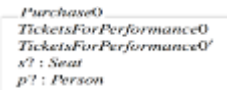
Notatia Z - O specificație formală:

- folosește notații matematice pentru a descrie într-un model precis ce proprietăți trebuie să aibă un sistem
- descrie ce trebuie să facă sistemul și nu cum ! - independent de cod
- poate fi folosită pentru înțelegerea cerințelor și analiza lor (uneori se poate si genera cod dintr-o specificație suficient de precisă)

În Z descompunerea unei specificații se face în mai multe piese numite scheme



Compunerea schemelor de operatii: Schemele de operatii se pot compune folosind unde:

- fiecare variabila cu ' din prima schema trebuie sa apara fara ' in a doua
- aceste variabile se identifica si se ascund (nu mai sunt vizibile in exterior)

Cerinte - Cerințe utilizator: afirmații în limbaj

natural și diagrame a serviciilor oferite de sistem laolaltă cu constrângerile operaționale. scrise pentru clienți. Trebuie să descrie cerințe funcționale și non-funcționale într-o manieră în care sunt pe înțelesul utilizatorilor sistemului care nu dețin cunoștințe tehnice detaliate. **Cerințele sistemului**: un document structurat stabilind descrierea detaliată a funcțiilor sistemului, serviciile oferite și constrângerile operaționale. poate fi parte a contractului cu clientul. Cerințele sistemului sunt specificate mai detaliat decât cerințele utilizator. Scopul principal al lor este acela de a fi baza proiectării sistemului. Cerințele trebuie să exprime ce poate face sistemul, iar proiectul trebuie să exprime cum se poate implementa sistemul. Ele pot fi incorporate în contract.

Cerințele utilizator se adresează: utilizatorilor finali inginerilor clientului proiectanților de sistem managerilor clientului managerilor de contracte

Cerințele de sistem se adresează: utilizatorilor finali inginerilor clientului proiectanților de sistem programatorilor. **Cerințe funcționale**: afirmații despre servicii pe care sistemul trebuie să le conțină, cum trebuie el să răspundă la anumite intrări și cum reacționeze în anumite situații. **Cerințe non-funcționale**: Constrângeri ale serviciilor si funcțiilor oferite de sistem cum ar fi constrângeri de timp, constrângeri ale procesului de dezvoltare, standarde

JML - Limbaj de specificație formală pentru Java.

Comentarii în textul sursă care descriu formal cum trebuie să se comporte un modul Java, prevenind astfel ambiguitatea. Folosește invarianți, pre- și postcondiții. Urmează paradigma "design by contract"

Specificatii de tip contract: apelatul garantează un anumit rezultat cu condiția ca apelantul să garanteze anumite premise. Contract = preconditie + postconditie, adica daca preconditia este respectata, postconditia este garantata. Adnotarile JML sunt integrate in codul sursa java.

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @
  @ also                                  Mai multe cazuri de specificare
  @                                       conectate prin also
  @ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter < 2
  @ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
  @*/
public void enterPIN (int pin) {...
```

Exemple: - **toate elementele unui vector vec sunt mai mici sau egale decât 2**:
(\forall int i; 0 <= i && i < vec.length; vec[i] <= 2)
- **variabila m are valoarea elementului maxim din vectorul vec**:
(\forall int i; 0 <= i && i < vec.length; m >= vec[i]) && (vec.length > 0 ==> (\exists int i; 0 <= i && i < vec.length; m==vec[i]))
- **toate instanțele clasei BankCard au câmpul cardNumber diferit**:
(\forall BankCard p1, p2; \created(p1) && \created(p2); p1 != p2 ==> p1.cardNumber != p2.cardNumber)

```
public class BankCard {
/*@ public static invariant
  @ (\forall BankCard p1, p2;
  @     \created(p1) && \created(p2);
  @     p1 != p2 ==> p1.cardNumber != p2.cardNumber);
  @*/

private /*@ spec_public @*/ int cardNumber;

  // restul clasei aici
}
```

UML in general - Modelare. De ce?

Complexitatea e o problema în dezvoltarea programelor. Folosirea unor modele poate înlesni abordarea de complexității. Un model este o reprezentare abstractă, de obicei grafică, a unui aspect al unui sistem. Acesta permite o mai bună înțelegere a sistemului și analiza unor proprietăți ale acestuia.

UML este un limbaj grafic pentru vizualizarea, specificarea, construcția și documentația necesare pentru dezvoltarea de sisteme software (OO) complexe.

Avantaje UML: UML este standardizat - existența multor tool-uri - flexibilitate: modelarea se poate adapta la diverse domenii folosind "profiluri" și "stereotipuri" - portabilitate: modelele pot fi exportate în format XMI (XML Metadata Interchange) și folosite de diverse tool-uri - se poate folosi doar o submulțime de diagrame - arhitectura software e importantă

Dezavantaje UML: Nu este cunoscută notația UML - UML e prea complex (14 tipuri de diagrame) - Notațiile informale sunt suficiente - Documentarea arhitecturii nu e considerată importantă

Folosit la: - modelarea unor aspecte ale sistemului – documentatia proiectului – diagrame detaliate folosite in tool-uri pentru a obtine cod generat

Use Case UML Diagrams - descriu

comportamentul sistemului din punctul de vedere al utilizatorului

- parti principale: sistem (componente si descrierile acestora), utilizatori (componente externe) - cuprinde: diagrama cazurilor de utilizare si descrierea lor

Componente: caz de utilizare (= unitate coerenta de functionalitate, task; repr. Printr-un oval), actor (element extern care interactioneaza cu sistemul), asociatii de comunicare (legaturi intre actori si cazuri de comunicare), descrierea cazurilor de utilizare (document ce descrie secventa evenimentelor)

Actori: primari (=beneficiari) / secundari (= cu ajutorul carora se realizeaza cazul de util.), umani / sisteme externe

Cazuri de utilizare: reprezinta multimi de scenarii referitoare la utilizarea unui sistem – pot avea complexitati diferite

Frontiera sistemului: face distinctia intre mediul extern si cel intern (= responsabilitatile sistemului) – cazurile de utilizare sunt inauntru, actorii sunt afara – se stabileste de obicei la frontiera dintre hardware si software

Relatia << include >>: - arata ca secventa de even. descrisa in cazul de utilizare inclus se gaseste si in secventa de even. a cazului de utilizare de baza – folosita atunci cand doua sau mai multe cazuri au o componenta comuna, sau pt. a evidentia anumiti pasi

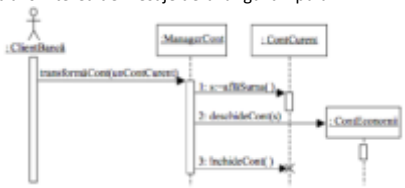
Relatia << extend >>: - folosita pt. separarea diferitelor comportamente ale cazurilor de util. i.e daca un caz de utilizare contine doua sau mai multe scenarii diferite – de obicei se foloseste pt. a pune in evidenta exceptiile – putem specifica si punctul de extensie

Relatia de generalizare: - intre cazuri indica faptul ca un caz poate mosteni comportamentul definit altuia – intre actori arata ca unul mosteneste comportamentul altuia – fol. Pt evidentierea anumitor versiuni ale unui task.

Folosite pentru: - analiza: identifica functionalitatea ceruta si o valideaza impreuna cu clientii – design si implementare: trebuie realizate – testare: baza pentur generarea cazurilor de testare

Sequence UML Diagrams - Evidentiaza

transmiterea de mesaje de-a lungul timpului



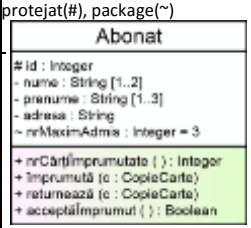
Tipuri de mesaje: - sageata 1 = mesaj sincron = obiectul pierde controlul pana cand primeste un raspuns – sageata 2 = mesaj raspuns = optional – sageata 3 = mesaj asincron = nu asteapta raspuns

Fragmente: - opt [garda] – alt [garda] [else] – loop(nr. rep)

Class UML Diagrams – folosite pentru a

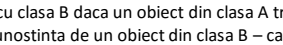
specifica structura statica a sistemului, adica ce clase exista in sistem si care este legatura dintre ele

- reprezentare grafica – dreptunghi cu 3 randuri: numele clasei, atribute si operatii
- cuantificatori de vizibilitate: public (+), privat(-), protejat(#), package(~)



Relatii intre clase: asociere, generalizare, dependenta, realizare

Asocieri: - legaturi structurale intre clase – clasa A este asociata cu clasa B daca un obiect din clasa A trebuie sa aiba cunostinta de un obiect din clasa B – cazuri: un ob. din clasa A trimite un mesaj catre un ob. din clasa B; un obiect din A creeaza un obiect din B; un ob. din A are un atribut ale carui valori sunt ob. sau colectii de ob. din B



EX: obiectul Curs are cunostinta de Student, insa nu invers. Daca asocierea nu are sageti, este implicit bidirectionala - **Agregarea** este modul cel mai general de a indica in UML o relatie de tip parte-intreg. Diferenta dintre o simpla asociere si agregare este pur conceptuala: folosirea agregarii indica faptul ca o clasa reprezinta un lucru mai mare, care contine mai multe lucruri mai mici - **Compunerea** este un caz special de agregare, in care relatia dintre intreg si partile sale e mai puternica – daca intregul este creat, mutat sau distrus, acelasi lucru se intampla si cu partile componente. De asemenea, o parte nu poate sa fie continuta in mai mult de un singur intreg. – **Asocieri mutual exclusive** = 2 sau mai multe asocieri care nu pot exista in acelasi timp [xor] – **Clase de asociere**: o asociere poate avea date si responsabilitati proprii; de exemplu, nota studentului la curs

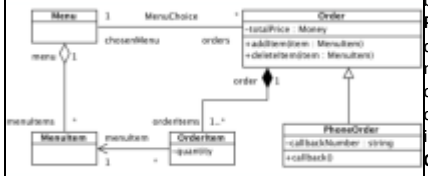
Generalizare: - relație între un lucru general (numit superclasă sau părinte, ex. Abonat) și un lucru specializat (numit subclasă sau copil, ex. AbonatPremium) - = mostenire simpla sau multipla – clase abstracte

Dependente: - o clasă A depinde de o clasă B dacă o modificare în specificația lui B poate produce modificarea lui A, dar nu neapărat și invers - cel mai frecvent caz de dependență este relația dintre o clasă care folosește altă clasă ca parametru într-o operație- notația este o săgeată cu linie punctată spre clasa care este dependentă de cealaltă clasă

Interfete: - În UML, o interfață specifică o colecție de operații și/sau atribute, pe care trebuie să le furnizeze o clasă sau o componentă - O interfață este evidențiată prin stereotipul « interface » deasupra numelui - Faptul că o clasă realizează (sau corespunde) unei interfațe este reprezentat grafic printr-o linie întreruptă cu o săgeată triunghiulară

Diferente intre INTERFETE si GENERALIZARE – interfata nu presupune o relatie stransa intre clase precum generalizarea – atunci cand se intentioneaza crearea unor clase inrudite, care au comportament comun, folosim generalizarea – daca se vrea doar o multime de obiecte care sunt capabile sa efectueze niste operatii comune, atunci interfata e de preferat

Stereotipuri: - O anumită caracteristică a unei clase (și nu numai) poate fi evidențiată folosind stereotipuri. Acestea sunt etichete plasate deasupra numelui



State UML Diagrams - descriu dependența

dintre starea unui obiect și mesajele pe care le primește sau alte evenimente recepționate –

Elemente: stari (dreptunghiuri cu colturi rotunjite), **tranzitii intre stari (**sageti), **evenimente** (declanseaza tranzitiile intre stari)

Stari - O stare este o mulțime de configurații ale obiectului care se comportă la fel la apariția unui eveniment - O stare poate fi identificată prin constrângeri aplicate atributelor obiectului

Eveniment - ceva care se produce asupra unui obiect, precum primirea unui mesaj. **Actiune** - ceva care poate fi făcut de către obiect, precum transmiterea unui mesaj. **Reprezentare pe tranziții: eveniment [garda] / acțiune. Garzi -** un eveniment declanșează o tranziție numai dacă atributele obiectului îndeplinesc o anumită condiție suplimentară (gardă). **Stari compuse** - O stare S poate conține substări care detaliază comportamentul sistemului în starea S. În acest caz, spunem ca S este o stare compusă - Exemplu: situația căutării unui canal de televiziune se face în timp ce televizorul este activ și poate fi reprezentată ca o diagramă de stare inclusă CăutareCanal - Astfel, starea Activ va deveni compusă, incluzând subcomportamentul de căutare. Pentru aceasta se folosește notația include/CăutareCanal. **Stari istoric -** Uneori este necesar ca submașina să-și "reamintească" starea în care a rămas și să-și reia funcționarea din acea stare - Pentru acest lucru se folosește o stare "istoric", reprezentată printr-un cerc în care apare litera H. **Stari concurente** - Există posibilitatea exprimării activităților concurente dintr-o stare - Grafic: se împarte dreptunghiul corespunzător stării compuse printr-o linie punctată, în regiunile obținute fiind reprezentate submașinile care vor acționa concurent.

Procese de dezvoltare software

Procesul de dezvoltare cascada "waterfall"

- cerinte -> design -> implementare -> testate -> mentenanta

- **analiza și definirea cerințelor**: Sunt stabilite serviciile, constrângerile și scopurile sistemului prin consultare cu utilizatorul. (ce trebuie să facă sistemul) - **design**: Se stabilește o arhitectură de ansamblu și funcțiile sistemului software pornind de la cerințe. (cum trebuie să se comporte sistemul) - **implementare și testare unitară**: Designul sistemului este transformat într- o mulțime de programe (unități de program); testarea unităților de program verifică faptul că fiecare unitate de program este conformă cu specificația **- integrare și testare sistem**. Unitățile de program sunt integrate și testate ca un sistem complet; apoi acesta este livrat clientului - **operare și mentenanță** Sistemul este folosit în practică; mentenanța include: corectarea erorilor, îmbunătățirea unor servicii, adăugarea de noi funcționalități.

Avantaje si dezavantaje - fiecare etapă nu trebuie sa înceapă înainte ca precedenta să fie încheiată - fiecare fază are ca rezultat unul sau mai multe documente care trebuie "aprobate" - bazat pe modele de proces folosite pentru productia de hardware **Avantaj**: proces bine structurat, riguros, clar; produce sisteme robuste

Probleme: dezvoltarea unui sistem software nu este de obicei un proces liniar; etapele se întrepătrund - metoda oferă un punct de vedere static asupra cerințelor - schimbarile cerințelor nu pot fi luate în considerare după aprobarea specificației nu permite implicarea utilizatorului după aprobarea specificației

Concluzie: Modelul cascadă trebuie folosit atunci cand cerințele sunt bine înțelese și când este necesar un proces de dezvoltare clar și riguros

Procesul incremental

- sunt identificate cerințele sistemului la nivel înalt, dar, în loc de a dezvolta și livra un sistem dintr-o dată, dezvoltarea și livrarea este realizată în părți (incremente), fiecare increment încorporând o parte de funcționalitate - cerințele sunt ordonate după priorități, astfel încât cele cu prioritatea cea mai mare fac parte din primul increment, etc - după ce dezvoltarea unui increment a început, cerințele pentru acel increment sunt înghețate, dar cerințele pentru noile incremente pot fi modificate.

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

deoarece cerințele cele mai importante fac parte din primele incremente, acestea vor fi testate cel mai mult – **Probleme** - dificultăți în transformarea cerințelor utilizatorului in incremente de mărime potrivită - procesul nu este foarte vizibil pentru utilizator (nu e suficientă 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-ul (de exemplu, prin limitarea documentației) și de a răspune rapid cerințelor în schimbare. **Se pune accent pe** - indivizii și interacțiunea 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 livrarea 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. Construiește proiecte în jurul oamenilor motivați. Oferă-le mediul propice și suportul necesar și ai încredere 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 — arta de a maximiza cantitatea de muncă nerealizată — 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 să devină mai eficientă, apoi își adaptează și ajustează comportamentul în consecință.
Aplicablitatea metodelor agile - in companii care dezvolta 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 implicați 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 si principle agile - "dezvoltarea incrementală" este susținută prin intermediul livrării de software în mod frecvent cu mici incremente - "implicarea clientului" înseamnă angajamentul "full-time" al clientului cu echipa de dezvoltare - "oameni, nu procese" prin programare 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: livrari:** Clientul înțelege domeniul de aplicare, prioritățile, nevoile business ale versiunilor care trebuie livrate: sortează "cartonașele" cu sarcini după prioritați; **iteratii:** Dezvoltatorii estimează riscurile și eforturile: sortează "cartonașele" după risc dacă o sarcină ia mai mult de 2-4 săptămâni, e distribuită pe mai multe "cartonașe"
- **Metafora** - = arhitectura sistemului – se evita cuvantul "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 - daca toate testele sunt pozitive, adaugă modificările în sistemul care se ocupa cu managementul codului sursă
- **Proiectare simpla:** "proiectează cel mai simplu lucru care funționează 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.
- **imbunatatirea codului**: îmbunătățirea codului prin "refactoring" este foarte importantă deoarece XP recomandă începerea implementarii foarte repede ex: "three strikes and you refactor"
- **programarea in 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 ineficientă (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 sfârșitului sprintului, rezultatul ar trebui să fie livrabil (adică folosit de client sau vandabil). n după o analiză a sprintului, se reiterează.

Metode agile	Metode cascadă	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 intre aceste universuri si operatori care le conecteaza permitand exprimarea diverselor tipuri de modalitati – daca 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 efectuarii instructiunilor – logica predicativa clasica se foloseste pentru a specifica relatii intre valorile dintr-o stare a variabilelor din program – ce lipseste este un mecanism care sa conecteze universurile starilor intre ele -> folosim CTL
Timpul in 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 alternativele din punctul de ramificare – E = exista cel putin o alternativa din punctul de ramificare
- **G si F – cuantifica de-a lungul ramurilor –** G = toate starile viitoare de pe drum – F = exista cel putin 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*:

$$AG(\mathrm{EF}\ restart)$$

un lift care se deplasează în sus la etajul 2 nu-și schimbă direcția dacă pasagerii merg la etajul 5:

$$AG(floor = 2 \wedge direction = up \wedge ButtonPressed5 \rightarrow \wedge direction = up\ \mathrm{U}\ floor = 5)$$

un lift poate rămâne inactiv la etajul 3 cu ușile închise

$$AG(floor = 3 \wedge idle \wedge door = closed \rightarrow \mathrm{EG}(floor = 3 \wedge idle \wedge door = closed))$$

EF *fault*
= este posibil să se ajungă într-o stare în care *fault* = true

AF AG *stable*
= în orice execuție, la un moment dat, stabile este invariant

este posibil să ajungem într-o stare unde un proces a început (*started*), dar nu este încă gata (*ready*):

$$\mathrm{EF}(started \wedge \neg ready)$$

pentru orice stare, dacă a apărut o cerere (*request*), atunci ea va fi ulterior confirmată (*acknowledged*):

$$\mathrm{AG}(request \rightarrow \mathrm{AF}\ acknowledged)$$

un proces este disponibil (*enabled*) de o infinitate de ori pe orice drum:

$$\mathrm{AG}(\mathrm{AF}\ enabled)$$

orice s-ar întâmpla, procesul va fi permanent blocat (*deadlocked*):

$$\mathrm{AF}(\mathrm{AG}\ deadlocked)$$

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 operatorii 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 !;
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 condițiile care determină execuția anumitor linii de cod; **Defecțiune** manifestarea unui defect: 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 modificari - probabilitatea ca mai multe erori să existe într-o secțiune a programului este proporțională cu numărul de erori deja descoperite în acea
- **testarea unitara** – o unitate/modul = de obicei, clasa sau functie sau biblioteca, driver – testarea unei unitati se face in izolare -> folosirea de stubs; - **testarea de integrare** – testeaza interactiunea mai multor unitati – 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 inainte 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 modul in care scaleaza 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 interfetei 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 citeste 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 interna este ignorata – se mai numeste si testare functionala deoarece se bazeaza pe functionalitatea descrisa in 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 putin aplicabile pentru situatii cand intrarile si iesirile sunt simple, dar procesarea este complexa – **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 specificația funcțională în unități (programe, funcții, etc.) care pot fi testate separat 2. pentru fiecare unitate, identifică parametrii și condițiile de mediu (ex. starea sistemului la momentul execuției) de care depinde comportamentul acesteia 3. găsește categoriile (proprietăți sau caracteristici importante) fiecărui 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 alegerea unei combinații de alternative din specificația de testare (fiecare categorie contribuie cu zero sau o alternativă). 7. creeaza date de test alegând o singură valoare pentru fiecare alternativă. – AVANTAJE si DEZAVANTAJE – pasii de inceput, adica iden. Parametrilor si a conditiilor de mediu precum si a categoriilor, nu sunt bine definiti. Pe de alta parte, oadata ce acesti pasi au fost trecuti, aplicarea metodei este clara – este mai clar decat celelalte metode cutie neagra si poate produce date de testare mai cuprinzatoare. Pe de alta parte, datorita exploziei combinatorice, pot rezulta date de test de foarte mari dimensiuni.

Testare de tip cutie alba – ia in calcul codul sursa al metodelor testate – se mai numeste testare structurala – datele de test sunt generate pe baza implementarii programului – structura programului poate fi reprezentata sub forma unui graf orientat – datele de test sunt alese ai sa parcurga toate elementele grafului macat o singura data

- acoperire la nivel de instructiune – fiecare instructiune (nod al grafului) este parcursa macar o data - este privită de obicei ca nivelul minim de acoperire pe care îl poate atinge testarea structurală - frecvent, aceasta acoperire nu poate fi obtinuta din pricina 1. Existentei unor portiuni de cod care nu pot fi atinse niciodata (eroare de design) 2. Portiuni de cod care nu se pot executa doar in situatii speciale. In acest caz, solutia este o inspectie riguroasa a codului. – AVANTAJE - realizează execuția măcar o singura dată a fiecărei instrucțiuni - în general, ușor de realizat – DEZAVANTAJE – nu testeaza fiecare conditie in parte in cazul conditiilor compuse - nu testeaza fiecare ramura – probleme la instructiunile if fara else

- acoperire la nivel de ramura – fiecare ramura a grafului e parcursa macar o data - generează date de test care testează cazurile când fiecare decizie este adevărată sau falsă – se mai numeste si “decision coverage” – DEZAVANTAJ – nu testeaza conditiile individuale ale fiecarei decizii

- **acoperire la nivel de conditie -** generează date de test astfel încât fiecare condiție individuală dintr-o decizie să ia atât valoarea adevărat cât și valoarea fals (dacă acest lucru este posibil) – AVANTAJE – se concentreaza asupra conditiilor individuale – DEZAVANTAJE – poate sa nu realizeze o acoperire la nivel de ramura – pentru a rezolva aceasta slabiciune se poate folosi testarea la nivel de decizie conditie

- **acoperire la nivel de conditie/decizie -** generează date de test astfel încât fiecare condiție individuală dintr-o decizie să ia atât valoarea adevărat cât și valoarea fals (dacă acest lucru este posibil) și fiecare decizie să ia atât valoarea adevărat cât și valoarea fals

- **acoperire MC/DC -** fiecare condiție individuală dintr-o decizie ia atât valoarea True cât și valoarea False - fiecare decizie ia atât valoarea True cât și valoarea False! ! 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

- acoperire la nivel de cale - generează date pentru executarea fiecărei căi măcar o singură dată - Problemă: în majoritatea situațiilor există un număr infinit (foarte mare) de căi - Soluție: Împărțirea căilor în clase de echivalență

Testare unitara cu jUnit: - @Before pt. initializari, @test public void numeFunctie() pt. Teste, assert pt verificare

Depanarea – folosind tiparirea – simplu de aplicat, nu necesita alte tool-uri – D: codul se complica, output-ul se complica, performanta uneori scade, e nevoie de recompilari repetate, exceptiile nu pot fi controlate usor etc. – **folosind log-urile –** fiecare clasa are asociat un obiect Logger – log-ul poate fi controlat prin program sau proprietati – D: codul se complica – Solutie: debugger – **debugger –** controlul executiei = poate opri executia la anumite locatii numite breakpoints – interpretor = poate executa instructiunile una cate una – inspectia starii programului = poate observa valoarea variabilelor, obiectelor sau a stivei de executie – schimbarea starii = poate schimba starea programului in timpul executiei – **breakpoint =** locatie in program care atunci cand este atinsa, opreste executia – strategie: se pune un BP la ultima linie unde stim ca starea e corecta – step into = executa instructiunea urmatoare, apoi se opreste – step over = considera un apel de metoda ca o instructiune – metode din bibliotecile Java sunt sarite – **inspectia starii programului –** cand executia e oprita, putem examina starea programului

Depanare sistematica – trebuie folosita deoarece: datele asociate unei probleme pot fi mari, programele pot avea mii de locatii de memorie si pot trece prin milioane de stari inainte de a se manifesta problema – **dependenta de date** - instrucțiunea B depinde cu ajutorul datelor (data dependent) de instrucțiunea A dacă, prin definiție: 1. A modifică o variabilă v citită de B și 2. există cel puțin o cale de execuție între A și B în care v nu este modificată “Rezultatul lui A influențează direct o variabilă citită de B” – **dependenta de control -** instrucțiunea B depinde prin control (control dependent) de instrucțiunea A dacă, prin definiție: 1. execuția lui B poate fi (potențial) controlată de A - “Rezultatul lui A poate influența dacă B e executată” - **dependență “înapoi”** - instrucțiunea B depinde în sens invers (backward dependent) de instrucțiunea A dacă, prin definiție: există o secvență de instrucțiuni A = A1, A2, … , An = B astfel încât: 1. pentru toți indicii i, Ai+1 este control-dependent sau data-dependent de Ai și 2. există cel puțin un indice i cu Ai+1 data-dependent de Ai - “Rezultatul lui A poate influența starea programului în B” - **Algoritm de localizarea sistematică a defectelor** - În I vom păstra un set de locații infectate (variabilă + contor de program) - În L păstrăm locația curentă într-o execuție care a eșuat 1. Fie L locația infectată raportată de eșec și I := {L} 2. Calculăm setul de instrucțiuni S care ar putea conține originea defectului: un nivel de dependență “înapoi” din L pe calea de execuție 3. Inspectăm locațiile L1, … , Ln scrise în S și dintre ele alegem într-o mulțime M $\subseteq$ { L1, … , Ln } pe cele infectate 4. În cazul în care M $\neq \emptyset$ (adică cel puțin un Li este infectat): 4.1 Fie I : = (I \ {L}) $\cup$ M (înlocuim L cu noii candidați din M) 4.2 Alegem noua locație L o locație aleatoare din I 4.3 Ne întoarcem la pasul 2. 5. L depinde doar de locații neinfectate, deci aici este locul de infectare! – **Simplificarea problemei intrarilor mari** – 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 – **Clasificarea defectelor -** 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

Design patterns - = solutii generale reutilizabile la probleme care apar frecvent in proiectare (OO) – un sablon e suficient de general pentru a fi aplicat in mai multe situatii, dar suficient de concret pentru a fi util in luarea deciziilor – **folositoare in urmatoarele feluri -** 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 șabloane: un mod de a explicita cum se fac lucrurile acolo – D: pot creste complexitatea si scadea performanta – **in inginerea software** – sunt solutii generale reutilizabile la probleme care apar frecvent in proiectare – **tipuri de sabloane –** arhitecturale (la nivelul arhitecturii ex. MVC, publish-subscribe)/de proiectare (la nivelul claselor/modulelor)/idiomuri (la nivelul limbajului ex. MVC, publish-subscrbe) – **sabloanele de proiectare –** 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 – **principii de baza –** programare folosind multe interfete – se prefera compozitia in loc de mostenire – se urmareste decuplarea – **sablonul creational SINGLETON –** asigura existenta unei singure instante pt. o clasa – ofera un punct global de acces la instanta – aceeasi instanta poate fi utilizata de oriunde fiind imposibil de a invoca direct constructorul de fiecare data – aplicabilitate: cand doar un obiect al unei clase e cerut, instanta este accesibila global, folosit in alte sabloane – consecinte: accesul e controlat la instanta, spatiu de adresare structurat – **sablonul c. ABSTRACT FACTORY -** oferă o interfață pentru crearea de familii de obiecte înrudite sau dependente fără a specifica clasele lor concrete – observatii: independent de modul in care produsele sunt create, compuse si reprezentate, produsele inrudite trebuie sa fie utilizate impreuna, pune la dispozitie doar interfata, nu si implementarea – consecinte: numele de clase de produse nu apar in cod familiiile de produse usor interschimbabile, cere consistenta intre produse – **sablonul c. BUILDER -** separă construcția unui obiect complex de 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 – **sablonul s. FAÇADE** - 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 **– sablonul co. OBSERVER -** presupunem o dependența 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 – consecinte: adaugarea de operatii se face usor, incalcare partiala a encapsularii – **ANTI-SABLOANE –** abstraction inversiopn, input kludge, interface bloat, magic pushbutton, race hazart, stovepipe system, anemic domain model etc.

Refactoring - 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 – schimbarile pot introduce noi defecte – nu sunt introduse noi functionalitati – imbunatateste structura, nu codul – nu trebuie sa introduca niveluri de complexitate inutile – **Semnale:** cod duplicat, metode lungi, clase mari, liste lungi de parametri, comunicare intensa intre obiecte – **optimizarea metodelor –** scop: simplificarea si cresterea coeziunii - impartirea unei metode in mai multe (metode care ret. O val. Si schimba devin 2 separate), adaugarea sau stergerea de parametri – **optimizarea claselor –** scop: cresterea coeziunii si reducerea cuplarii – mutarea metodelor – mutarea campurilor – extragerea de clase – inlocuirea valorilor de date cu obiecte

Metrici – dimensiune – complexitate (nivelul de dificultate in intelegerea unui modul) – **dimensiune –** LOC = line of code = linie de cod nevida, nu e comentariu – LOC corelata cu productivitatea, costul si calitatea – **complexitatea ciclomatica - =** indica nivelul de dificultate in intelegerea unui modul – M = e – n + 2*p, unde n = nr. De noduri e = numarul de arce p = numarul de componente conexe (pentru un modul este 1) – complexitatea ciclomatica a unui modul este numarul de decizii + 1 – aceasta metrica este corelata cu dimensiunea odulului si cu numarul de defecte – **variabile vii -** variabilă este vie de la prima până la ultima referențiere dintr-un modul, incluzând toate instrucțiunile intermediare - pentru o instrucțiune, numărul de variabile vii reprezintă o măsură a dificultății de înțelegere a acelei instrucțiuni – **anvergura -** numărul de instrucțiuni dintre 2 utilizări succesive ale unei variabile - dacă o variabilă este referențiată de n ori, atunci are n – 1 anverguri - anvergura medie este numărul mediu de instrucțiuni executabile dintre 2 referiri succesive ale unei variabile

Licente - = 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 apartin de categoria generala numita proprietate intelectuala - **patente -** 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 – **copyright** protejeaza codul sursa, nu si ideea – **licente -** comerciale (pe calculator sau utilizator) - shareware (acces limitat temporal sau funcțional) - freeware (gratuit) - open source: cod sursa disponibil si redistribuibil – **GPL -** cere ca orice modificări sau adaptări ale unui cod GPL, inclusiv software-ul care folosește biblioteci GPL, să fie sub licența GPL (natura virala) - 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 – **LGPL -** LGPL impune restricții copyleft pe cod, dar nu și pentru software-uri care doar folosesc codul respectiv

Management – proiect = set de activitati planificate definit prin inceput si sfarsit, obiectiv, domeniu de aplicare, buget, nerepetitiv – proiect de succes = executat in timpul dat, in bugetul disponibil si in parametrii de calitate ceruti – **managementul proiectlui –** consta in distributia si controlul bugetului, timpului si personalului – **manager de proiect:** planificare, organizare, constituire echipa, monitorizare, control, reprezentare – **etape ale unui proiect:** studiu de fezabilitate sau business case, planificare, executie – **studiu de fezabilitate:** 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 **– planificarea:** - Domeniul de aplicare și obiectivele - Identificarea infrastructurii de aplicare și obiectivelor - Analiza caracteristicilor proiectului - Identificarea activităților și livrabilelor - Estimatarea eforturilor pentru fiecare activitate