



UNIVERSITATEA BABEŞ-BOLYAI
Facultatea de Matematică și Informatică



Programare orientată obiect

Curs 10

Laura Dioşan

POO

- Analiză și proiectare orientată obiect
- Șabloane de proiectare (*Design patterns*)
 - Singleton
 - Model View Controller

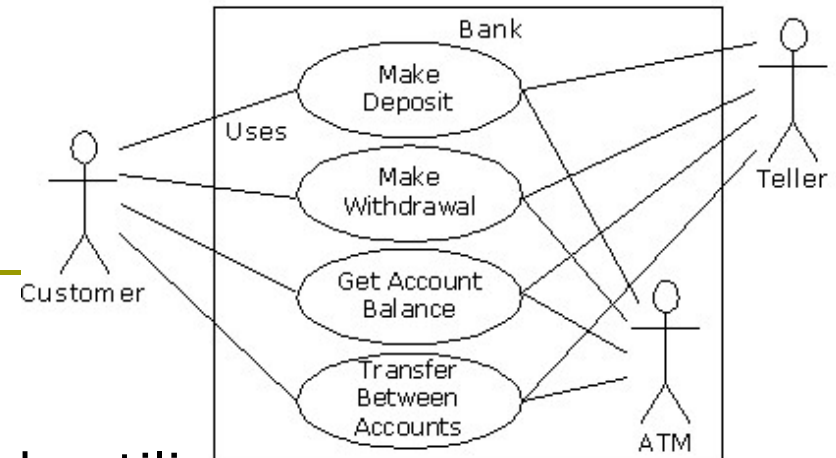
Analiză și proiectare orientată pe obiecte (APOO)

- Definiție și etape
- Limbajul UML
- Relații între clase
 - Asocieră
 - Agregare/Compoziție
 - Clase imbricate
- Liste și iteratori
 - Listă simplu înlănțuită
 - Iterator exterior
 - Iterator interior

Analiză și proiectare orientată pe obiecte (APOO)

- Abordare a ingineriei informației care modelează sistemele ca un grup de obiecte care interacționează
- AOO este o descriere a *ceea ce* sistemul trebuie să facă, sub forma unui model conceptual
 - Cazuri de utilizare
 - Diagrame de clase
 - Diagrame de interacțiune
- Proiectarea OO transformă modelul conceptual în implementare

APOO



□ 5 etape:

- Realizarea unui plan
- Ce trebuie realizat? -> cazuri de utilizare:
 - Cine va utiliza sistemul?
 - Cine sunt actorii sistemului?
 - Cum vor acționa actorii?
 - Ce probleme pot să apară?
- Cum se va construi?
 - Numele claselor
 - Responsabilitățile claselor: ce ar trebui să facă
 - Colabările între clase: cum vor interacționa clasele?
- Construcția nucleului
- Iterarea cazurilor de utilizare
- Evoluția

Limbajul UML

□ UML

- Unified Modelling Language
- Limbaj standard pentru specificarea și proiectarea artefacturilor unei aplicații orientată pe obiecte
- Un limbaj:
 - general de modelare
 - independent de limbajul de programare

Limbajul UML

- UML oferă vizualizarea elementelor arhitecturale ale unui sistem:
 - actori
 - procesele business
 - componentele (logice)
 - activitățile
 - scheme ale bazelor de date
 - reutilizabilitatea componentelor

Diagrame UML

□ Tipologie

- Diagrame de comportament → pt. înțelegerea cerințelor de funcționare a sistemului
 - Diagrama cazurilor de utilizare
 - Diagrama de secvențe
 - Diagrama de colaborare
 - Diagrama activităților
 - Diagrama stărilor

- Diagrame de structură → pt. organizarea obiectelor și stabilirea relațiilor între ele
 - Diagrama de clase
 - Diagrama de obiecte
 - Diagrama de componente
 - Diagrama de desfășurare

- Diagrame de organizare a modelului → pt. a descrie cum și unde sunt implementate obiectele
 - Diagrama de pachete
 - Diagrama de subsisteme
 - Diagrama modelului

Diagramme UML

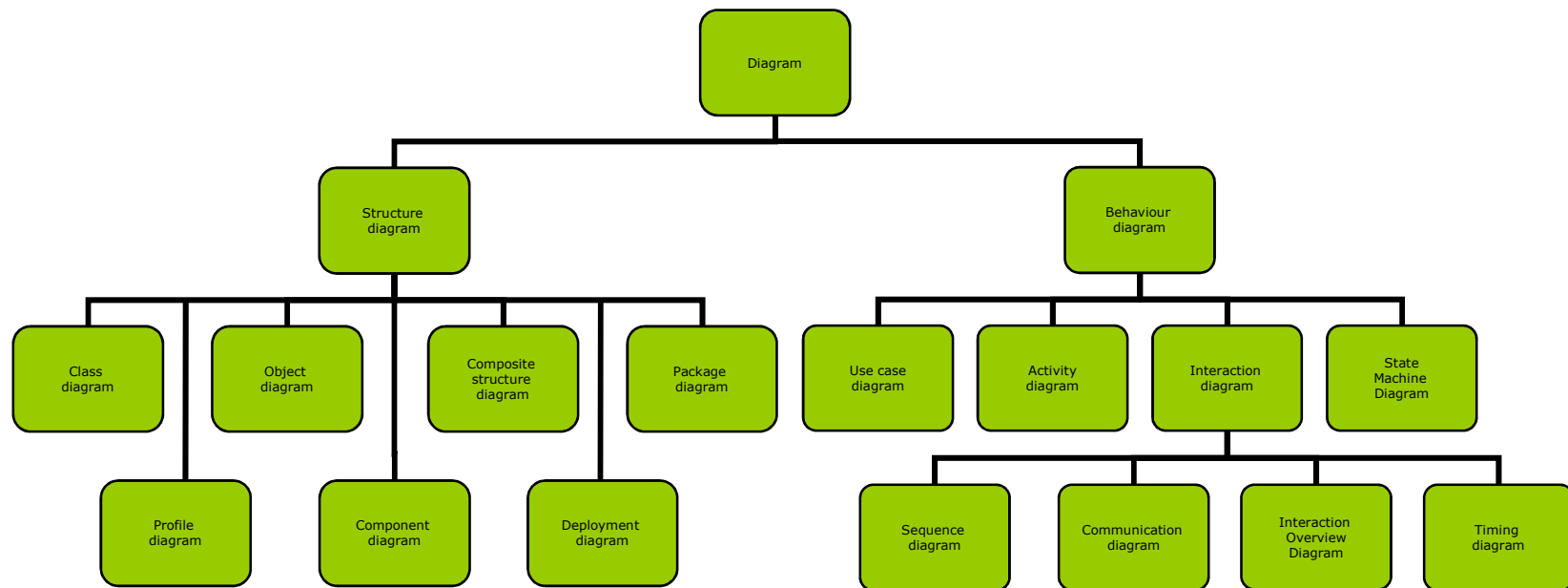


Diagrama de clasă

□ Specificarea unei clase

Numele clasei
Secțiunea de date <ul style="list-style-type: none">- protecție- numele datelor- tipul datelor
Secțiunea de metode <ul style="list-style-type: none">- protecția- numele metodei- parametrii metodei- tipul metodei

Flower
- name : String - price : Integer
+ Flower() <<constructor>> + Flower(String, Integer) <<constr>> + Flower(String) <<constr>> + Flower(const Flower &) <<constr>> + ~Flower() <<destructor>> + setName(String) + setPrice(Integer) + getName() : String + getPrice() : Integer + toString() : String + compare(Flower&) : Boolean

Flower Class
Fields
name : char* price : int
Methods
~Flower() compare(Flower& f) : bool Flower() Flower(char* name, int p) Flower(char* s) Flower(const Flower& f) getName() : char* getPrice() : int setName(char* n) : void setPrice(int p) : void toString() : char*

□ Protecția:

- + - public
- - - private
- # - protected

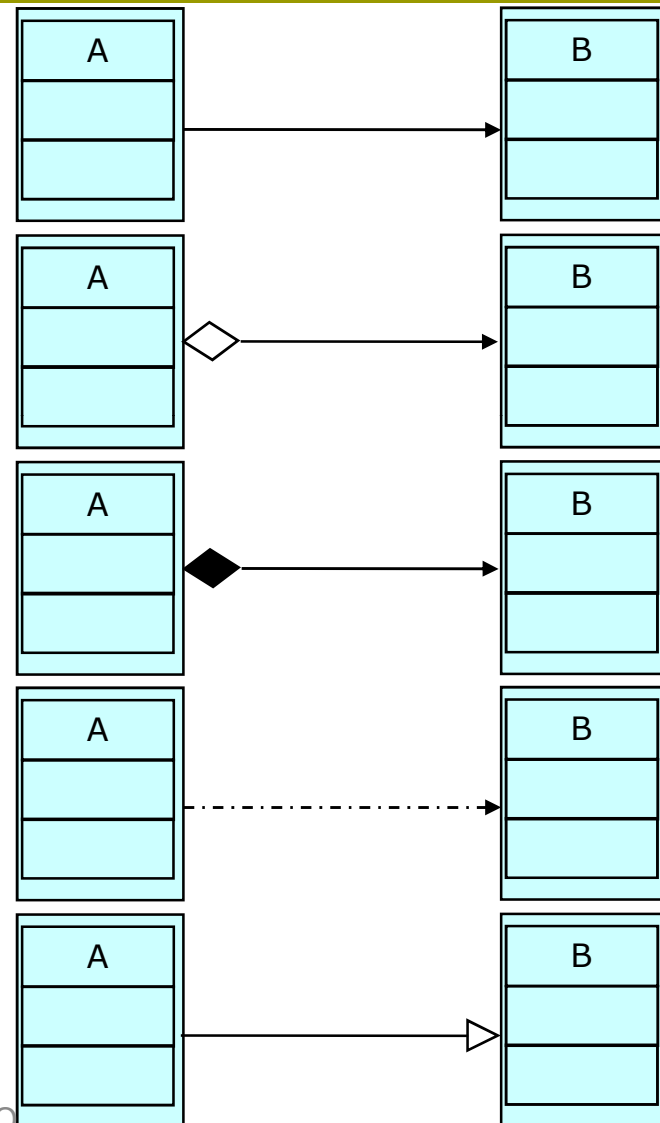
UML

- Tipuri de date predefinite în UML:
 - Integer
 - Real
 - Boolean
 - String
 - char

Diagrama de clase

□ Relații între clase

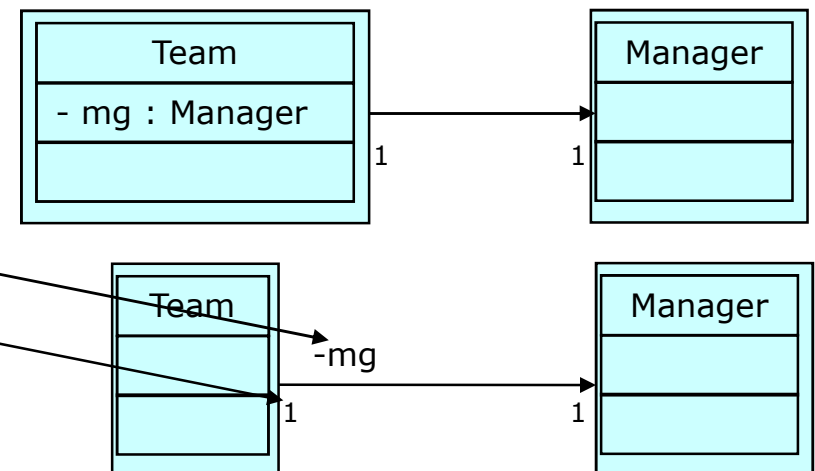
- asociere (colaborare)
 - A utilizează B
 - **Grădinarul** utilizează **Stropitoarea**
- agregare
 - A conține 1/mai multe B-uri
 - B există fără A
 - **Grădina** conține **Flori**
- compoziție
 - A conține 1/mai multe B-uri
 - B este creat de către A
 - **Floarea** este compusă din mai multe **Petale**
- dependență
 - A depinde (într-un anumit fel) de B
 - **Forma** depinde de un **ContextDeDesenare**
- moștenire
 - A este un B
 - **Floarea** este o **Plantă**



Asocierea (colaborarea)

- presupune două elemente între care există o relație
- implementată, de obicei, ca instanță a unei clase (în alta clasă)

- poate conține:
 - numele rolului la fiecare capăt,
 - cardinalitatea,
 - direcția,
 - constrângeri



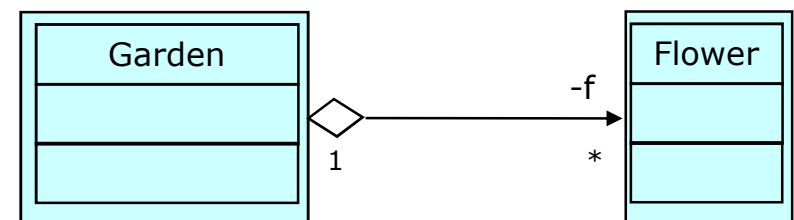
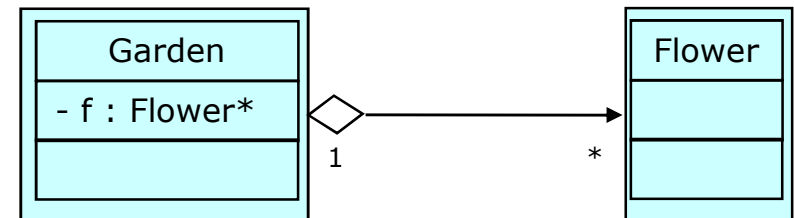
- O **echipă** are un **manager**

Exemplu de relație de asociere

- a se consulta directorul 10/association
 - *Manager.h, Manager.cpp*
 - *Team.h, Team.cpp*
 - *Test.cpp*

Agregarea

- se folosește pentru a ilustra elemente formate din componente mai mici
- este o specializare a asocierii, specificând o relație de tip întreg-parte între 2 obiecte
- partea și întregul au diferite durate de viață
- partea poate exista și fără întreg
 - A conține (1/mai multe) B-uri
 - B există fără A
- poate include:
 - numele rolului la fiecare capăt,
 - cardinalitatea,
 - direcția,
 - constrângeri
- **Grădina** conține **Flori**

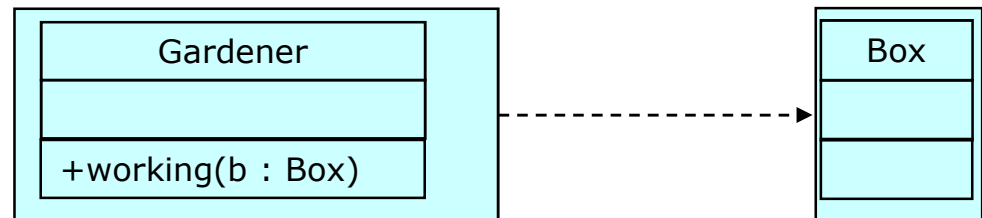


Exemplu de relație de agregare

- a se consulta directorul 10/aggregation
 - *Flower.h, Flower.cpp*
 - *Gardener.h, Gardener.cpp*
 - *test.cpp*

Dependența

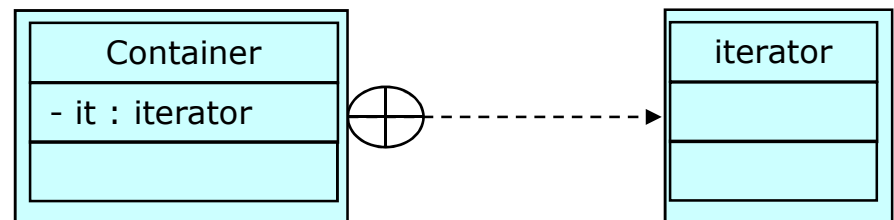
- o relație foarte slabă între 2 clase (care nu e implementată prin variabile membre)
- poate fi implementată prin intermediul argumentelor unei metode



- Exemplu – a se consulta directorul 10/dependency
 - *Box.h, Box.cpp*
 - *Gardener.h, Gardener.cpp*
 - *test.cpp*

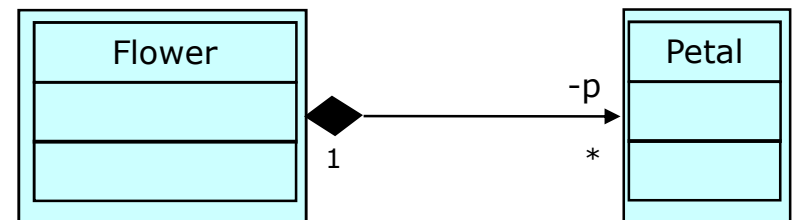
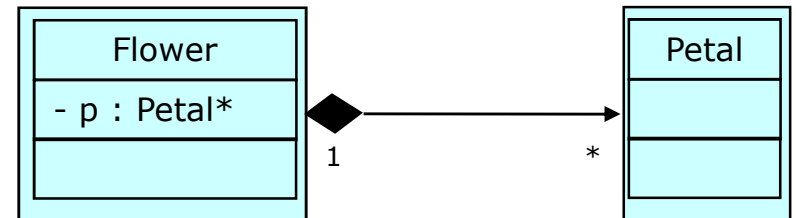
Imbricarea

- arată că elementul sursă este imbricat în elementul destinație
- clase imbricate (interioare)



Compoziție

- este o formă puternică de asociere în care întregul și partea au aceeași durată de viață
- în general, întregul controlează durata de viață a părții
- partea nu poate exista fără întreg
 - A conține (1/mai multe) B-uri
 - B este creat de către A
- poate include:
 - numele rolului la fiecare capăt,
 - cardinalitatea,
 - direcția,
 - constrângeri
- **Floarea** este compusă din **Petale**



Exemplu de relație de compoziție

□ Listă simplu înlănțuită

■ Nod

- Clasă exterioara Listei
- Clasă interioară Listei

■ Iterator

- Clasă exterioara Listei
- Clasă interioară Listei

TAD Listă Simplu Înlănțuită

1. Specificare TAD

- Domeniu

$D = \{l \mid l = (el_1, el_2, \dots), \text{ unde } el_i, i=1,2,3\dots \text{ sunt de același tip TE}\}$

- Operații:

- create
- addElem
- removeElem
- getElem
- getLength
-

TAD Listă Simplu Înlănțuită

□ Specificarea operațiilor

■ create

- Data: -
- Precond: true
- Results: l
- Postcond: $l \in D$, l este vidă

■ addElem

- Data: l, el
- Precond: $l \in D$, $e \in TE$, $l = (el1, el2, \dots, eln)$
- Results: l'
- Postcond: $l' \in D$, $l' = (el1, el2, \dots, eln, el)$

■ removeElem

- Data: l, el
- Precond: $l \in D$, $e \in TE$, $l = (el1, el2, \dots, eln)$
- Results: l'
- Postcond: $l' \in D$, $l' = (el1, el2, \dots, eln)$
without el if $el \in l$
 $l' = l$, altfel

■ getElem

- Data: l, pos
- Precond: $l \in D$, $pos \in \mathbf{Z}$, $l = (el1, el2, \dots, eln)$
- Results: el
- Postcond: $el \in TE$, $el = elpos$

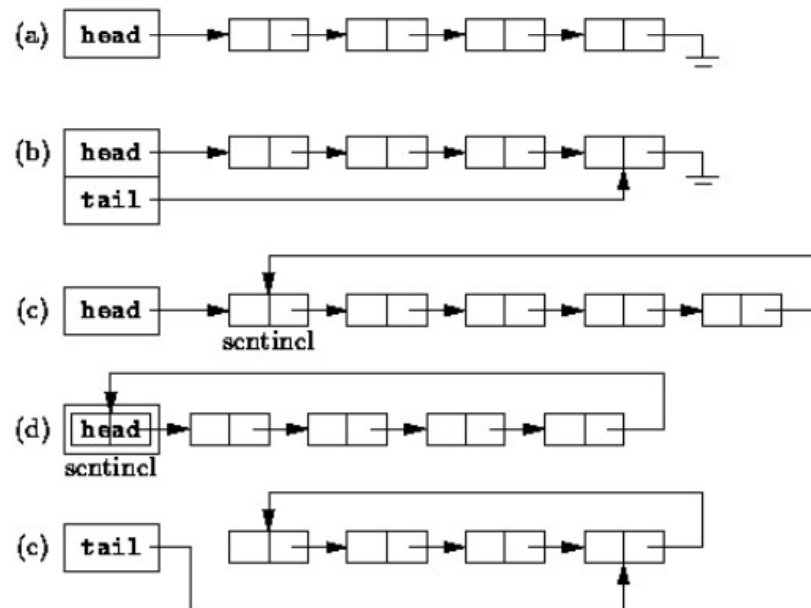
■ getLength

- Data: l
- Precond: $l \in D$, $l = (el1, el2, \dots, eln)$
- Results: n
- Postcond: $n \in \mathbf{Z}$

TAD Listă Simplu Înlănțuită

2. Proiectarea TAD-ului

- Reprezentarea TAD-ului
 - static – cu 2 vectori
 - dinamic – cu alocare dinamică de memorie



- Operațiile TAD în pseudo-cod

TAD Listă Simplu Înlănțuită

- Observații:
 - O listă reține:
 - un **cap** -> un pointer către primul element al listei
 - o **coadă** -> un pointer către ultimul element al listei (opțional)
 - Accesul la elementele listei începe cu primul element (cap) și utilizează legăturile între noduri
 - Un element nou poate fi inserat oriunde în listă
 - Nu există restricții privind capacitatea listei (decât cele date de Heap)
 - Orice listă are asociat un iterator – pentru accesarea elementelor

Iterator

- Un obiect care se mișcă printr-un container de obiecte și selectează unul dintre aceste obiecte, fără a oferi acces direct la implementarea containerului
- Pointer inteligent (*smart pointer*) → de obicei, imită operațiile unui pointer
- Desemnat a fi sigur
- O abstractizare a genericității

Iterator

- ❑ Orice container are asociată o clasă numită **iterator**
- ❑ Se declară numele clasei iterator
- ❑ Iteratorul se declară a fi prieten (friend) cu containerul
- ❑ Se definește clasa iterator
- ❑ Câteva funcții importante ale iteratorului:
 - *moveFirst()* \leftrightarrow $i = 0$ sau $crt = head$
 - *moveNext()* \leftrightarrow $i++$ sau $crt = crt->next$
 - *hasNext()* \leftrightarrow $i < n - 1$ sau $crt->next \neq NULL$
 - *isValid()* \leftrightarrow $i < n$ sau $crt \neq NULL$
 - *getCrtElem()* \leftrightarrow return $elem[i]$ sau return $crt->info$

Iteratori - tipologie

- Locul declarării
 - Iteratori externi
 - Iteratori interni (*true iterators*)

- Capacități
 - IO
 - Iteratori de intrare (doar citire, se deplasează înainte)
 - Iteratori de ieșire (doar scriere, se deplasează înainte)
 - Mișcare
 - Înainte (se deplasează doar înainte)
 - Bidirecționali (se deplasează înainte și înapoi)
 - Acces aleator (similar unui pointer)

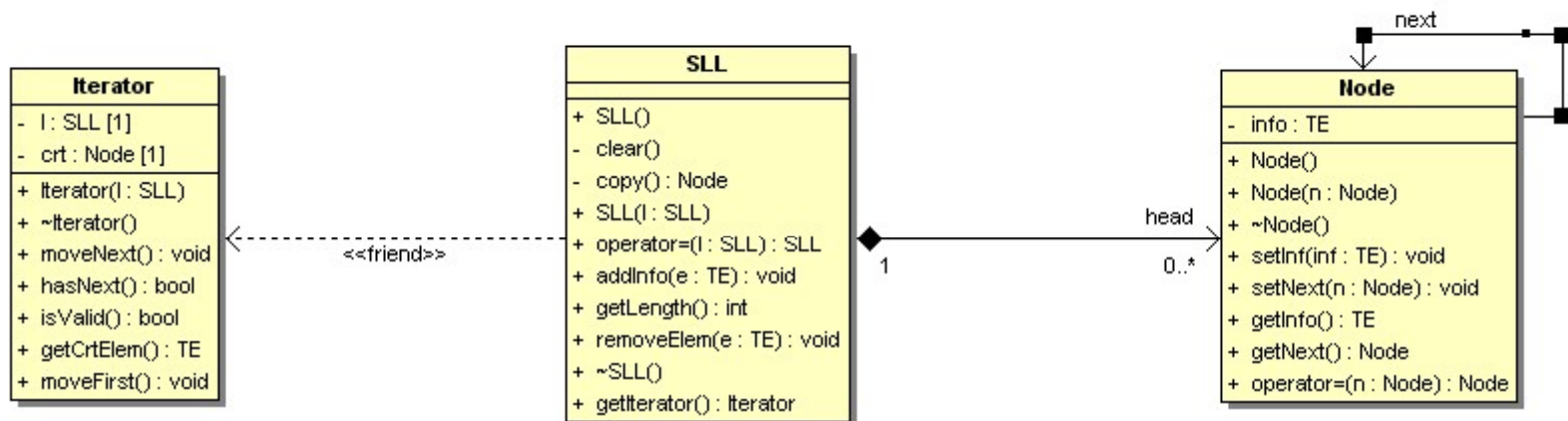
TAD Listă Simplu Înlănțuită

3. Implementare TAD

- a se consulta subdirectoarele directorului **10/SLL/**
 - *SLL_OuterNode_int*
 - *SLL_OuterNode_InnerIterator_Pointer_int*

 - *SLL_InnerNode_int*
 - *SLL_InnerNode_InnerIterator_Pointer_int*
 - *SLL_InnerNode_InnerIterator_Pointer_Flower*

TAD LSI – diagrama UML



Aplicație Medic de familie - pacienți

□ Problemă

- Dezvoltați o aplicație pentru un medic care dorește o mai bună administrare a pacienților săi. Aplicația va permite:
 - Adăugarea unui pacient nou (PIN, nume, adresă)
 - Adăugarea unei noi consultații pentru un pacient (data, diagnostic, medicamente)
 - Afișarea istoricului unui pacient (toate consultațiile)
 - Afișarea tuturor pacienților (în ordine alfabetică a numelor)
 - Afișarea pacienților care suferă de o anumită boală
 - Afișarea pacienților care iau un anumit medicament

Analiza și proiectarea problemei

- Pași:
 - Formularea detaliată a problemei
 - Cerințele sistemului și planificarea lor
 - Tabelul cerințelor
 - Definirea cazurilor de utilizare
 - Identificarea entităților
 - Analiza
 - Modelul conceptual
 - Identificarea entităților
 - Stabilirea relațiilor între entități
 - Stabilirea proprietăților fiecărei entități
 - Evenimente sistem
 - Comportamentul sistemului conform cazurilor de utilizare
 - Proiectare
 - Descrierea cazurilor de utilizare reale
 - Specificarea TAD-urilor
 - Diagrama de clasă
 - Implementare
 - Testare
 - Dezvoltare bazată pe testare
 - Cutia deschisă
 - Cutia închisă

Cerințele sistemului și planificarea lor

□ Tabelul cerințelor

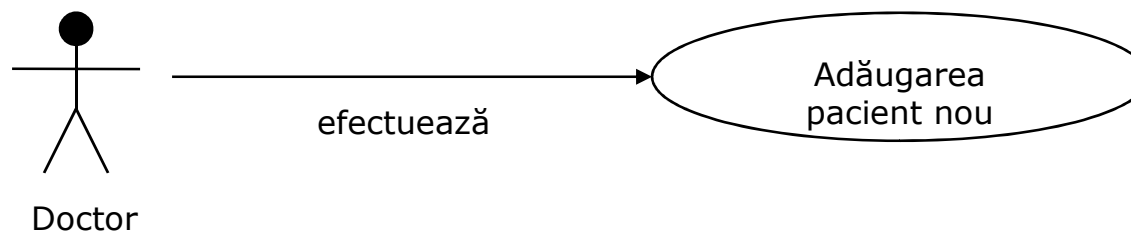
nr	explicație	Tip
1.	Dezvoltarea unei aplicații pentru un medic care dorește o mai bună administrare a pacienților săi	Evident
2.	Adăugarea unui nou pacient	Evident
3.	Adăugarea unei noi consultații	Evident
4.	Afișarea istoricului unui pacient	Evident
5.	Afișarea tuturor pacienților în ordine alfabetică	Evident
6.	Afișarea tuturor pacienților care suferă o anumită boală	Evident
7.	Afișarea tuturor pacienților care iau un anumit medicament	Evident
8.	Datele de intrare se preiau din fișierul " <i>patients.txt</i> "	Ascuns
9.	Datele de intrare se preiau din fișier	Ascuns
10.	Datele de ieșire se vor reține în fișiere	Ascuns
11.	Implementarea unei funcții de căutare a unui pacient	Ascuns

Cerințele sistemului și planificarea lor

- Cazuri de utilizare – interacțiunile între actori și sistem
 - Adăugarea unui nou pacient
 - Modificarea unui pacient existent
 - Eliminarea unui pacient
 - Afișarea tuturor informațiilor despre un pacient
 - Adăugarea unei noi consultații

Cerințele sistemului și planificarea lor

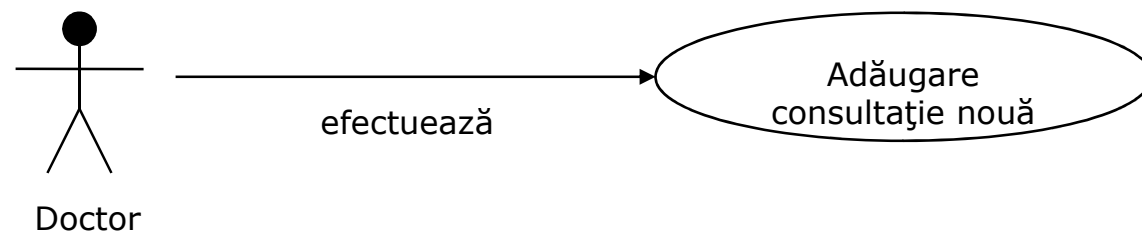
- Cazuri de utilizare – detalii
 - Denumire:
 - Adăugarea unui nou pacient
 - Descriere:
 - Adăugarea informațiilor (nume, adresă, data nașterii, telefon, etc) despre noul pacient
 - Autor:
 - doctorul
 - Referințe:
 - 2 (din tabelul de cerințe)
 - Precondiție:
 - Doctorul are suficient spațiu pentru un nou pacient
 - Postcondiție:
 - Noul pacient va aparține listei de pacienți a doctorului



Cerințele sistemului și planificarea lor

□ Cazuri de utilizare – detalii

- Denumire:
 - Adăugarea unei noi consultații
- Descriere:
 - Adăugarea informațiilor despre o consultație (data, simptome, boală, medicamente) pentru un anumit pacient (identificat prin nume)
- Autor:
 - doctor
- Referințe:
 - 3 (din tabelul de cerințe)
- Precondiție:
 - Doctorul are suficient spațiu pentru adăugarea unei noi consultații, iar pacientul dat este unul valid
- Postcondiție:
 - Noua consultație este adăugată în lista de consultații a doctorului și în istoricul pacientului respectiv



Cerințele sistemului și planificarea lor

□ Identificarea entităților

- Doctor
- Pacient
- Consultație
- Boală
- Medicament

Analiza

- Modelul conceptual
 - Identificarea entităților
 - Stabilirea relațiilor între entități
 - Stabilirea proprietăților fiecărei entități

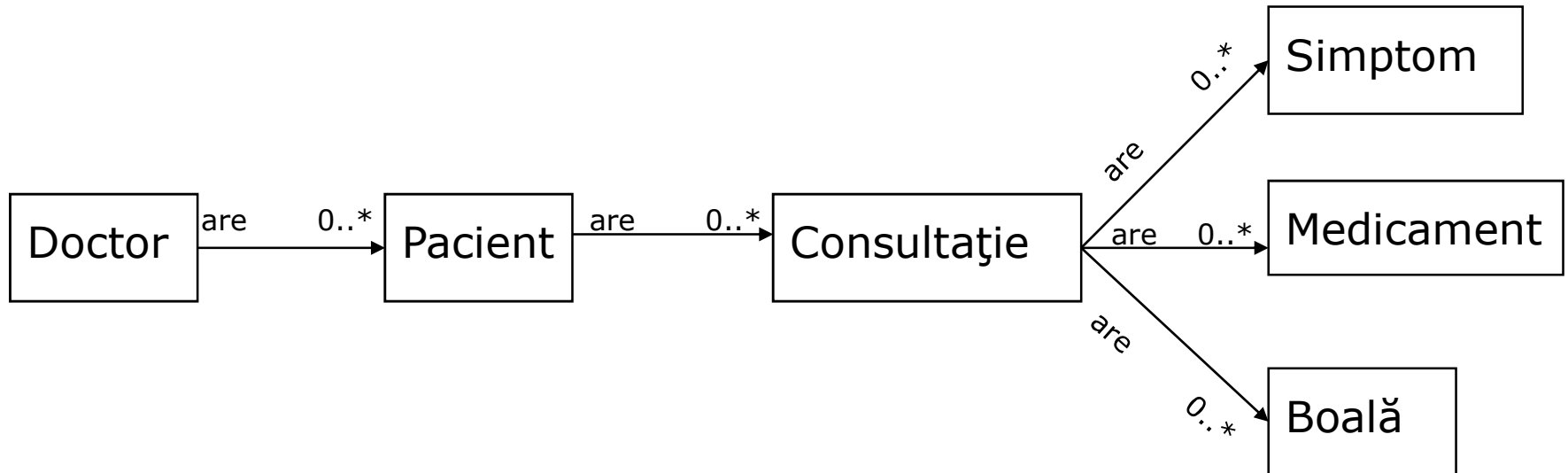
Analiza – modelul conceptual

□ Identificarea entităților

- Doctor
- Pacient
- Consultație
- Boală
- Medicament
- Simptom

Analiza – modelul conceptual

□ Relații între entități



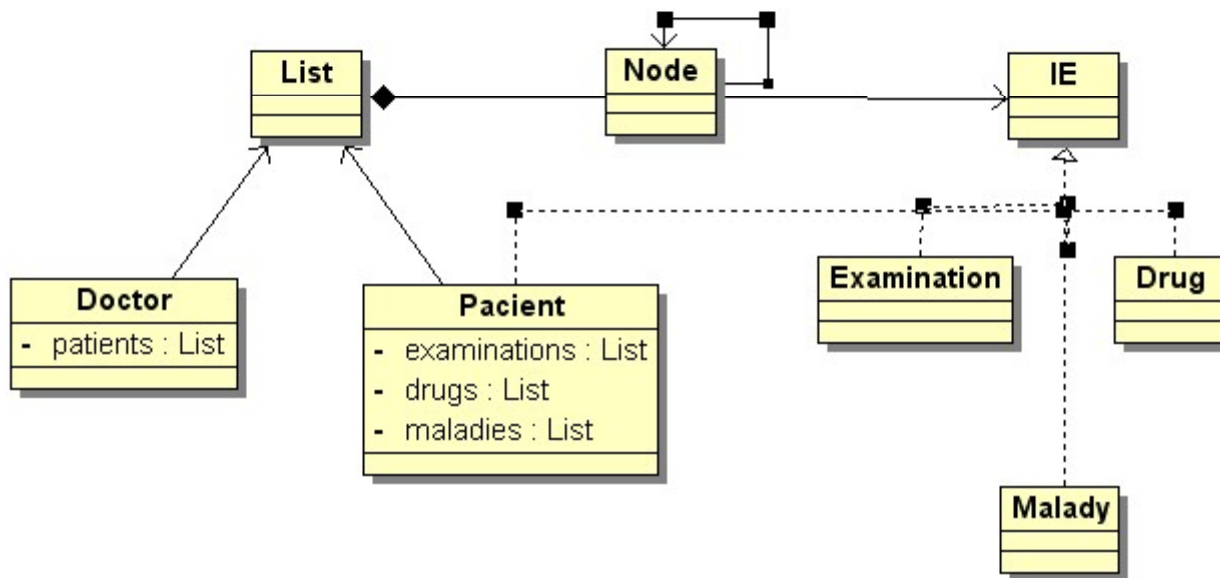
Analiza – modelul conceptual

□ Proprietățile entităților

- Doctor
 - Nume, specialitate, ...
- Pacient
 - Nume, data nașterii, adresă, telefon, ...
- Consultație
 - Data, ...
- Boală
 - Nume, intensitate, ...
- Medicament
 - Nume, cantitate, ...
- Simptom
 - Nume, tip, ...

Proiectare

- Specificarea TAD-urilor
- Diagrama de clase



Implementare

- A se consulta exemplu din directorul *10/doctor*

Testare

- Cutia închisă
 - Metodă bazată pe specificarea unui subalgoritm
- Cutia deschisă
 - Metodă bazată pe codul unui subalgoritm
- Dezvoltare bazată pe testare (*Test Driven Development – TDD*)

Testare

```
void Doctor::addExaminationForPacient(char* pn, Examination* e){
    IIterator* it = patients->getIterator(); bool found = false;           (1)
    while ((it->isValid()) && (!found)){                                     (2)
        if (strcmp(((Patient*)it->getCrtElem())->getName(), pn) == 0)      (3)
            found = true;                                                 (4)
        else
            it->moveNext();                                                 (5)
    }                                                                       (6)
    if (!found){                                                         (7)
        char* s = new char[1000];
        sprintf(s , "%s %s %s", "patient ", pn, " is not in the list");    (8)
        throw s;
    }
    else{
        ((Patient*)it->getCrtElem())->addExamination(e);                 (9)
    }
}                                                                           (10)
```

Testare *Black box* (cutia închisă)

□ Specificarea algoritmului

Data	d, s, ex
Precond	d este un doctor, s este un șir de caractere, ex este o consultație
Rez	d'
Postcond	$d' = d \cup \{ex \text{ pentru pacientul } s \text{ dacă doctorul } d \text{ are ca pacient persona cu numele } s\}$ sau $d' = d$, altfel

Testare *Black box* (cutia închisă)

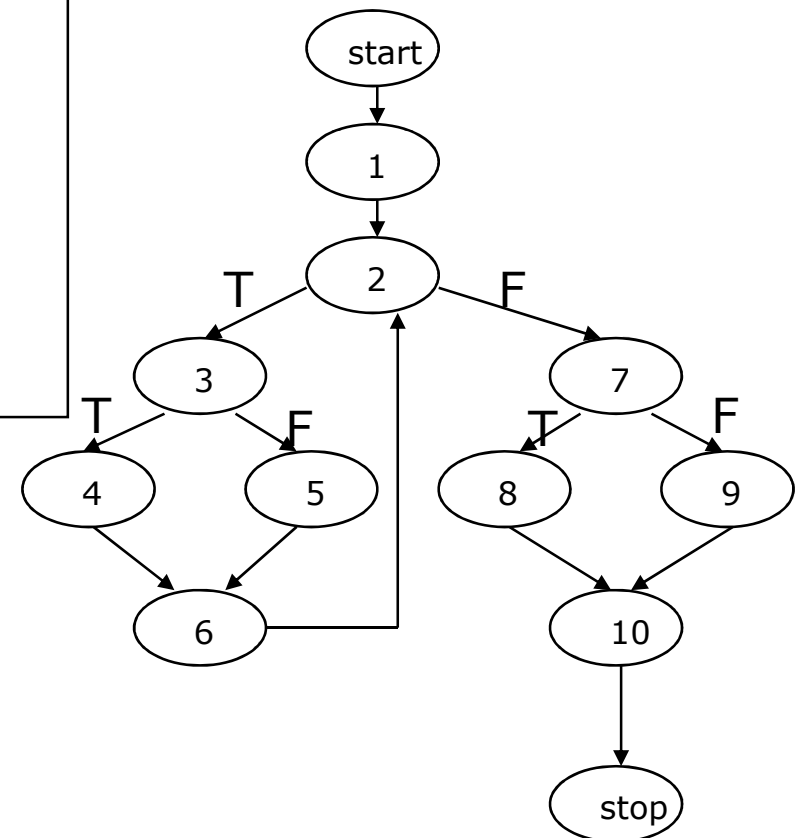
□ Cazuri de testare

Caz de testare	Valoare
T1	Doctorul d are 0 pacienți
T2	Doctorul d are mai mulți pacineți, printre care și pacientul cu numele s
T3	Doctorul d are mai mulți pacineți, dar printre care nu se află și pacientul cu numele s

Testare *White box* (cutia deschisă)

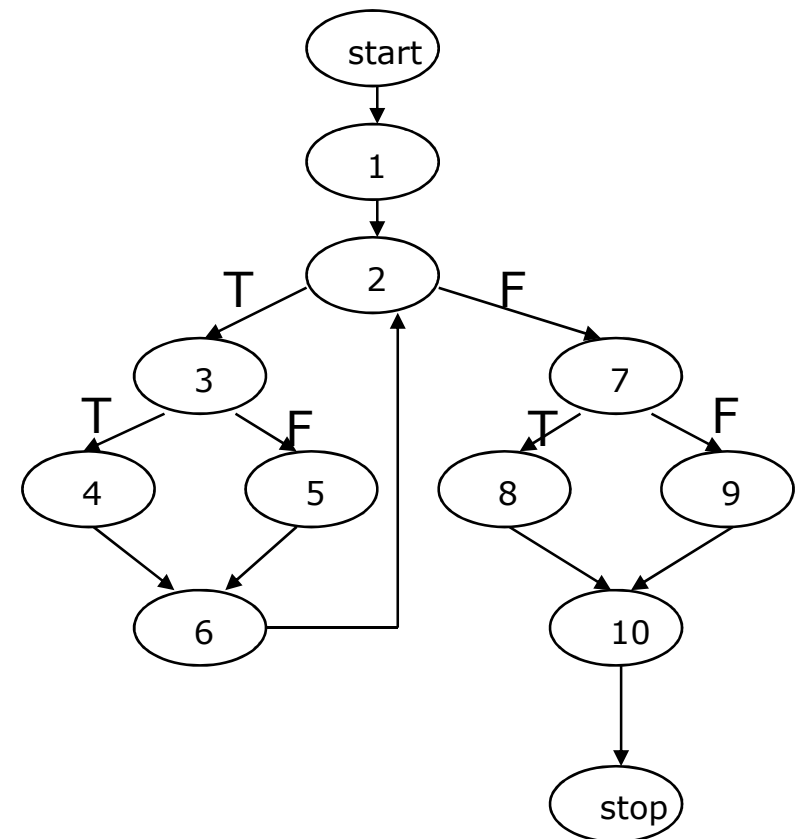
□ Graful de flux

```
void Doctor::addExaminationForPacient(char* pn, Examination* e){  
    IIterator* it = patients->getIterator(); bool found = false;      (1)  
    while ((it->isValid()) && (!found)){                               (2)  
        if (strcmp(((Patient*)it->getCrtElem())->getName(), pn) == 0) (3)  
            found = true;                                           (4)  
        else  
            it->moveNext();                                           (5)  
    }                                                                 (6)  
    if (!found){                                                    (7)  
        char* s = new char[1000];  
        sprintf(s, "%s %s %s", "patient ", pn, " is not in the list"); (8)  
        throw s;  
    }  
    else{                                                            (9)  
        ((Patient*)it->getCrtElem())->addExamination(e);  
    }                                                                 (10)  
}
```



Testare *White box* (cutia deschisă)

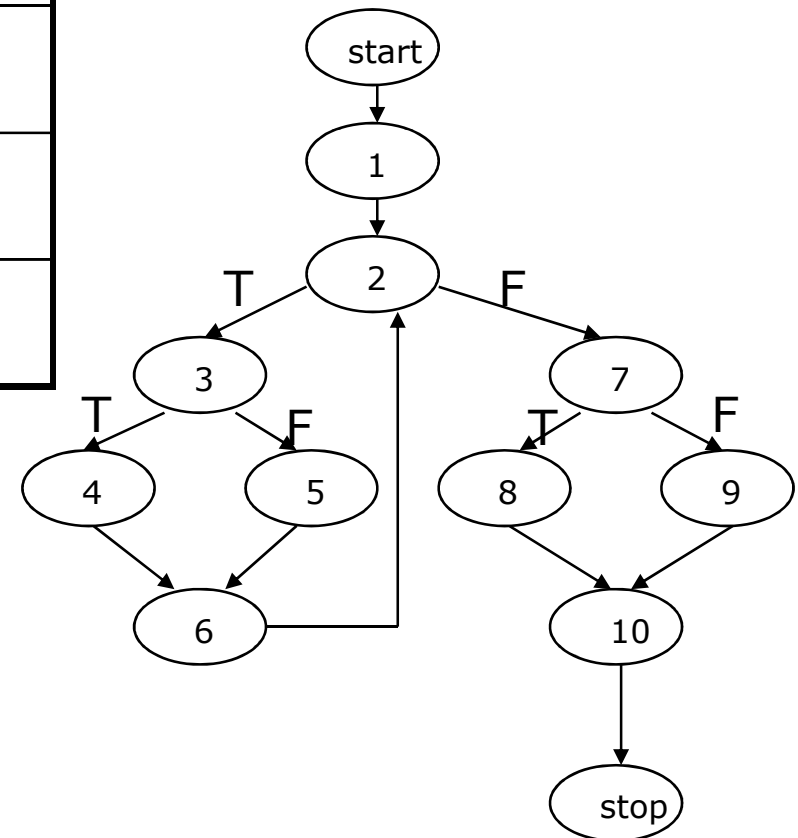
- Complexitate ciclometrică = nr. min de drumuri în graf
 - # predicatelor + 1
 - = 3 + 1
 - # muchilor - # nodurilor + 2
 - = 12 - 10 + 2



Testare *White box* (cutia deschisă)

□ Drumuri posibile în graf

Drum	Caz de testare
1 2 7 9 10	T1
1 (2 3 4 6) 7 8 10	T2
1 (2 3 5 6) 7 9 10	T3



Dezvoltare bazată pe testare

(Test Driven Development – TDD)

- tehnică de dezvoltare a aplicațiilor
- stil de programare în care testele se scriu înainte codurilor care se presupun a trebui testate

- CUM?
 - Funcții assert
 - Biblioteci speciale
 - CppUnit
 - ECUT
 -

Șabloane de proiectare (*Design patterns*)

- Singleton
- Model View Controller

Șabloane de proiectare (*Design patterns*)

- Un mod special de a rezolva anumite probleme

Singleton

- Permite crearea unei singure instanțe la o clasă

- De ce?
 - Pentru a preveni clientul programator de a controla durata de viață a unui obiect

- Cum?
 - Constructor(i) privat(i)
 - Instanțe statice
 - A se consulta exemplul din directorul *10/singleton*

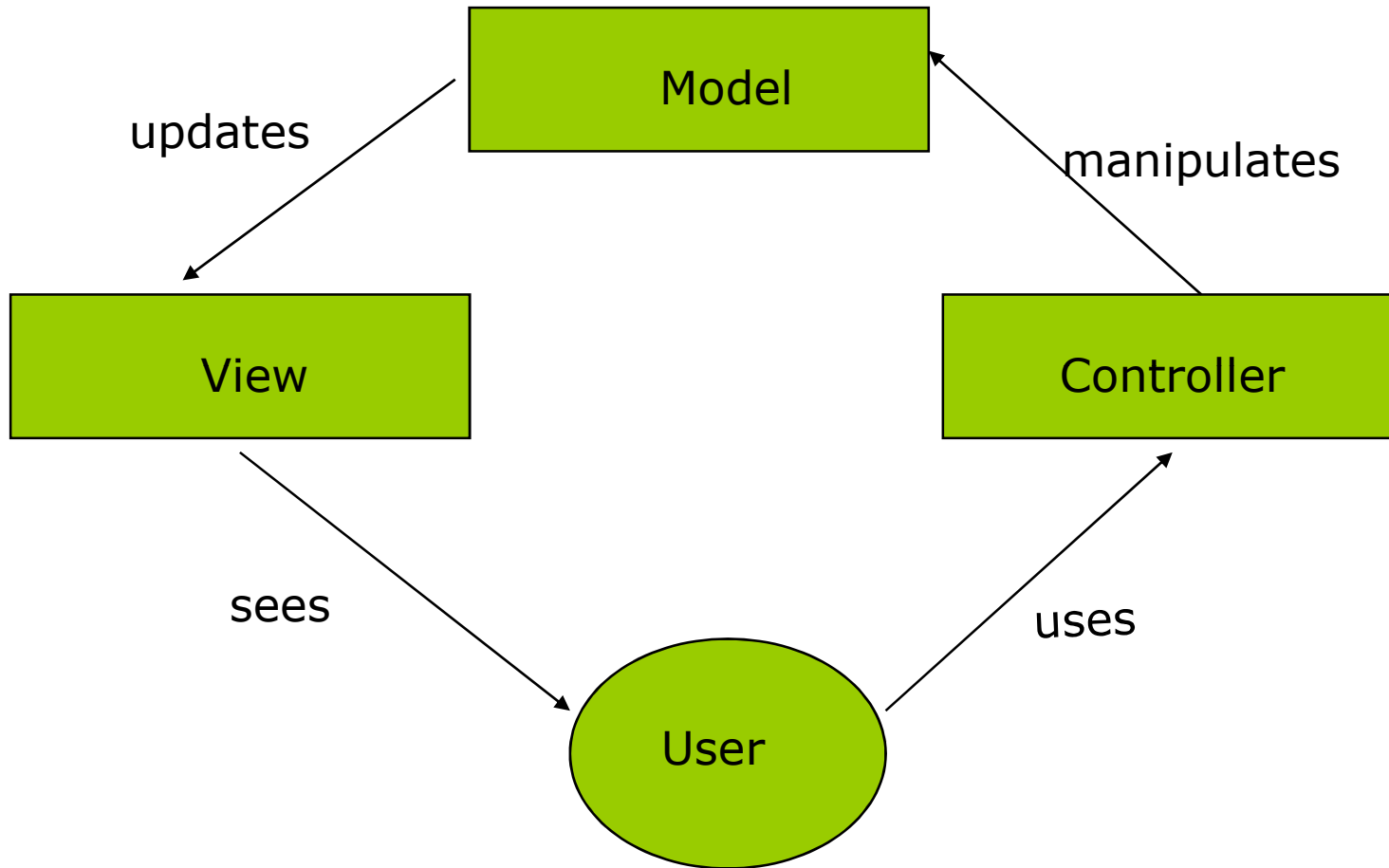
Model View Controller

- Şablon architectural → arhitectură pe mai multe nivele

- 3 nivele:
 - Prezentare → View
 - Nivel logic → Model and Controller
 - Nivel de date → stocarea și accesarea datelor

- MVC
 - Model → controlează informați și notifică observatorii când aceasta se modifică
 - View → prezintă modelul într- formă “frumoasă”, potrivită pentru interacțiune
 - Controller → primește intrări și inițiază răspunsuri

Model View Controller



Model View Controller – aplicația medicală

□ Model

- Obiecte
 - *Pacient*
 - *Consultație*
- Containeri
 - *Listă*
 - *Iterator*
- Excepții
- Alte clase

□ View

- Interfață tip consolă
- Interfață grafică

□ Controller

- *Doctor*