

# Curs 1

1. Ce este Ingineria Programării?

Metodologiile folosite în rezolvarea de proiecte mari rezolvate în echipă, presupun utilizarea principiilor ingineresti în analiza, dezvoltare, punere în funcțiune, testare, întreținere și retragere software. Ne dorim să obținem programe sigure, eficiente cu resursele pe care le avem.

2. Ce înseamnă un program bun?

Este sigur, oferă funcționalitățile cerute, ușor de modificat, nu irosește resurse hardware și e ușor de folosit.

3. Care ar putea fi dificultățile în ceea ce privește ingineria programării?




Sisteme vechi care trebuie întreținute, eterogenitate sisteme, presiunea de a livra repede.

4. Care sunt etapele dezvoltării unui program?

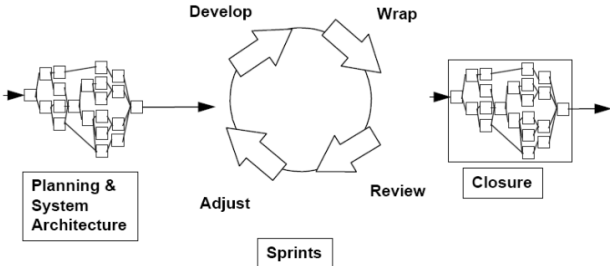
- Inginerie Cerințe
  - Ce se dorește de la program
  - Înregistrăm cerințele și încercăm să evităm situațiile în care informația este ambiguă
- Proiectare Arhitecturală
  - Programele mari sunt împărțite în module mai simple care sunt abordate individual
- Proiectare Detaliată
  - Proiectăm fiecare modul în detalii
- Implementare
  - Transpun proiectul planificat Într-un limbaj de programare
- Integrare Componentă
  - Big-Bang
  - Incremental
- Validare
  - ne asigurăm că programul face ce trebuie
- Verificare
  - ne asigurăm că nu există erori(stabil și funcționează corect)
- Întreținere
  - gestionăm problemele care pot apărea după livrare:
    - greșeli care trebuie reparate
    - cerințe noi
    - instruire utilizatori

5. Modele de Dezvoltare

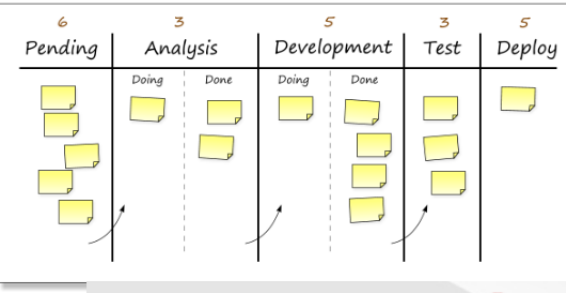
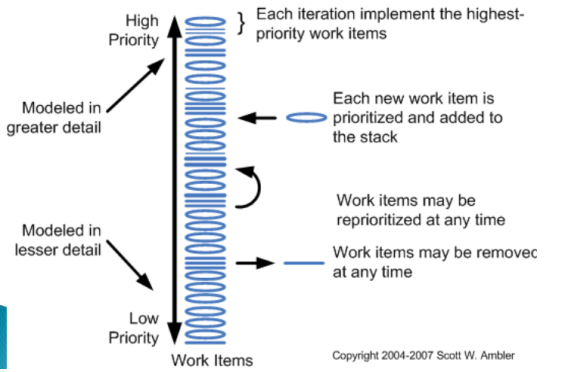
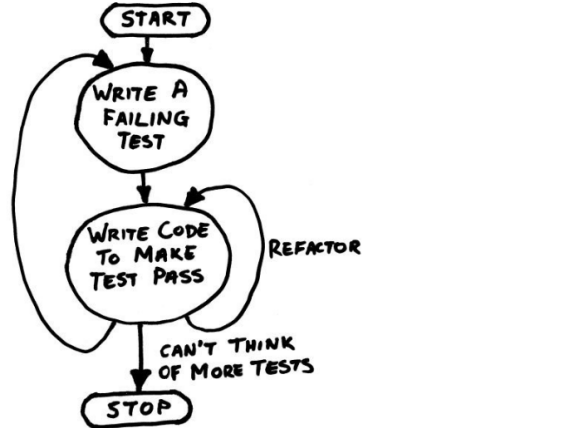
Nume	Principii	Avantaje	Dezavantaje
------	-----------	----------	-------------

Model în cascadă	<div><div><div><div>Ingineria Cerințelor</div><div>Proiectarea Arhitecturală</div><div>Proiectarea Detaliată</div><div>Implementare</div><div>Testarea Unităților</div><div>Testarea Sistemului</div><div>Acceptare</div></div><div><div>• Winston W. Royce in 1970</div></div></div><div></div><div>Fiecare pas din etapele de dezvoltare apare se face separat(unul după altul), nu se suprapun</div></div>	<ul style="list-style-type: none"><li>• sarcină complexă împărțită în pași mici</li><li>• ușor de controlat</li><li>• văd un rezultat după fiecare pas</li><li>• știm mereu unde ne aflăm și ce trebuie sa facem</li></ul>	<ul style="list-style-type: none"><li>• erorile se propagă</li><li>• nu există mecanisme de reparare erori</li></ul>
Cascadă cu întoarcere	<div><div><div><div>Ingineria Cerințelor</div><div>Proiectarea Arhitecturală</div><div>Proiectarea Detaliată</div><div>Implementare</div><div>Testarea Unităților</div><div>Testarea Sistemului</div><div>Acceptare</div></div><div>Modelul în Cascadă cu Întoarcere</div></div><div></div></div>	<ul style="list-style-type: none"><li>• pot să descoperim erorile de la pasul anterior</li></ul>	<ul style="list-style-type: none"><li>• dacă apar erori la pasul i + 2 de la pasul i, nu le pot remedia</li><li>• nu am un produs după fiecare pas</li></ul>
Spirală	<div><div><div><div>1 : pregătirea [take stock]</div><div>2 : gestiunea riscului [dealing with risk]</div><div>3 : dezvoltarea [development]</div><div>4 : planificarea următorului stadiu [planning]</div></div><div></div></div><div>La fiecare pas fac pașii de mai sus</div></div>	<ul style="list-style-type: none"><li>• introduc noțiunea de risc(deadline nerespectat, pleacă n angajat, firmă concurentă cu aceeași idee), păstrând avantajele de la cascadă</li></ul>	
Prototipare	<div><div>Tipuri de prototipuri:</div><div>De aruncat:<div>clarific cerințele</div>realizat rapid ca un exemplu, apoi programul se face de la 0</div><div>Evoluționar:<div>Construire produs final pe baza lui, prototipul este nucleul produsului final</div></div></div>	<ul style="list-style-type: none"><li>• specifică exact dorințele</li><li>• ieftin de gestionat</li><li>• întreținere ieftină</li><li>• facilitează instruire utilizatori</li></ul>	nu pot identifica toate problemele e într-un mediu artificial programatorii pot considera că munca lor e în van, din moment ce în cazul prototipurilor de aruncat, programul se scrie ulterior de la 0

<p><b>RUP(Rational Unified Process)</b></p>	<div data-bbox="270 197 789 533"> </div> <p><b>Se împarte în următoarele etape:</b>  <b>Inception:</b> validare costuri și buget, studiu de risc  <b>Elaboration:</b> analiză problemă, arhitectura  <b>Construction:</b> construire prima versiune  <b>Transition:</b> trec în producție</p>		
<p><b>V - Model</b></p>	<div data-bbox="270 768 789 1041"> </div> <p><b>V- Verificare și Validare</b>  pe de-o parte analizez cerințele, pe de altă parte le integrez și le verific  Astfel, minimizez riscurile, îmbunătățesc calitatea, reduc costurile, îmbunătățesc comunicare</p>	<ul style="list-style-type: none"> <li>● utilizatorii iau parte la dezvoltare</li> <li>● grup care se întâlnește odată pe an și controlează modificările</li> <li>● asistențe la fiecare etapă</li> </ul>	<ul style="list-style-type: none"> <li>● proiecte mici</li> <li>● nu asigură întreținerea</li> </ul>
<p><b>Extream Programming</b></p>	<div data-bbox="270 1253 596 1579"> </div> <p><b>Model modern, ușor, inspirat din RUP</b>  Dezvolatre nu înseamnă ierarhii, înseamnă colaborare între oamenii, astfel oamenii sunt încurajați să se afirme pentru a deveni programatori mai buni</p> <p>Nu mă focusez pe documentație, mă focusez pe scrierea de programe, astfel toată lumea are proiectul în minte și lucrează astfel</p> <p>Mereu am un reprezentant al clientului la dispoziție în caz că trebuie făcute clarificări</p>	<p><b>Fără lucru suplimentar</b></p> <p>Refactor constant și fără milă</p> <p>Integrare continuă</p>	

	<p><b>Programare în echipe care se schimbă la finalul iterației</b></p> <p><b>Planificare:</b>          Artefact: user stories          Eveniment: daily, planificare iterație          Măsoară performanța cu unitatea vitezei          Proiectul e împărțit în iterații</p> <p><b>Coding:</b>          clientul e mereu disponibil          scriu unit teste mai întâi          integraz constant          pair program          optimizare la final</p> <p><b>Design</b>          simplu          fără funcționalități în faze incipiente          refactor constant</p> <p><b>Testing</b>          tot codul are unit test, pe care trebuie să le treacă înainte de release  <b>teste pentru buguri</b></p>		
Agile	<p>Oferire continuă de software(săptămânal)          Progresul se măsoară în funcționalități          Accept Modificări târzii          discuțiile face-to-face sunt importante</p>	<ul style="list-style-type: none"> <li>● <b>nu am documentație</b></li> <li>● se lucrează cu senior level</li> <li>● greu de negociat</li> </ul>	<ul style="list-style-type: none"> <li>● <b>crește productivitatea</b></li> </ul>
Scrum	<p><b>SCRUM Methodology</b></p>  <p>Clientul e parte din echipa de dezvoltare</p> <p>Discuții zilnice</p>	<ul style="list-style-type: none"> <li>● Verificări și validări intermediare</li> <li>● Transparență în planificare și dezvoltare</li> <li>● Nu ascund probleme</li> <li>● Muncă eficientă nu multă</li> </ul> <p>ARTEFACTE:</p> <ul style="list-style-type: none"> <li>● product backlog             <ul style="list-style-type: none"> <li>○ tot ce vreau de la aplicație</li> </ul> </li> <li>● sprint backlog             <ul style="list-style-type: none"> <li>○ tot ce plănuiesc să fac într-un sprint</li> </ul> </li> <li>● increment             <ul style="list-style-type: none"> <li>○ produs final</li> </ul> </li> </ul> <p>ROLURI:</p>	

		<ul style="list-style-type: none"><li>● scrum master<ul style="list-style-type: none"><li>○ ajuta echipa sa se organizeze</li><li>○ motiveaza echipa</li><li>○ preda valorile scrum</li><li>○ elimina impedimentele care pot opri progresul</li></ul></li><li>● product owner<ul style="list-style-type: none"><li>○ vocea clientului, responsabil de calitatea produsului și de product backlog</li></ul></li><li>● dev team</li><li>● livrează un produs funcțional</li><li>● 3- 9 oameni</li><li>● se autoorganizeaza</li></ul> <p>EVENIMENTE:</p> <ul style="list-style-type: none"><li>● sprint planning<ul style="list-style-type: none"><li>○ pregătește product backlog și ce fac în Sprint</li></ul></li><li>● daily scrum<ul style="list-style-type: none"><li>○ se răspunde la următoarele întrebări<ul style="list-style-type: none"><li>○ ce am făcut ieri</li><li>○ ce fac azi</li><li>○ ce probleme au apărut</li></ul></li></ul></li><li>● retrospectiva sprint<ul style="list-style-type: none"><li>○ ce a mers bine</li><li>○ ce poate și îmbunătăți</li><li>○ acțiuni</li></ul></li><li>● review sprint<ul style="list-style-type: none"><li>○ vad ce am reușit sa fac în sprint</li><li>○ participa toata echipa</li></ul></li></ul>	
Lean(sprijin)	Elimin lucrurile nefolositoare <b>Amplific învățarea</b> Membrii echipelor au responsabilități Proiect integru Mă uit în ansamblu la proiect		

<b>Kanban</b>	<div></div> <p>Productie JIT(Just-In-Time) Controlează evenimentele din punct de vedere a producției Știu când să produc și să livrez în funcție de ce am în plan Vizualizez evenimentele într-un board</p>	<ul style="list-style-type: none"><li>Management al sarcinilor fără să încarc membrii echipei</li></ul>	
<b>Model Driven Development</b>	<div></div> <p>Implementare rapidă, eficientă și la cost minim Creez modele înainte să scriu cod e.g. Object Management Group(OMG), Model Driven Architecture (MDA)</p>		
<b>Test Driven Development</b>	<div></div>		

Curs 2

6. Analiza Cerințelor  
Ce este? Procesul de înțelegere ale nevoilor clientului și așteptări acestuia. Comportament și caracteristici sistem  
Cine? Project Manager, Program Manager, Business Analyst

**De ce?** Pentru a ne asigura că am înțeles ce se dorește de la program

**Pași:**

**1. Stabilire limite aplicație**

Care este scopul aplicației și cât de multe poate face

**2. Găsire Client**

Cine este utilizatorul final al aplicației, ca să știm pe cine să întrebăm pentru clarificări

**3. Identificare Cerințe**

Colectez cerințele și identific ce îmi doresc utilizatorii de la aplicație

Probleme: ambiguități, inconsistență, date insuficiente, modificări cerințe târziu

Cine: o persoană care poate să comunice cu grupuri diferite de persoane și are și cunoștințe tehnice

Metode: interviuri, documentație existentă, prototipuri, diagrama use - case, interfețe utilizator

**4. Analiză Cerințe**

Analiză cu:

raționament automat

privire critică

verificare consistență

raționament analogic și pe exemple

**5. Specificare Cerințe**

**Clar, Neambiguu**

**Scriere Document**, circulă între toate persoanele implicate(client, grupuri de utilizatori, echipe de dev și testare)

**Scop Document:**

validare cerințe de către client

negociere contract

bază proiectare sistem

sursă pentru scenarii de testare

Aș putea avea două documente:

**Cerințe utilizator:** scrise clar, fără noțiuni tehnice

**Cerințe sistem:** model matematic sau programatic, informații legate de implementare(format date, adresă server, etc)

**Tipuri de Cerințe:**

**Cerințe Utilizator**

unde va fi folosit, eficient, durată de viață

**Cerințe funcționale**

cum se fac anume calcule

**Cerințe de performanță**

modul în care sunt apelate funcțiile cantitativ

**Constrângeri**

ex. nu introduc două persoane simultan în baza de date

**6. Gestionare Cerințe**

**Verificare și validare acestora**

Eliminare erori, ambiguități

Requirement Quality	Example of bad requirement	Example of good requirement
Atomic	*Students will be able to enroll to undergraduate and post graduate courses	*Students will be able to enroll to undergraduate courses *Students will be able to enroll to post-graduate courses
Uniquely identified	1– Students will be able to enroll to undergraduate courses1– Students will be able to enroll to post-graduate courses	1.Course Enrolment 2.Students will be able to enroll to undergraduate courses 3.Students will be able to enroll to post-graduate courses
Complete	A professor user will log into the system by providing his username, password, and other relevant information	A professor user will log into the system by providing his username, password and department code
Consistent and unambiguous	A student will have either undergraduate courses or post-graduate courses but not both. Some courses will be open to both under-graduate and post-graduate	A student will have either under-graduate or post graduates but not both
Traceable	Maintain student information-mapped to BRD req.ID?	Maintain student information-Mapped to BRD req ID 4.1
Prioritized	Registered student-Priority 1Maintain User Information-Priority 1Enroll courses-Priority 1View Report Card-Priority 1	Register Student-Priority 1Maintain User Information-Priority 2Enroll courses-Priority 1View Report Card-Priority3
Testable	Each page of the system will load in an acceptable time-frame	Register student and enrol courses pages of the system will load within 5 seconds

7. Ce sunt scenariile de utilizare?

Folosește Actori:

- umani
- software
- hardware

Folosește Scenarii(Use Case)

- interacțiune actor cu sistem
- cum reacționează sistemul
- rezultat

8. Despre USE CASE:

Descriere mulțime de secvențe de acțiuni

Funcționalități program

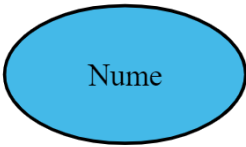
Nu precizez cum implementez o funcționalitate

Tipuri:

- Pe scurt - cazul de succes
- Cazul - ce se face în caz că apare ceva
- Detaliat - prezintă toate situațiile posibile

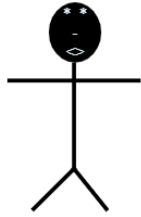
Conține:

- Use Case - funcționalități
  - Notăție



- Actori = entități externe cu care interacționează sistemul
  - Notăție





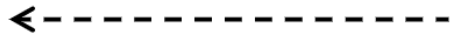
- Relații
  - Asociere: Actor- UseCase, UseCase - UseCase
    - Actor - UseCase: actorul inițiază execuția cazului de utilizare sau oferă funcționalitate
    - UseCase - UseCase: transfer date, trimitere mesaje



- Generalizare: Actor - Actor, UseCase - UseCase
  - Element Particular altui element



- Dependență: UseCase - UseCase(<<include>>, <<extend>>)
  - <<include>> folosește comportamentul unui alt UseCase
  - <<extend>> extinde comportamentul altui use case



## CURS 3

### 1. Ce este Modelarea și de ce se face?

Model: simplificare realitate

De ce modelăm?

înțelegem mai bine ce e de făcut

ne concentrăm pe un anume aspect la un moment dat

Scop?

Vizualizare sistem

Specificare structură și comportament

Oferire șablon

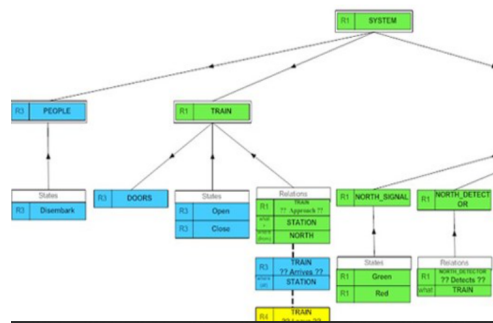
Documentare decizii

### 2. Limbaje de modelare:

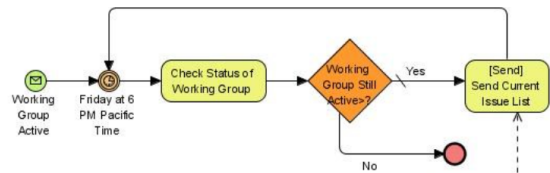
Limbaj artificial care poate fi folosit să exprime informații sau cunoaștere sau sisteme

**Tipuri:**

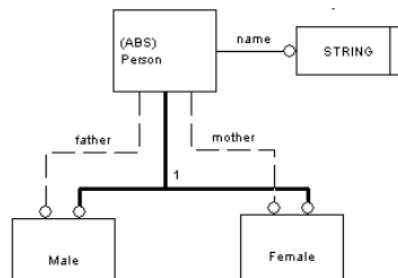
- **Limbaje Grafice:**
  - arbori comportamentali



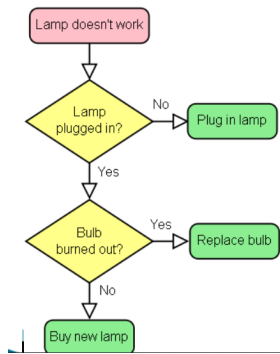
modelare procese business



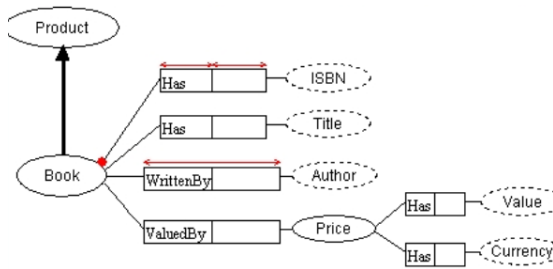
EXPRESS



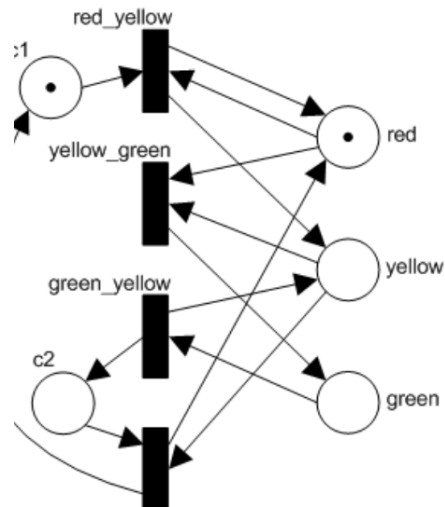
flowchart



ORM



rețele petri

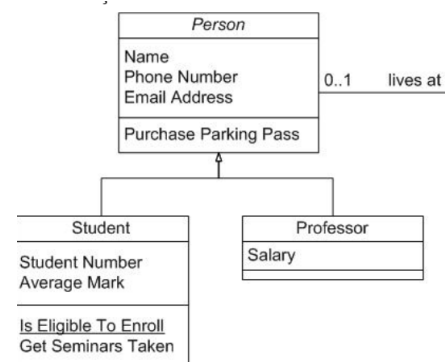


## ○ UML( UNIFIED MODELING LANGUAGE)

- succesori cele mai bune 3 limbaje :
  - Booch
  - OMT
  - OOSE
- Recomandări:
  - diagramele să nu fie nici prea complicate, dar nici simple
  - nume sugestive elementelor componente
  - să nu intersectăm liniile
  - pot realiza mai multe diagrame de același tip
- TIPURI DE DIAGrame

### ● Diagrame de Structura

#### ○ Diagrame de Clasă



Scop:

- modelează vocabularul sistemului
- conexiuni semantice sau interacțiuni
- modelează structură program

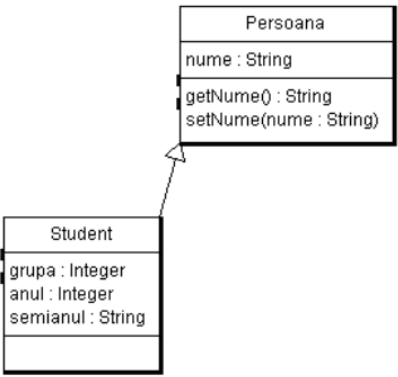
Conține:

- clase/interfețe
  - nume
  - attribute: proprietăți
  - metode: implementare serviciu

- obiecte
- relații(Asociere, Agregare, Generalizare, Dependență)

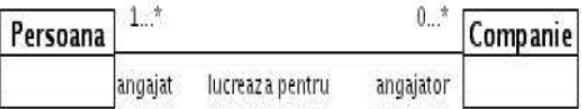
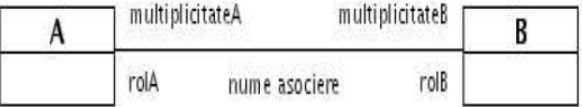
- **GENERALIZARE**

- moștenire clase
- relație de tip **IS A**

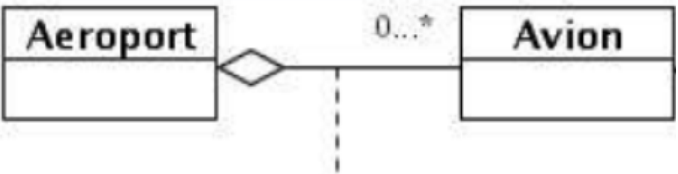


- **ASOCIERE**

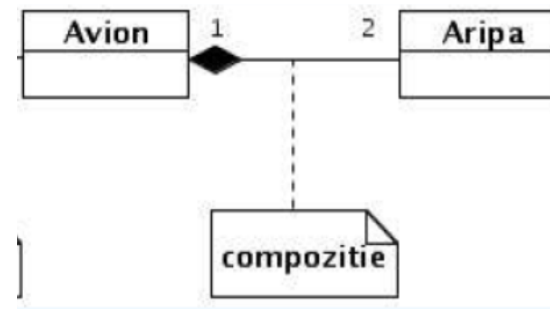
- conexiune semantică sau interacțiune obiecte
- **Elemente:**
  - nume
  - multiplicitate = câte instanțe sunt implicate



- **AGREGARE**

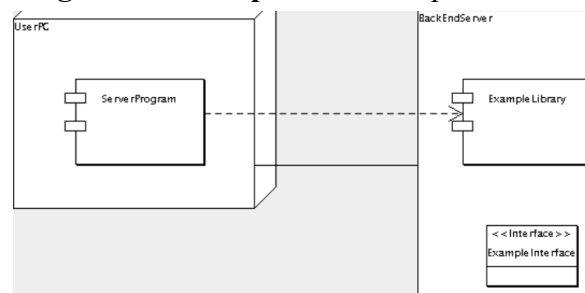


- caz particular asociere
- parte - întreb
- specific doar multiplicitatea
- obiect format din mai multe componente
- Agregare mai puternică = compoziție(HAS-A)

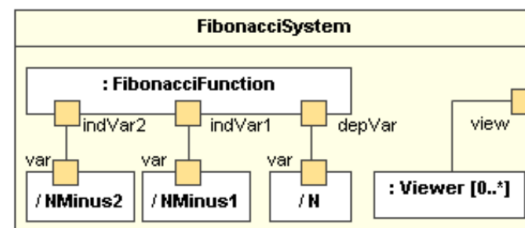


- aripa nu există independent de avion

- **Diagramă de componente:** componente sistem și legături

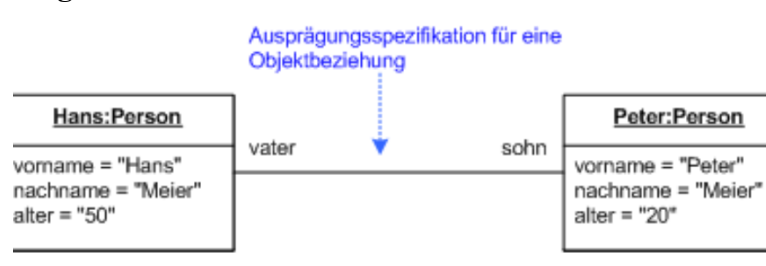


- **Diagramă structură compozită:** structură internă



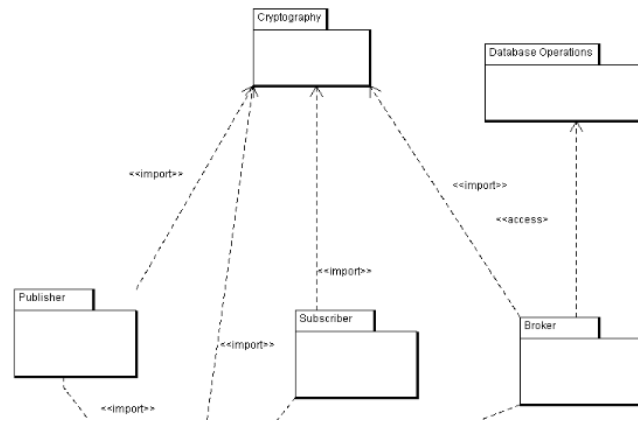
- **Deployment:** hardware
  - componente hardware
    - server web
    - server aplicații
    - server baze de date
  - componente software
    - aplicație web
    - bază de date
  - cum sunt conectate cele două

- **Diagramă de obiecte:** structură sistem într-un obiect



- **Diagramă pachete:** împărțire sistem în pachete

- împart un sistem mare în subsisteme mai mici
- dezvoltare paralelă
- definire interfețe clare între pachete
- Pachet
  - container logic pentru elemente între care se stabilesc legături
  - namespace
  - poate conține subpachete
- Relații
  - <<access>> = import privat
  - <<import>> = import public
- Se realizează în cadrul diagramelor de clasă



## ● Diagrame COMPORTAMENTALE

- **diagrame activitate:** prezentare business și flux activități
  - dinamică proces sau operație
  - pentru
    - clasă
    - pachet
    - 
    - implementare operații
- **diagrame de stare:** prezentare stările obiectelor
  - stări și tranziții
- **diagramă use case:** funcționalități sistem folosind actori, use case, dependențe între ele

## ● Diagramă Interacțiune

- **Diagramă comunicare:** interacțiune obiecte
- **Diagramă de secvență:** prezintă modul în care obiectele comunică între ele
  - secvența acțiunilor care au loc în sistem, invocare metode și ordinea de timp
  - bidimensională
    - vertical: viață obiect
    - orizontală: secvența creării sau invocărilor
- **Diagramă de Colaborare**
  - organizarea structurală a obiectelor care participă la interacțiune
  - ramificări complexe

- conține:
  - obiecte, clase, actori
  - legături
  - mesaje

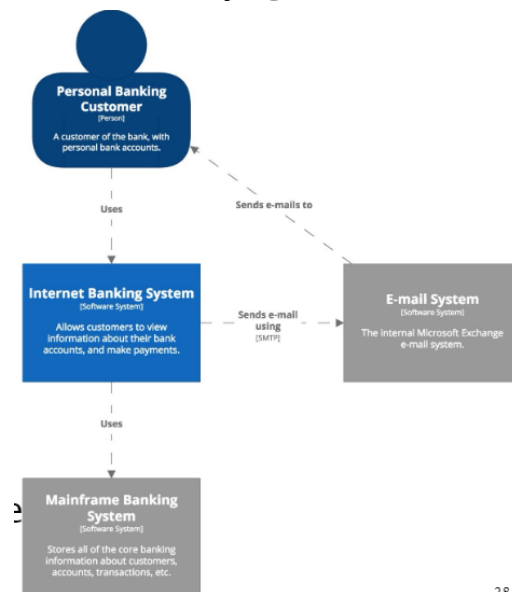
- **Limbaje Specifice:**
  - modelare algebrica (AML)
  - modelare domenii(DSL)
  - modelare arhitecturi specifice
  - modelare obiecte
  - modelare realități virtuale

### 3. Ce este C4 Model?

Nivele diferite de abstractizare

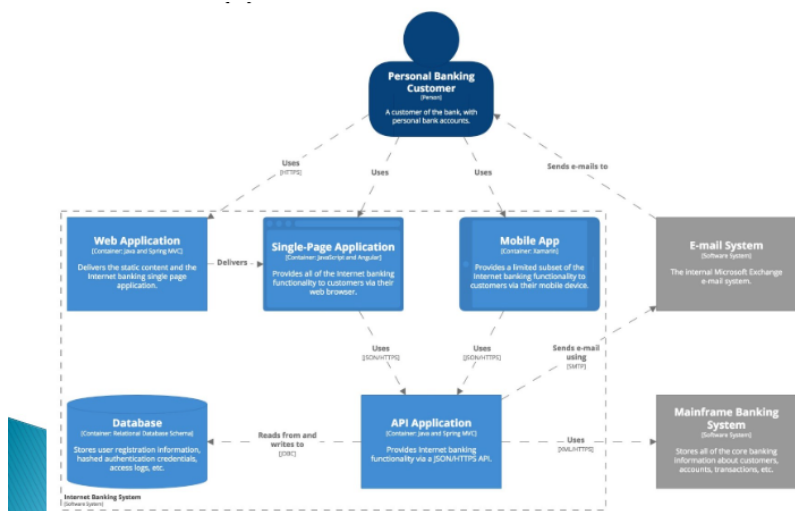
Tipuri:

- **Context**
  - arată sistemul construit și cum se încadrează în lume(cine îl folosește, oameni sau sisteme software)
  - Culori: Există deja: gri, Trebuie construit: blue



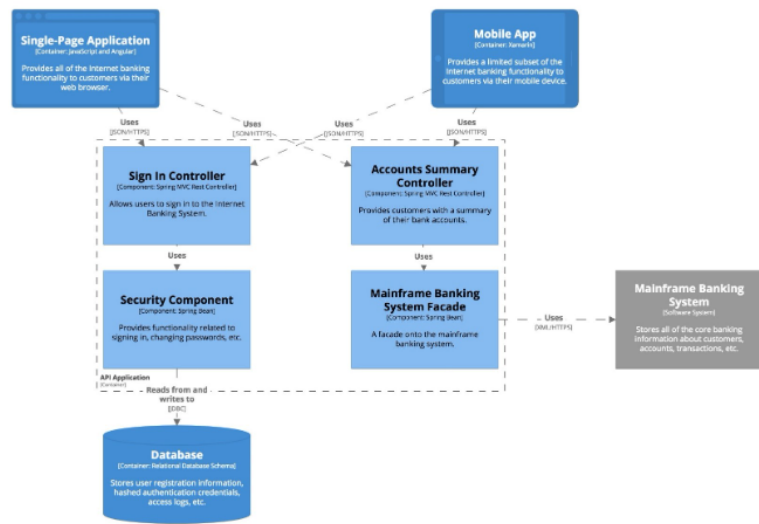
28

- **Container**
  - Face zoom in software și arată containerele din aplicație



- **Components**

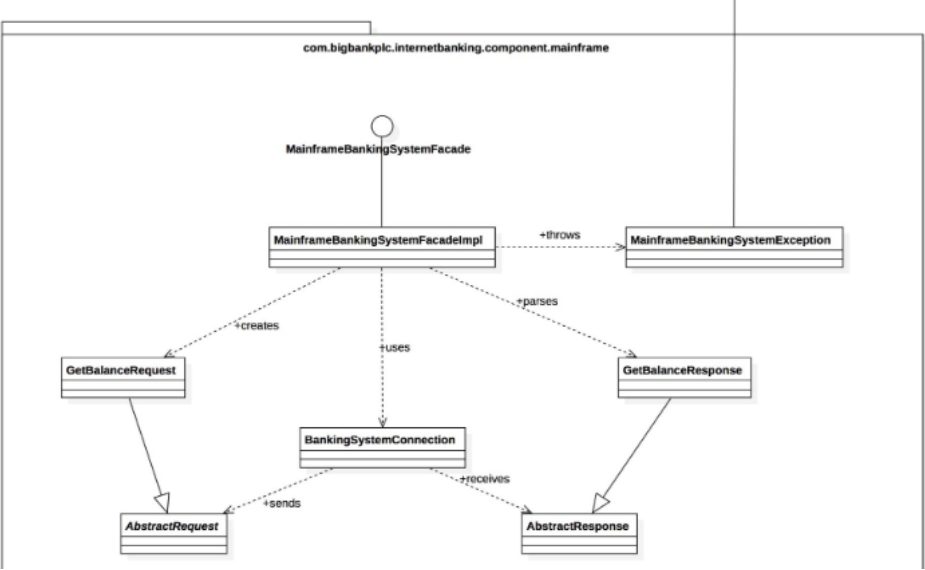
- Merge pe fiecare container și arată componentele din el



- **Code**

- merge pe fiecare componentă și arată cum e implementată





3. Forward și Reverse Engineering

**Forward:** de la cerințe la implementare

**Doc**

CURS 5

SOLID

Nume	Definiție	Probleme	Soluții
S - Single Responsibility	Fiecare obiecte are o responsailitate unică. Low coupling - legături puține între clase Strong Cohesion - elementele încapsulate în clasă sunt strâns legate între ele	Obiecte care pot să se afișeze singure Obiecte care se salvează și restaurează singure	Interfețe multiple cu responsabilități clare și distincte
O - Open/Close	Deschis la extindere, dar închis la Modificare. Open - Pot să adaug funcționalități Close to Modification - nu modific entitățile deja existente	Modificări în cascadă între module care pot conduce la apariția de bug-uri și drept urmare trebuie să testez totul  Logică care depinde de condiții	Ne bazăm pe abstractizări, astfel putem adauga clase noi cu funcționalități noi, fără să le afectăm pe cele deja existente
L - Liskov principle of substitution	Dacă face ca o rață, arată ca o rață, dar are nevoie de baterii probabil nu e o rață  O clasă copil trebuie să poate înlocui clasa părinte în orice situație.  O clasă copil nu trebuie să modifice sau să elimine comportamentele clasei de bază	Probleme legate de polymorphism(nu pot înlocui peste tot părintele cu copilul drept urmare pot apărea probleme)  Metode care trebuie suprascrise implementate  Verificare tip la apel de funcție	Refactor clase

I - Interface Segregation	Clienții nu trebuie să depindă de metode pe care nu le folosesc.  Folosesc interfețe mici și cu funcționalități clar definite	Folosesc doar o parte din clasă Metode neimplementate	Separ interfața în interfețe mai mici și focusate pe taskuri exacte
D - Dependency Inversion	Nivele superioare nu trebuie să depindă de nivele inferioare. Ambele folosesc abstractizări. Abstractizarea nu depinde de detalii.  Clasele declară lucrurile de care au nevoie și le primesc în constructor	utilizare de new, metode statice	Dependency Injection, trimit obiectul în constructor(‘Don’t call us, we’ll call you!’). Legat modulele între ele la Run time  Container Inversion of Control

Altele:

**DRY(Don’t repeat Yourself)**

- toate informațiile trebuie să fie reprezentate unic în sistem

**YAGNI(You Ain’t Gonna Need It)**

- adaug funcționalități doar când chiar am nevoie de ele

**Keep it Simple, Stupid**

- sisteme simple

**GRASP(General Responsibility Assignment Software Patterns)**

- alocare de responsabilități claselor în cel mai elegant mod posibil
- Ce responsabilități pot avea clasele sau obiectele?
  - Crearea unui obiect
  - Realizare de calcule
  - Inițializare acțiuni
  - Controlare și coordonare activități
  - Să cunoască: atribute, obiecte, lucruri pe care le poate face sau le poate apela
- Exemple de Patterns din GRASP:
  - INFORMATION EXPERT**
    - cărei clase îi aloc un anume comportament
    - Răspuns: clasei care are cunoștințele necesare pentru a realiza acea operație
  - Creator**
    - cine crează o instanță?
    - Asignez lui A responsabilitatea să creeze instanțe din B dacă
      - A agregă obiecte de tip B
      - A conține obiecte de tip B
      - A folosește obiecte de tip B
      - A are datele care trebuie trimise la construcția unui obiect din B
  - Low Coupling(Cuplaj redus)**
    - o clasă nu trebuie să depinde de multe alte clase
    - LEGEA lui Demeter:
      - o metodă a unui obiect apelează doar metode:
        - care îi aparțin
        - unui parametru a metodei
        - unui obiect creat sau pe care îl conține

- **High Cohesion**
  - cât de focalizate sunt responsabilitățile unei clase
- **Controller**
  - cine tratează elementele generate de un actor
  - un controller recepționează evenimente

## Curs 6

1. Ce este un design pattern?  
Soluții pentru probleme care apar frecvent în proiectarea software  
Denumirea soluțiilor facilitează comunicarea
2. Cum e definit un Design Pattern?  
Nume(contribuie la comunicarea eficientă între membrii unei echipe), Problemă(Când aplic un anumit pattern, condiții care trebuie îndeplinite înainte), Soluție(design, relații, responsabilități, colaborări), Consecințe(avantaje și dezavantaje)

### Creational Patterns(SMBPA)

Abstractizeaza procesul de creare a obiectelor

Nume	Problemă	Soluție	Când?	Avantaje	Dezavantaje
Singleton(POO)	Vreau o singură instanță a clasei la care să am acces global.(ex. fișier, database)	Constructor Default Private = nu mai pot crea instanțe ale clasei de dinafară Instanță statică în clasă, creată de o metodă statică care se comportă ca un constructor și care fie returnează instanța deja creată, fie creează una dacă nu există	Când vreau o instanță unică disponibilă tuturor(ec. bază de date)  Control strict al variabilelor globale	Sunt sigură că am doar o instanță  Punct de acces global la aceea instanță	Violează Single Responsibility, clasa se ocupă de două probleme  Nu e thread safe  Greu de testat
Factory Method	Interfață de creare obiecte, dar subclasa decide tipul clasei instantiate	Înlocuiesc Construcția directă de obiecte cu new, cu apelarea la factory method. Creatorul o sa îmi returneze un produs(obiectele care pot fi returnare trebuie să aibe o bază comună)	Nu știu tipurile și dependențele exacte cu care vom lucra  Reutilizare de obiecte	Evit cuplarea puternică între creator și produse  Single Responsibility: separ crearea de funcționalitate  Open/Close: pot să introduc cu ușurință tipuri noi de produse	Cod Complex
Builder	Separea Construcției unui obiect complex de reprezentarea acestuia.	Extrag partea de construcție a obiectului într-o clasă separată în	Vreau să scap de constructorul telescopic	Construiesc obiecte pas cu pas	Cod Complex

	Evitarea unui constructor cu un număr mare de condiții	care organizez procesul de construcție pe pași. Builder poate fi o clasă abstractă și să am mai multe clase ce o moștenesc și fac pașii de construcție în moduri diferite. Având un builder, apelez pașii pe care doresc să îi realizez și returnez obiectul creat	Vreau reprezentări diferite ale aceluiași obiect	Reutilizez aceeași pași de construcție pentru mai multe metode  Single Responsibility	
Prototype	Copiez obiecte fără să fiu dependent de clasa de care aparțin.  Să zicem să vreau să fac un copiez un obiect, aş putea să merg din câmpurile acestuia și să le pun într-un obiect nou, dar e posibil să nu am acces la toate câmpurile(câmpuri private)	Procesul de clonare devine responsabilitatea obiectului clonat. Toate obiectele care permit clonarea vor avea o interfață Cloneable. Astfel pentru a copia un obiect nu va trebuie să am o relație cu clasa din care provine acesta, de asemenea voi avea acces și la câmpurile private astfel	Nu vreau să depind de clasa obiectului copiat	Clonez obiecte fără să am legături cu clasa lor  Elimin repetarea inițializărilor  Produc obiecte complexe mai rapid	Obiecte care au referințe circulare
Abstract Factory(POO )	Produc familii de obiecte fără să le specific clasa	Interfață pentru fiecare tip de produs pe care îl pot crea și Abstract Factory care conține metodele necesare pentru a crea toate obiectele dintr-o familie. Apoi Factory concrete pentru fiecare variație a obiectelor	Lucrez cu familii de obiecte, dar nu vreau să depind de clasa concretă a acestora	Produse Compatibile din același factory  S - crearea separat de funcționalitate  O - pot adăuga familii noi de produse fără a modifica codul celor vechi	Cod complex
Lazy Initialization	Vreau să Amân realizarea unui proces mai dificil până când chiar am nevoie de el.	Folosesc un flag care îmi zice dacă e ready sau nu.		salvez memorie	consumă timp dacă am nevoie de multe instanțe
Object Pool	Crearea de obiecte e costisitoare.	Refolosesc obiecte deja construite salvate într-un object pool	Conexiuni la baze de date	Salvez resurse având mai puține instanțe	Greu de făcut management la pool

Structurale(ABCDFFP)

Cum sunt clasele compuse pentru a forma structuri mai mari

Nume	Problemă	Soluție	Când?	Avantaje	Dezavantaje
Adapter	Comunicare între obiecte cu interfețe incompatibile	Creez un adapter, care convertește interfața unui obiect în ceva de poate fi înțeles de cel cu care comunică	Vreau să utilizez clase deja existente, dar care nu au o interfață comună	Single Responsibility - Separ interfața de conversia de date Open Close - introduc adapters noi fără să le modific pe ele existente	Cod Complex
Bridge	Cum fac să dezvolt o clasă care se poate dezvolta pe mai multe componente (ex cub + culori)	În loc să am ambele dimensiuni pe care pot extinde clasa, le separ în două ierarhii separate care se dezvoltă individual și comunică prin compunere. Sunt introduse conceptele de: abstractizare(interfață) și implementare(cel care face munca delegată de abstractizare)	Vreau să divid un monolit în funcționalități.  Vreau să dezvolt clasa pe mai multe dimensiuni  Vreau să modific implementarea la RUNTIME în funcție de abstractizare	Single Responsibility  Open/Close  Clientul nu are acces la detaliile de implementare	Cod complex( clase suplimentare)
Composite(și la POO)	Am nevoie de structuri care pot fi reprezentate ca un arbore(ex. produse și cutii în care sunt puse acestea)	Folosesc Produse și Cutiile(objecte simple și compuse) printr-o interfață comună, astfel sunt tratate la fel, dar fiecare oferă o implementare diferită pentru anumite metode	Am de implementat o structură arborescentă de obiecte  Vreau să tratez obiectele simple și cele compuse uniform	Folosesc polimorfismul pentru a lucra cu o structură de tip arbore  Open/Close	Dificil să creez o interfață care să fie ok pentru toate clasele fără să fac generalizări
Decorator	Am nevoie să adaug funcționalități pentru un program la RUN time, fără să modific comportamentul clasei sau fără să apelez la moștenire dacă nu este nevoie neapărat	Am o interfață decodator care primește obiectul îi adaugă în funcție de cerere noi funcționalități și îl returnează înapoi.	Vreau să adaug comportamente la run time fără să modific codul claselor  Când e ciudat să extind comportamentul prin moștenire	Single Responsibility  Adaug și elimin comportamente la run time  Nu am nevoie de subclase noi ale obiectului	Greu de eliminat un wrapper  Nu pot determina ordinea în care au fost aplicate
Facade	Vreau să folosesc doar anumite funcționalități ale unei clase complexe	Ofer o interfață simple pentru un sistem complex, care conține funcționalități limitate de care am nevoie	Am nevoie de o interfață directă și limitată la un subsistem complex.  Vreau să structurez sistemul	Izolez codul de complexitatea subsistemului	Facade ar putea să devină GOD OBJECT, adică să știe prea multe despre toate clasele cu care comunică

			pe layers		
<b>Flyweight</b>	Am nevoie să folosesc multe obiecte asemănătoare	Iau anumite caracteristici care sunt la fel pentru toate obiectele într-o clasă separată	Folosesc multe obiecte care nu au loc în RAM	Salvez RAM	Consumul de RAM e înlocuit cu consum de CPU, datele din contextul obiectului trebuie recalculate  Cod complex
<b>Proxy</b>	Am un obiect complex, care consumă multe resurse și nu e viabil să fie activ tot timpul	Creez un Proxy cu aceeași interfață ca obiectul original, clientul comunică cu Proxy, care accesează ulterior obiectul original, de asemenea pot să adaug verificări, funcții noi, înainte de utilizarea obiectului de bază	Lazy initialization  Vreau să am control al accesului  Executare de remote services  Logging	Controlez serviciile unui obiect fără  Înștiințarea clientului  Control al ciclului de viață al obiectului  Open/Close	Cod complex  Răspuns delayed

## Comportamentale(CCII MMO SSTV)

### Aspecte ale programului care se pot modifica

ne	Problemă	Soluție	Cand	Avantaje	Dezavantaje
<b>Chain of control</b>	Vreau să fac consecutiva mai multe chestii asupra unui request	Separ funcțiile care trebuie aplicate în mai multe clase numite handler. Fiecare handler are o referință la următorul handler din chain care trebuie apelat sau poate decide să se oprească	Vreau să procesez request-uri în moduri diferite, pe care nu le știu de dinainte.	controlez ordinea în care se face handler la request  Single Responsibility <ul style="list-style-type: none"> <li>- clasele care invocă sunt separate de cele care realizează operațiile</li> </ul> Open/Close <ul style="list-style-type: none"> <li>- introduc handlere noi fără să le modific pe cele deja existente</li> </ul>	Unhandled requests
<b>Command</b>	Vreau să definesc funcționalități pentru anumite evenimente	Înglobează informația trimisă de GUI în comenzi și comanda e trimisă mai departe pentru a fi gestionată	Vreau să transform un apel al unei funcții într-un obiect  Când vreau să programez operațiile	Single Responsibility: <ul style="list-style-type: none"> <li>- decuplez clasele care invocă de clasele care realizează operații</li> </ul> Open/Close <ul style="list-style-type: none"> <li>- introduc comenzi noi</li> </ul>	Cod complex prin introducerea unui layer într cel care trimite și cel care primește

				Implement undo/redo Comenzi compuse din mai multe comenzi simple	
<b>Iterator</b>	Vreau să parcurg o colecție independent de obiectele pe care le conține și de tipul colecției	Scot funcția de traversare a colecțiilor în obiecte separate numite Iteratori, iteratorii știu poziția curentă și câte elemente mai sunt de parcurs.	Colecția folosește o structură de date complexă pe care vreau să o ascund  Reduc cod duplicat utilizat pentru iterare	Single Responsibility Open/Close  Pot itera în paralel pe aceeași colecție	Poate fi overkill pentru colecții simple
<b>nterpreter</b>	Am o limbă și vreau să definesc o gramatică pentru aceasta				
<b>Memento</b>	Cum pot să salvez starea internă a unui obiect pentru mecanisme precum cel de savepoint, undo, etc, având în vedere că poate exista situația în care obiectul are câmpuri private care nu pot fi accesate din exterior	Crearea unui memento devine responsabilitatea clasei care deține obiectul, deoarece are acces la starea internă a acestuia	Vreau să salvez starea unui obiect pentru a reveni la ea	Creare snapshot	Consumă RAM  Limbajele dinamice(Python, PHP), nu garantează că se va menține starea
<b>Mediator</b>	Cum reprezint interacțiunea între anumite obiecte fără a le face dependente unul de altul	Creez o clasă Mediator care se ocupă de comunicare între cele două obiecte și redirecționează apelurile la componentele corespunzătoare.	Vreau să reduc cuplajul între clase  Vreau să refolosesc o componentă	Single Responsibility  Open/Close  Reduc cuplajul între componente  Reutilizez componente	God Object - știe prea multe lucruri despre toate clasele
<b>Observer</b>	Vreau să avertizez anumite clase dacă apar modificări într-o altă clasă, pentru a păstra de exemplu consistența operațiilor	Obiectul urmărit se numește subject și publisher deoarece îi anunță pe restul. Cei care urmăresc se numesc subscriberi. Publisherul are o listă de subscriberi și în cazul unor modificări apelează metoda update a subscriberilor	Schimbările într-un obiect se reflectă în alt obiect	Open/Close  Relații între obiecte la Run Time	Notificarea se face într-o ordine random
<b>State</b>	Situații în care	Creez câte o clasă nouă	Obiecte care se	Single Responsibility	Poate fi overkill dacă am

	comportamentul unei clase trebuie să se modifice în funcție de starea în care se află acesta	în care definesc comportamentul în funcție de stare. Obiectul de bază se numește context și păstrează o referință la un obiect asociat unei stări	comportă diferit în funcție de stare și există multe variante de stări.  Elimin condițiile care se ocupă de modificare stării obiectelor	Open/Close  Elimin expresii condiționale lungi și complexe	puține stări
Strategy	Pentru o acțiune am mai mulți algoritmi pe care vreau să îi interschimb la run time	Scot algoritmi în clase separate numite strategii.  Obiectul original are o referință la o strategie, strategia e trimisă de utilizator	Vreau să schimb algoritmi folosiți la run time  Elim clasele la fel care diferă doar prin faptul că au o anume funcționalitate diferită  Vreau să separ logica de business, de metode de implementare	Schimb algoritmi la run time  Izolez detaliile de implementare de codul care le folosește  Open/Close	Nu e util dacă am puțini algoritmi care se modifică rar  Clienții trebuie să fie capabili să aleagă o strategie
Template Method	Vreau să am o funcționalitate de bază, dar care să fie implementată diferit în fiecare subclasă	Funcționalitatea e împărțită în pași și subclasele oferă implementări diferite pentru fiecare pas.  Există pași care pot avea o implementare default întrucât sunt folosiți de către toate metodele	Vreau să modific doar anumite părți dintr-un algoritm.  Am mai multe clase care folosesc același algoritm cu puține modificări	Elimin codul duplicat  Clientul suprascrive bucăți de cod	Scheletul ar putea să nu fie util pentru toți utilizatorii  Riscul să încalc Liskov prin modificarea pașilor default
Visitor	Vreau să separ algoritmi de obiectele pe care operează	Comportamentul nou e pus într-o clasă nouă și nu în cele deja existente.  Obiectul original e trimis ca parametru la Visitor metodele din visitor care sunt diferite în funcție de obiect	Vreau să fac o operație pe toate elementele unei structuri complexe  Logica principală e separată de funcțiile auxiliare	Open/ Close  Single Responsibility	Visitori trebuie modificați la fiecare inserare sau ștergere de obiecte  Visitors ar putea să nu aibă acces la toate câmpurile necesare

Mai multe pattern-uri:



- **Concurențiale**
  - Single Threaded
  - Scheduler
  - Producer - Consumer
- **Testing**
  - Black Box - verific că respectă cerințele
  - White Box - testează toate situațiile importante
  - Unit - testează clase individuale
  - Integration - clasele împreună
  - System - tot programul
  - Regression - refac testele pentru a mă asigura că în cazul unor modificări nu am stricat ceva
  - Acceptance - software-ul îndeplinește cerințele clientului
  - Clean Room - cei care fac programe nu discută specificații de implementare cu cei care testează pentru a nu în influența

## Curs 11: QA

- proces prin care confirmăm că produsul face ce trebuie
- nu asigură calitatea 100%, dar o face mai probabilă
  - fit for purpose - face ce trebuie
  - right first time - nu exista erori

### Software Quality Assurance

Monitorizez procesul de dezvoltare pentru a asigura calitatea

ex. de standard ISO

Compus din:

prevenire defecte

dezvoltare continuă

### Testarea Software

Proces empiric care oferă informații legate de calitatea codului

Caut să găsesc bugs

Validez și verific că aplicația îndeplinește cerințele business și tehnice

Poate fi realizat oricând și trebuie integrat în toate etapele dezvoltării

Trebuie să știm când putem opri procesul de testare, pentru acest lucru putem defini mai multe criterii de oprire: niciodată, când nu mai găsesc un număr de erori, când nu mai găsesc erori critice

Trebuie să identific cât mai rapid erorile

De unde pot apărea problemele:

- comunicare greșită
- neînțelegere
- neprofesionalism
- lipsă de timp

Testing vs Debugging

Testing	Debugging
verifică respectarea cerințelor realizat de o entitate exterioară planificat și controlat	verifică corectitudinea unor secvențe de cod realizat de developer proces random

Nivele de testare:

Nume	Metodologie
Unit testing	realizat de programatori testează anumite funcții predefinit rezultate documentate
Integration	testează mai multe module simultan realizat ori de programatori/ ori de testeri rezultate documentate
Sistem	sistem testat complet pentru a verifica că face ceea ce trebuie black box

Tipuri de Testare

Nume	Metodologie
------	-------------

White Box	testerul are acces la cod poate verifica <ul style="list-style-type: none"> <li>- api</li> <li>- fault injection</li> <li>- static testing</li> </ul>
Black Box	nu am acces la codul sursă testez specificații <ul style="list-style-type: none"> <li>- exploratory - random</li> <li>- boundary - cazuri extreme</li> <li>- model based</li> </ul>
Gray Box	am acces la cod când scriu testele, dar testarea efectivă se face black box
GUI testing	verific interfața grafică <ul style="list-style-type: none"> <li>- validare date, câmpuri incorecte</li> <li>- ordine rânduri</li> </ul>
Teste de acceptare	Black -box înainte de livrarea produsului Poate fi realizat și de către client și se numește User Acceptance Testing
Regression	Reface de teste când apar modificări pentru a ne asigura că totul funcționează în continuare cum trebuie

## MANUALĂ VS AUTOMATĂ

Manual	Automat
<p>Rezolvă probleme care țin de aspect Uneori testele automate pot fi dificil de realizat</p> <p>Are nevoie de un tester care joacă rolul utilizatorului Testerul realizează <b>Test case</b> și <b>Test Plan</b></p> <p><b>Test Strategy</b> = creat de project manager, cum și se testăm  <b>Test Plan</b> = test lead, ce,cum , când și cine testează            Care va fi workflow-ul de testare            Include:</p> <ul style="list-style-type: none"> <li>- verificare design</li> <li>- production test</li> <li>- acceptance test</li> <li>- regression test</li> </ul> <p>Conține:</p> <ul style="list-style-type: none"> <li>- Identificator test</li> <li>- Elemente test</li> <li>- Ce se testează și ce nu</li> <li>- Responsabilități</li> </ul>	<p>Ieftin Rapid Mai ușor de scris cod</p> <p>Cum?</p> <p>Scriu programe care testează funcționalități            GUI - framework care generează evenimente și observă schimbări            Code - driven = clase și module testare prin introducerea de date</p>



**Test Scenario** = numele un caz de test  
**Test Case** = o condiție care se validează prin testare  
condiții și variabile care stabilesc dacă sistemul funcționează cum trebuie  
secvența de pași  
măcar câte un test pentru fiecare funcționalitate  
Conține:

- autor
- id
- descriere
- dacă a trecut sau nu
- remarci
- rezultat acceptat

**Ce este un BUG?**

Problemă a unei componente a programului  
Ca să le previn:

- style code
  - ca să poată fi înțeles de toată lumea, să fie ușor de citit și menținut
- tehnici de programare
- metodologii de programare
- analiză cod

# CURS 12

## Testare non - funcțională:

Compusă din:

- Performance sau Load Testing
  - verifică dacă pot să mă descurc cu multe date
  - Tipuri
    - load - mulți utilizatori
    - stress - extreme load, să văd când crapă aplicația
    - endurance - susțin consumul constant
    - spike - rezistență la creșteri bruște
- Usability
  - ușor de înțeles și utilizat
  - Trebuie să îndeplinească:
    - Confidențialitate
    - Integritate
    - Autentificare
    - Autorizare
    - Disponibilitate
    - Non-repudiere - nu pot nega că ceva s-a întâmplat
- Security
  - mai ales dacă folosesc date confidențiale
- Internalization și localization
  - Suportă adaptarea la limbi și regiuni multiple

## Degradare COD

- Rigiditate
- Fragilitate
- Imobilitate
- Vâscozitate

## REFACTORIZARE SOLID

### Coeziune

- Release/Reuse
- Common Closure
- Common Reuse

### Couplare

- Dependențe aciclice
- Stable dependency
- stable abstraction

# Curs 13

<b>Ce înseamnă calitate software?</b>
Calitatea design. Estetic: GUI sau source code sau comportament cu alte resurse
Trebuie să definesc atributele care mă interesează

## Aspecte:

- **siguranță**
  - complet + consistent(merge mereu conform așteptărilor) + robust
- **eficientă**
  - utilizează eficient resursele
- **mentenanță**
  - cum pot modifica și adapta
- **usability**
  - cât de ușor este de folosit
- **simplicity**
  - **control flow** - căi executabile
  - **information flow** - date transmise
  - **understanding** - operatori/ identificatori
- **modularity**
  - **coeziune**
  - **cuplare**

## Metrici:

Kilo Lines of Code

# Exemple de întrebări

1. Ce este un design pattern + tipuri de design pattern. GOF  
Un design pattern este o metodă standard de rezolvare a unor probleme care apar frecvent în dezvoltare software. Tipuri de design pattern: creational(Singleton, Factory Method, Builder, Prototype, Abstract Factory), structurale(Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy) și comportamentale(Command, Chain of Control, Iterator, Interpreter, Mediator, Memento, Observer, State, Strategy, Template method, Visitor)
2. Design patterns comportamentale  
Design patterns comportamentale rezolvă probleme legate de algoritmi și asignare de responsabilități

Ex:

Chain Of Control: trimite un request la mai mult clase care fac anumite funcții asupra acestuia

Iterator: parcurgere unei colecții fără a oferi informații legate de structura internă a acesteia

Memento: salvarea stări pentru operații de tipul savepoint, undo

Strategy: vreau să schimb modul de realizare a unui anume algoritm la run-time

**Q: Exemplele de design patterns trebuie sa fie și ele elaborate? Adică trebuie sa explicăm ce problema rezolva un anumit design pattern dat ca exemplu de noi?**

R: Da.

R2: eu n-am avut ip, dar cel dinaintea mea a avut dp-uri creationale și trebuia sa dea exemple de 4 și sa zica cate o fraza la fiecare (ce fac)

### 3. Design patterns creaționale

Cum se face crearea obiectelor

Ex:

Singleton: instanță unică la care am acces global

Abstract Factory: crearea de obiecte din aceeași familie

Factory Method: las subclasa să decidă tipul obiectului creat

Prototyp: clonare de obiecte

### 4. Design patterns structurale

Cum sunt organizate obiectele

Ex.

Adapter: comunicare între obiecte cu interfețe incompatibile

Bridge: dezvoltarea pe două dimensiuni diferite ale unui obiect. se bazează pe diferența între abstractizare și implementare

Composite: organizarea de obiecte în structuri ierarhice, dar obiectele compuse sunt tratate la fel ca obiectele simple

Decorator: adaug comportamente noi la run time

### 5. Principiile SOLID

S - single responsibility: o clasă are o responsabilitate unică

O - open/close: pot să extind funcționalitatea, dar nu să o modific pe cea deja existentă

L - liskov principle of substitution: subclasa trebuie să fie capabilă să înlocuiască superclasa în orice context

I - interface segregation: interfețe mici, specializate

D - dependency inversion: un modul superior nu trebuie să depindă de unul inferior

### 6. Etapele dezvoltării unui produs software

1. Ingineria Cerințelor

2. Planificarea arhitecturală

3. Planificarea detaliată

4. Implementare

5. Validare

6. Verificare

7. Mentenanță

### 7. Principiul lui Liskov -> L-ul de la SOLID

L - liskov principle of substitution: subclasa trebuie să fie capabilă să înlocuiască superclasa în orice context. O subclasă nu ar trebui să modifice comportamentul clasei de bază sau să îl elimine

### 8. Tipuri de testare. Exemple

Testare White Box, Black Box, Grey Box, Acceptare, Regression

### 9. Testare: definiții și dilema Testării.

Metodă prin care verific că produsul software creat îndeplinește cerințele. Dilema testării: care sunt criteriile de oprire a testării, nu știu exact când să opresc testarea

### 10. Etapele dezvoltării unei aplicații de dimensiuni mari

11. Dependency inversion [asta era subiect separat sau era subpunct la SOLID? -> e D-ul de la SOLID]

12. Metodologia scrum. Evenimente, artefacte, roluri

Evenimente: sprint, daily scrum

Artefacte: Sprint Backlog, Product Backlog, Increment

Roluri: Scrum Master, Project Manager, Dev Team

13. Metode de dezvoltare (XP, Scrum, Agile și TDD)

XP: Extreme programing

nu se lucrează peste program

se bazează pe dezvoltarea persoanelor din echipă

pair programming

Agile:

Importantă funcționalitatea nu documentația

Se lucrează cu Senior Level

unit testing

Scrum:

clientul face parte din echipă

Evenimente: sprint, daily scrum

Artefacte: Backlog

Roluri: Scrum Master, Project Manager, Dev Team

TDD:

Test Driven Development: scriu teste și apoi codul ca să trec de ele

14. Reverse Engineering. Definiții. Exemple.

Descoperire principiilor tehnologice după analizarea device-ului final, fie din curiozitate sau pentru a fura și copia tehnologia altcuiva. Cel de-al doilea război mondial, China a copiat avioane de la Rusia și SUA

TIPURI:Device-uri mecanice(3d Scan), smart-card(iau layer cu layer), aplicații militare(WWII), software

R: În cazul exemplelor, după ce dai un exemplu din produse software, posibil sa ti se ceara si unul din alt domeniu, cum ar fi domeniul militar

15. Testare manuala vs testare automată

Manuală: realizata de tester care se comportă ca un user

Automat: mai ieftin, mai rapid, pot să le refolosesc