

**Avantaje** - clienții nu trebuie să aștepte până ce  
trez sistemul a fost livrat pentru a beneficia de el.  
Primul increment include cele mai importante cerințe,  
peci sistemul poate fi folosit imediat - primele  
creșterea pot fi prototipuri din care se pot stabili  
cerințele pentru următoarele incremente - se  
micșorează riscul ca proiectul să fie un eșec deoarece  
cerințele cele mai importante sunt livrate la început -  
deoarece cerințele cele mai importante fac parte din

mele incrementale, acestea vor fi testate cel mai mult. – **Probleme** - dificultăți în transformarea cerințelor utilizatorului în incremente de mărime potrivită - procesul nu este foarte vizibil pentru utilizator (nu e suficient documentație între iterații) - codul se poate degrada în decursul ciclurilor.

**Metodologii "agile"** - se concentrează mai mult pe cod decât pe proiectare - se bazează pe o abordare iterativă de dezvoltare de software - produc rapid versiuni care funcționează, acestea evoluând repede pentru a satisface cerințe în schimbare - scopul metodelor agile este de a reduce cheltuielile în procesul de dezvoltare a software-ului (de exemplu, prin limitarea documentației) și de a răspunde rapid cerințelor în schimbare. **Se pune accent pe** - indivizii și interacțiunile înaintea proceselor și uneltelor - software-ul funcțional înaintea documentației vaste - colaborarea cu clientul înaintea negocierii contractuale - receptivitatea la schimbare înaintea urmăririi unui plan. **Principii ale manifestului agil:** 1. Prioritatea noastră este satisfacția clientului prin livrare rapidă și continuă de software valoros. 2. Schimbarea cerințelor este binevenită chiar și într-o fază avansată a dezvoltării. Procesele agile valorifică schimbarea în avantajul competitiv al clientului. 3. Livrarea de software funcțional se face frecvent, de preferință la intervale de timp cât mai mici, de la câteva săptămâni la câteva luni. 4. Clienții și dezvoltatorii trebuie să colaboreze zilnic pe parcursul proiectului. 5. Construiște proiecte în jurul oamenilor motivați. Oferă-le medii propice și suportul necesar și îi încadrează că obiectivele vor fi atinse. 6. Cea mai eficientă metodă de a transmite informații înspre și în interiorul echipei de dezvoltare este comunicarea față în față. 7. Software funcțional este principala măsură a progresului. 8. Procesele agile promovează dezvoltarea durabilă. Sponsorii, dezvoltatorii și utilizatorii trebuie să poată menține un ritm constant pe termen nedefinit. 9. Atenția continuă pentru excelență tehnică și design bun îmbunătățește agilitatea. 10. Simplitatea este esențială. 11. Cele mai bune arhitecturi, cerințe și design se obțin de către echipe care se auto-organizează. 12. La intervale regulate, echipa reflectează la cum s-a devină mai eficientă, apoi își adaptează și ajustează comportamentul în consecință. **Aplicabilitatea metodelor agile** - în companii care dezvoltă produse software de dimensiuni mici sau mijlocii - în cadrul companiilor unde se dezvoltă software pentru uz intern (proprietary software), deoarece există un angajament clar din partea clientului (intern) de a se implica în procesul de dezvoltare și deoarece nu există o mulțime de reguli și reglementări externe care afectează software-ul. **Probleme** - dificultatea de a păstra interesul clienților implicati în acest procesul de dezvoltare pentru perioade lungi - membrii echipei nu sunt întotdeauna potriviți pentru implicarea intensă care caracterizează metodele agile - prioritizarea modificărilor poate fi dificilă atunci când există mai multe părți interesate - menținerea simplității necesită o muncă suplimentară - contractele pot fi o problemă ca și în alte metode de dezvoltare incrementală **Extreme programming** - noile versiuni pot fi construite de mai multe ori pe zi; acestea sunt livrate clienților la fiecare 2 săptămâni; toate testele trebuie să fie executate pentru fiecare versiune și o versiune e livrabilă doar în cazul în care testele au rulat cu succes - **XP și principiile agile** - "dezvoltare incrementală" este susținută prin intermediul livrării de software în mod

frecvent cu mici incremente - "implicarea clientului" înseamnă angajamentul "full-time" al clientului cu echipa de dezvoltare - "oameni, nu procese" prin programare pereche, proprietatea colectivă și un proces care să evite orele lungi de lucru - "receptivitate la schimbare" prin livrări frecvente - "menținerea simplității" prin refactoring constant de cod. **Planificare: livrări:** Clientul înțelege domeniul de aplicare, prioritățile, nevoile business ale versiunilor care trebuie livrate: sortează "cartonașele" cu sarcini după priorități; **iteratii:** Dezvoltatorii estimează riscurile și eforturile: sortează "cartonașele" după risc dacă o sarcină la mai mult de 2-4 săptămâni, e distribuită pe mai multe "cartonașe" **Metafora** - = arhitectura sistemului - se evita cuvântul "arhitectura pentru a sublinia faptul ca nu avem de-a face cu o structura generala **Integrare continua:** atunci când dezvoltatorii au terminat o parte din implementare: o integrează cu codul existent - rulează teste și corectează eventualele probleme - dacă toate testele sunt pozitive, adaugă modificările în sistemul care se ocupa cu managementul codului sursă **Proiectare simpla:** "proiectează cel mai simplu lucru care funcționează acum. Nu proiecta și pentru mâine, pentru că s- ar putea să nu fie nevoie" **"test-driven development":** se scriu teste înaintea codului pentru a clarifica cerințele - testele sunt scrise ca programe în loc de date, astfel încât acestea să poată fi executate automat - fiecare testul include o condiție de corectitudine - toate testele anterioare și cele noi sunt rulate automat atunci când sunt adăugate noi funcționalități, verificând astfel că noua funcționalitate nu a introdus erori. **îmbunătățirea codului:** îmbunătățirea codului prin "refactoring" este foarte importantă deoarece XP recomandă începerea implementării foarte repede ex: "three strikes and you refactor" **programarea în echipe de 2** **AVANTAJE** - soluție bună pentru proiecte mici - programare organizată - reducerea numărului de greșeli - clientul are control (de fapt, toată lumea are control, pentru că toți sunt implicați în mod direct) - dispoziție la schimbare chiar în cursul dezvoltării **DEZAVANTAJE** - nu este scalabilă - necesită mai multe resurse umane "pe linie de cod"(d.ex. programare în doi) - implicarea clientului în dezvoltare (costuri suplimentare și schimbări prea multe) - lipsa documentelor "oficiale" - necesită experiență în domeniu ("senior level" developers) - poate deveni uneori o metoda inefficientă (rescriere masivă de cod) **SCRUM** - metoda agile axata pe managementul dezvoltarii incrementale **Pasi** - un proprietar de produs creează o listă de sarcini numită "backlog" - apoi se planifică ce sarcini vor fi implementate în următoarea iterație, numită "sprint" - această listă de sarcini se numește "sprint backlog" - sarcinile sunt rezolvate în decursul unui sprint care are rezervată o perioadă relativ scurtă de 2-4 săptămâni - echipa se întrunește zilnic pentru a discuta progresul ("daily scrum"). Ceremoniile sunt conduse de un "scrum master" - la sfârșitul sprintului, rezultatul ar trebui să fie livrabil (adică folosit de client sau vandabil). nă după o analiză a sprintului, se reiterează.

Metode agile	Metode cascada	Metode formale
criticalitate scăzută	criticalitate ridicată	criticalitate extremă
dezvoltatori seniori	dezvoltatori juniori	dezvoltatori seniori
cerințe în schimbare	cerințe relativ fixe	cerințe limitate
echipe mici	echipe mari	echipe mici
cultură orientată spre schimbare	cultură orientată spre ordine	cultură orientată spre calitate și precizie

**Model-checking** = metoda de verificare bazata pe modele, automata, verifica proprietati, folosita mai mult pentru sisteme concurente, reactive, folosita initial in post-dezvoltare. Prin contrast, **verificarea programelor** este bazata pe demonstratii, asistata de calculator (necesita interventia omului), folosita mai mult pentru programe care se termina si produc un rezultat. **Structurile Kripke** - introduc posibilitatea mai multor universuri (locale) - exista o relatie de accesibilitate între aceste universuri si operatori care le conecteaza permitand exprimarea diverselor tipuri de consecințe - dacă ceea ce produce trecerea de la un univers la altul este timpul, atunci logicile rezultate se numesc logici temporale. Programele se potrivesc foarte bine in aceasta filozofie - un univers corespunde unei stari - relatia ed accesibilitate este data de tranzitia de la o stare la alta datorata efectuării instructiunilor - logica predicativa clasica se foloseste pentru a specifica relatii între valorile dintr-o stare a variabilelor din program - ce lipseste este un mecanism care sa conecteze universurile stărilor între ele -> folosim CTL **Timpul** în logicile temporale poate fi - linear sau ramificat - discret sau continuu. **CTL foloseste timp ramificat si discret.** Vrem sa raspundem la intrebarea M, s | - phi?, unde - M este un model al sistemului analizat sub forma Kripke si s este o stare a modelului - phi este o formula CTL care vrem sa fie satisfacuta de sistem **CTL:** - **conectori temporali:** AX, EX, AU, EU, AG, EG, AF, EF - **A si E - cuantificare in latime** - A = se iau toate alternative din punctul de ramificare - E = exista cel puțin o alternativa din punctul de ramificare - **G si F - cuantifica de-a lungul ramurilor** - G = toate stările viitoare de pe drum - F = exista cel puțin o stare viitoare pe drum - **X** = starea urmatoare de pe drum - **U** = until - **Prioritati:** 1. AX EX AG EG AF EF 2. Si, sau 3. Implica, AU, EU - **Viitorul contine prezentul** (din orice stare este posibil să revenim într-o stare dată **restart:** AG(EF restart) un lift care se deplasează în sus la etajul 2 nu-și schimbă direcția dacă pasagerii merg la etajul 5:  $AG(floor = 2 \wedge direction = up \wedge ButtonPressed5 \rightarrow A[direction = up \vee floor = 5])$  un lift poate rămâne inactiv la etajul 3 cu ușile închise  $AG(floor = 3 \wedge idle \wedge door = closed \rightarrow EG(floor = 3 \wedge idle \wedge door = closed))$  EF finish = este posibil să se ajungă într-o stare în care finish = true AF AG stable = în orice execuție, la un moment dat, stable este invariant  $AG( req \rightarrow A(req \vee U grant))$  = întotdeauna, un req rămâne activ până se obține un grant

este posibil să ajungem într-o stare unde un proces a început (started), dar nu este încă gata (ready):  $EF(started \wedge \neg ready)$  pentru orice stare, dacă a apărut o cerere (request), atunci ea va fi ulterior confirmată (acknowledged):  $AG(request \rightarrow AF acknowledged)$  un proces este disponibil (enabled) de o infinitate de ori pe orice drum:  $AG(AF enabled)$  orice s-ar întâmpla, procesul va fi permanent blocat (deadlocked):  $AF(AG deadlocked)$  **LTL** - o formula LTL este evaluata pe un drum, ori pe o multime de drumuri; de aceea cuantificarile din CTL "exists" si "any" dispar aici - putem, insa, amesteca operatori modali intr-un mod care nu este posibil in CTL. O formula LTL phi este satisfacuta in starea s a unui model M daca phi este satisfacuta in toate drumurile care incep cu s. **Testare - Verificare** - construim corect produsul? - se referă la dezvoltarea produsului; **Validare** - construim produsul corect? - se referă la respectarea specificațiilor, la utilitatea produsului I; **Terminologie: Eroare** = o acțiune umană care are ca rezultat un defect în produsul software; **Defect** - consecința unei erori în produsul software - un defect poate fi latent: nu cauzează probleme cât timp nu apar condiții care determină execuția unui anumitor linii de cod; **Defecțiunile** manifestarea defectelor: când execuția programului întâlnește un defect, acesta provoacă o defecțiune - abaterea programului de la comportamentul așteptat; **Testare si depanare:** **testarea de validare** - intenționează să arate că produsul nu îndeplinește cerințele - testele încearcă să arate că o cerință nu a fost implementată adecvat; **testarea defectelor** - teste proiectate să descopere prezența defectelor în sistem - testele încearcă să descopere defecte; **depanarea** - are ca scop localizarea și repararea erorilor corespunzătoare - implică formularea unor ipoteze asupra comportamentului programului, corectarea defectelor și apoi re-testarea programului; **Asigurarea calitatii** spre deosebire de testare, ea se refera la prevenirea defectelor - se ocupa de procesele de dezvoltare care sa conduca la producerea unui software de calitate - include procesul de testare a produsului - **principii de testare** - o parte necesară a unui caz de test este definirea ieșirii sau rezultatului așteptat - programatorii nu ar trebui să-și testeze propriile programe (excepție - testarea unitară) - organizațiile ar trebui să folosească și companii (sau departamente) externe pentru testarea propriilor programe - rezultatele testelor trebuie analizate amănunțit - trebuie scrise cazuri de test atât pentru condiții de intrare invalide și neașteptate, cât și pentru condiții de intrare valide și așteptate - pe cât posibil, cazurile de test trebuie salvate și re-executate după efectuarea unor modificări - probabilitatea ca mai multe erori să existe într-o secțiune a programului este proporțională cu numărul de erori deja descoperite în acea - **testarea unitară** - o unitate/modul = de obicei, clasa sau funcție sau bibliotecă, driver - testarea unei unități se face în izolare -> folosirea de stubs; - **testarea de integrare** - testează interacțiunea mai multor unități - testarea e determinata de arhitectura; - **testarea sistemului** - testeaza aplicatia ca intreg - aplicatia trebuie sa execute cu succes toate scenariile - se face cu script-uri care ruleaza cu o serie de parametri si colecteaza rezultatele - trebuie realizat de o echipa

separata; - **testarea de acceptanta** - det. daca sunt indeplinite cerintele unei specificatii/contract cu clientul - mai multe tipuri: teste rulate de dezvoltator înainte de livrare, teste rulate de utilizator, teste de operationalitate, alfa si beta; - **testele de regresiune** - un test valid genereaza un set de rezultate verificate, "standardul de aur" - aceste teste sunt utilizate la re-testare pentru a asigura faptul ca noile modificari nu au introdus noi defecte; - **testarea performantei** - reliability - **securitatea** - **utilizabilitatea** - **load testing** (asigura faptul ca sistemul poate gestiona un volum asteptat de date - verifica eficient sistemului si modului in care scalezaza acesta pentru un mediu real de executie) - **testarea la stres** (solicita sistemul dincolo de incarcarea maxima proiectata - testeaza modul in care cade sistemul - soak testing presupune rularea sistemului pentru o perioada lunga de timp); - **testarea interfeței cu utilizatorul** - presupune memorarea unor parametri si elaborarea unor modalitati prin care mesajele sa fie transmise din nou aplicatiei, la un moment ulterior - se folosesc script-uri pentru testare; - **testarea uzabilitatii** - test. Cat de usor de folosit este sistemul - se poate face cu utilizatori din lumea reala, cu log-uri, prin recenzii ale unor experti, A/B testing (modificare unui element din UI si verificare comportamentului unui grup de utilizatori); - **inspectiile codului** - citirea codului cu scopul de a detecta erori - 4 membri: moderatorul (programator competent), programatorul (a scris codul), designer-ul (daca e diferit de programator), specialist in testare - programatorul citește logica programului instr. cu instr. Iar ceilalti pun intrebari - programatorul nu trebuie sa fie defensiv, ci constructiv **Testare de tip cutie neagra** - se iau in considerare numai intrarile si iesirile dorite, conform specificatiilor - structura internă este ignorată - se mai numeste si testare functionala deoarece se bazeaza pe functionalitatea descrisa în specificatii - poate fi folosita la orice nivel de testare - metode de testare: - **partitionare in clase de echivalenta** - datele de intrare sunt partitionare in clase ai datele dintr-o clasa sunt tratate in mod identic, fiind suficient sa alegem cate o valoare din fiecare clasa - **AVANTAJE** - reduce drastic numarul de date de test doar pe baza specificatiei - potrivita pentru aplicatii de tipul procesarii datelor, in care intrarile si iesirile sunt usor de identificat si iau valori distincte - **DEZAVANTAJE** - modul de definire al claselor nu este evident - desi specificatia ar putea sugera ca un grup de valori sunt procesate identic, acest lucru nu este tot timpul adevarat - mai puțin aplicabile pentru situatii cand intrările si ieșirile sunt simple, dar procesarea este complexă - **analiza valorilor de frontiera** - folosita impreuna cu partitionarea de echivalenta - se concentreaza pe examinarea valorilor de frontiera ale claselor, care de regula sunt o sursa importanta de erori - **partitionarea in categorii** - se vazeaza pe cele 2 metode anterioare - **PASI:** 1. descompune specificatia functională în unități (programe, funcții, etc.) care pot fi testate separat 2. pentru fiecare unitate, identifică parametri și condițiile de mediu (ex. starea sistemului la momentul execuției) de care depinde comportamentul acestora 3. găsește categoriile (proprietăți sau caracteristici importante) fiecăru parametru sau condițiile de mediu. 4. partiționează fiecare categorie în alternative. O alternativă reprezintă o mulțime de valori similare pentru o categorie. 5. scrie specificația de testare. Aceasta constă din lista categoriilor și lista alternativelor pentru fiecare categorie. 6. creează cazuri de testare prin



<p>...gerea unei combinații de alternative din specificația de testare (fiecare categorie contribuie cu zero sau o alternativă). 7. crearea date de test alegând o singură valoare pentru fiecare alternativă. – AVANTAJE și DEZAVANTAJE – pași de început, adică iden.</p> <p>Parametrilor și a condițiilor de mediu precum și a categoriilor, nu sunt bine definiți. Pe de alta parte, oada ca acești pași au fost trecuți, aplicarea metodei este clară – este mai clar decât celelalte metode cutie neagră și poate produce date de testare mai cuprinzătoare. Pe de alta parte, datorită exploziei combinatorice, pot rezulta date de test de foarte mari dimensiuni.</p> <p><b>Testare de tip cutie albă</b> – ia în calcul codul sursă al metodelor testate – se mai numește testare structurală – datele de test sunt generate pe baza implementării programului – structura programului poate fi reprezentată sub forma unui graf orientat – datele de test sunt alese ai sa parcurga toate elementele grafului macar o singura data</p> <p>– <b>acoperire la nivel de instrucțiune</b> – fiecare instrucțiune (nod al grafului) este parcursă macar o dată – este posibil de obicei ca nivelul minim de acoperire pe care îl poate atinge testarea structurală – frecvent, aceasta acoperire nu poate fi obținută dintr-o singură pricină 1. Existenței unor porțiuni de cod care nu pot fi atinse niciodată (eroare de design) 2. Porțiuni de cod care nu se pot executa doar în situații speciale. În acest caz, soluția este o inspecție riguroasă a codului. – AVANTAJE – realizează execuția măcar o singură dată a fiecărei instrucțiuni – în general, ușor de realizat – DEZAVANTAJE – nu testează fiecare condiție în parte în cazul condițiilor compuse – nu testează fiecare ramură – probleme la instrucțiunile în care else</p> <p>– <b>acoperire la nivel de ramură</b> – fiecare ramură a grafului e parcursă macar o dată – generează date de test care testează cazurile când fiecare decizie este adevărată sau falsă – se mai numește și "decision coverage" – DEZAVANTAJ – nu testează condițiile individuale ale fiecărei decizii</p> <p>– <b>acoperire la nivel de condiție</b> – generează date de test astfel încât fiecare condiție individuală dintr-o decizie să ia atât valoarea adevărată cât și valoarea falsă (dacă acest lucru este posibil) – AVANTAJE – se concentrează asupra condițiilor individuale – DEZAVANTAJE – poate să nu realizeze o acoperire la nivel de ramură – pentru a rezolva aceasta slăbiciune se poate folosi testarea la nivel de decizie</p> <p>– <b>acoperire la nivel de condiție/decizie</b> – generează date de test astfel încât fiecare condiție individuală dintr-o decizie să ia atât valoarea adevărată cât și valoarea falsă</p> <p>– <b>acoperire MC/DC</b> – fiecare condiție individuală dintr-o decizie ia atât valoarea True cât și valoarea False – fiecare decizie ia atât valoarea True cât și valoarea False! I fiecare condiție individuală influențează în mod independent decizia din care face parte – AVANTAJE – acoperire mai puternică decât acoperirea condiție/decizie simplă, testând și influența condițiilor individuale asupra deciziilor – produce teste mai puține – depinde liniar de numărul de condiții</p> <p>– <b>acoperire la nivel de cale</b> – generează date pentru executarea fiecărei căi măcar o singură dată –</p>	<p>...nfectate, deci aici este locul de infectare! – <b>Simplificarea problemei intrărilor mari</b> – Dorim un test mic care eșuează O soluție divide-et-impera 1. tăiem o jumătate din intrarea testului 2. verificăm dacă una din jumătăți conduce încă la o problemă 3. continuăm până când obținem un test minim care eșuează – <b>Clasificarea defectelor</b> - defecte critice: afectează mulți utilizatori, pot întârzia proiectul - defecte majore: au un impact major, necesită un volum mare de lucru pentru a le repara, dar nu afectează substanțial graficul de lucru al proiectului - defecte minore: izolate, care se manifestă rar și au un impact minor asupra proiectului - defecte cosmetice: mici greșeli care nu afectează funcționarea corectă a produsului software urmărire</p> <p><b>Design patterns</b> – = soluții generale reutilizabile la probleme care apar frecvent în proiectare (OO) – un sablon e suficient de general pentru a fi aplicat în mai multe situații, dar suficient de concret pentru a fi util în luarea deciziilor – <b>folositoare în următoarele feluri</b> - ca mod de a învăța practici bune - aplicarea consistentă a unor principii de generale de proiectare - ca vocabular de calitate de nivel înalt (pentru comunicare) - ca autoritate la care se poate face apel - în cazul în care o echipă sau organizație adoptă propriile design patterns - un mod de a explicita cum se fac lucrurile acolo – D: pot crește complexitatea și scadea performanța – în <b>ingineria software</b> – sunt soluții generale reutilizabile la probleme care apar frecvent în proiectare – <b>tipuri de sabloane</b> – arhitecturale (la nivelul arhitecturii ex. MVC, publish-subscribe)/de proiectare (la nivelul claselor/modulelor)/idiomuri (la nivelul limbajului ex. MVC, publish-subscribe) – <b>sabloanele de proiectare</b> – 23 de sabloane clasice, creationale (instantierea), structurale (compunerea), comportamentale (comunicarea) – creationale = Abstract Factory, Builder, Factory Method, Prototype, Singleton – structurale = adapter, bridge, composite, decorator, façade, flyweight, proxy – comportamentale = chain of responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template method, visitor – <b>principii de bază</b> – programare folosind multe interfețe – se prefera compoziția în loc de moștenire – se urmărește decuplarea – <b>sablonul creational SINGLETON</b> – asigură existența unei singure instanțe pt. o clasă – oferă un punct global de acces la instanța – aceeași instanță poate fi utilizată de oriunde fiind imposibil de a invoca direct constructorul de fiecare dată – aplicabilitate: cand doar un obiect al unei clase e cerut, instanța este accesibilă global, folosit în alte sabloane – consecințe: accesul e controlat la instanța, spațiul de adresare structurat – <b>sablonul c. ABSTRACT FACTORY</b> – oferă o interfață pentru crearea de familii de obiecte înrudite sau dependente fără a specifica clasele lor concrete – observatii: independent de modul în care produsele sunt create, compuse și reprezentate, produsele înrudite trebuie să fie utilizate împreună, pune la dispoziție doar interfața, nu și implementarea – consecințe: numele de clasă de produse nu apar în cod, familiile de produse ușor interschimbabile, cere consistența între produse – <b>sablonul c. BUILDER</b> - separă construcția unui obiect complex de</p>	<p>...reprezentarea sa, astfel încât același proces de construcție poate crea reprezentări diferite – observatii: folosit când algoritmul de creare a unui obiect complex este independent de părțile care compun obiectul și de modul lor de asamblare și când procesul de construcție trebuie să permită reprezentări diferite pentru obiectul construit - comparație cu Abstract Factory: Builder creează un produs complex pas cu pas. Abstract Factory creează familii de produse, de fiecare dată produsul fiind complet – <b>sablonul s. FAÇADE</b> – oferă o interfață unificată pentru un set de interfețe într-un subsistem - observații: o interfață simplă la un subsistem complex când sunt multe dependențe între clienți și subsistem, este redusă cuplarea – <b>sablonul c. OBSERVER</b> - presupunem o dependență de 1:n între obiecte - schimbarea stării unui obiect înștiințează toate obiectele dependente - de exemplu: poate menține consistența între perspectiva internă și cea externă – observatii: Structura obiect cu mai multe interfețe diferite - operațiuni distincte și independenți pe structura obiect – nu este potrivit pentru evoluția structurilor de obiecte – consecințe: adăugarea de operații se face ușor, încalcare parțială a encapsulării – <b>ANTI-SABLOANE</b> – abstraction inversiopl, input kludge, interface bloat, magic pushbutton, race hazard, stovepipe system, anemic domain model etc.</p> <p><b>Refactoring</b> – schimbare în structura internă a unui produs software cu scopul de a-l face mai ușor de înțeles și de modificat fără a-i schimba comportamentul observabil – schimbările pot introduce noi defecte – nu sunt introduse noi funcționalități – îmbunătățește structura, nu codul – nu trebuie să introducă niveluri de complexitate inutile – <b>Semnale</b>: cod duplicat, metode lungi, clase mari, liste lungi de parametri, comunicare intensă între obiecte – <b>optimizarea metodelor</b> – scop: simplificarea și creșterea coeziunii - împartirea unei metode în mai multe (metode care ret. O val. si simba devin 2 separate), adăugarea sau ștergerea de parametri – <b>optimizarea claselor</b> – scop: creșterea coeziunii și reducerea cuplării – mutarea metodelor – mutarea campurilor – extragerea de clasă – înlocuirea valorilor de date cu obiecte</p> <p><b>Metrici</b> – dimensiune – complexitate (nivelul de dificultate în înțelegerea unui modul) – <b>dimensiune</b> – LOC = line of code = linie de cod nevida, nu e comentariu – LOC corelată cu productivitatea, costul și calitatea – <b>complexitatea ciclomatică</b> – = indica nivelul de dificultate în înțelegerea unui modul – <math>M = e - n + 2 * p</math>, unde <math>n</math> = nr. De noduri <math>e</math> = numărul de arce <math>p</math> = numărul de componente conexe (pentru un modul este 1) – complexitatea ciclomatică a unui modul este numărul de decizii + 1 – aceasta metrică este corelată cu dimensiunea codului și cu numărul de defecte – <b>variabile vii</b> - variabilă este via de la prima până la ultima referință dintr-un modul, incluzând toate instrucțiunile intermediare - pentru o instrucțiune, numărul de variabile vii reprezintă o măsură a dificultății de înțelegere a acelei instrucțiuni – <b>anvergura</b> - numărul de instrucțiuni dintre 2 utilizări succesive ale unei variabile - dacă o variabilă este referențiată de n ori, atunci are n – 1 anverguri -</p>	<p>...anvergura medie este numărul mediu de instrucțiuni executabile dintre 2 referiri succesive ale unei variabile</p> <p><b>Licente</b> – = consimțământul pe care titularul dreptului de autor îl dă unei persoane pentru a putea reproduce, folosi, difuza sau importa copii ale unui program de calculator – drepturi de autor aparțin de categoria generală numită proprietate intelectuală - <b>patente</b> - mai puternice decât drepturile de autor: oprește alte persoane să producă acel obiect, chiar dacă l-au inventat independent – putem patenta un algoritm sau un proces de afaceri – <b>copyright</b> - protejează codul sursă, nu și ideea – <b>licente</b> - comerciale (pe calculator sau utilizator) - shareware (acces limitat temporal sau funcțional) - freeware (gratuit) - open source: cod sursă disponibil și redistribuibil – <b>GPL</b> - cere ca orice modificări sau adaptări ale unui cod GPL, inclusiv software-ul care folosește bibliotecă GPL, să fie sub licența GPL (natura virală) - nu obligă distribuirea codului modificat și nu împiedică perceperea de taxe pentru furnizarea software-ului; și nici nu împiedică taxarea pentru întreținere sau modificări - adecvată atunci când se dorește ca software-ul să fie accesibil în mod liber și să nu poată fi folosit de către cineva care nu oferă codul sursă utilizatorilor externi – <b>LGPL</b> - LGPL impune restricții copyleft pe cod, dar nu și pentru software-uri care doar folosesc codul respectiv</p> <p><b>Management</b> – proiect = set de activități planificate definit prin început și sfârșit, obiectiv, domeniu de aplicare, buget, nerepetitiv – proiect de succes = executat în timpul dat, în bugetul disponibil și în parametrii de calitate ceruți – <b>managementul proiectului</b> – consta în distribuția și controlul bugetului, timpului și personalului – <b>manager de proiect</b>: planificare, organizare, constituire echipa, monitorizare, control, reprezentare – <b>etape ale unui proiect</b>: studiu de fezabilitate sau business case, planificare, execuție – <b>studii de fezabilitate</b>: Obiective – Motivație – Rezumat: descriere sumară a produsului/serviciului, descriere generală a soluției propuse, descriere generală a planului de implementare propus - Detalii privind soluția propusă – impactul proiectului – costuri – <b>planificarea</b>: - Domeniul de aplicare și obiectivele - Identificarea infrastructurii de aplicare și obiectivelor - Analiza caracteristicilor proiectului - Identificarea activităților și livrabilor - Estimarea eforturilor pentru fiecare activitate</p>
---	--	--	--