagramele UML de secvență (sequence diagrams) sunt folosite ca punct de plecare pentru sesiunile de CRC Cards în urma cărora se construiesc diagramele UML de clase (class diagrams).

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.



*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.

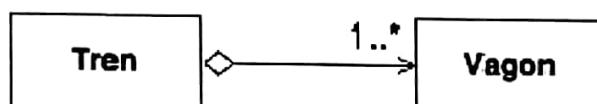
Adevarat. Testarea functiilor cu 2 cicluri imbricate incepe cu testarea buclei din interior care se face ca testarea unei bucle simple pastrand iteratorul buclei exterioare la valoarea minima. Dupa aceasta se trece la testarea celei de a doua bucle in maniera testarii unei bucle simple.

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”.

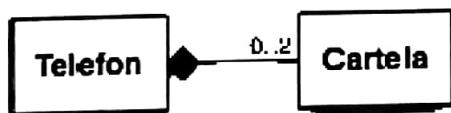
7. Precizati daca urmatoarea afirmatie este adevarata sau falsa, si justificati succint (1-2 fraze) raspunsul: *Testarea de regresie este bazata pe ideea ca atunci cand se opereaza schimbari intr-un modul, imediat dupa modificarile efectuate de testare trebuie concentrate exclusiv asupra modulului modificat.*

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 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 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 în jos, începând cu modulul principal, și coborând în ierarhie spre modulele de nivel inferior. Stuburile sunt niste înlocuitori pentru modulele care nu au fost încă 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 procesează, și tipăresc rezultatele testului, însă ele trebuie create indiferent de procesul de testare ales, deci nu spune că acestea constituie un dezavantaj specific testării top-down integration.

FALS Dreptea nu este deosebită.
 a tipătoris Y