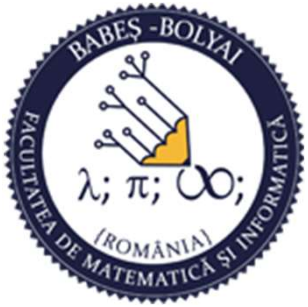
UNIVERSITATEA BABEŞ-BOLYAI
Facultatea de Matematică şi Informatică



Programare orientată obiect

Curs 11

Laura Dioşan

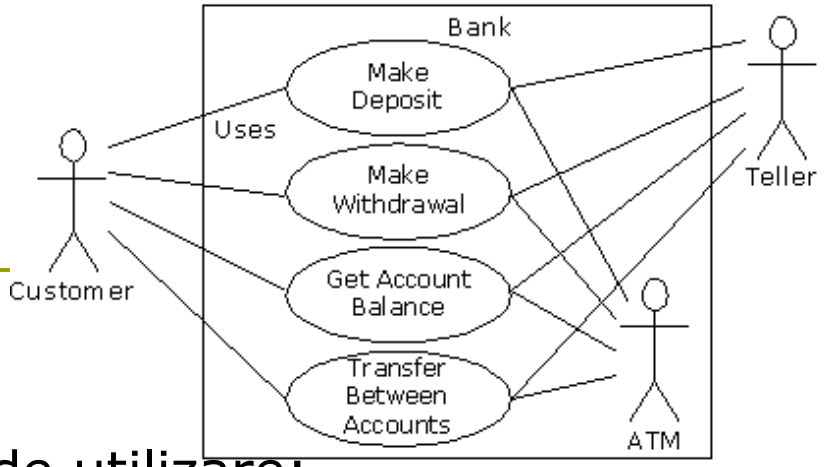
POO

- **Analiză şi proiectare orientată pe obiecte** (APOO)
  - Definire şi etape
  - Limbajul UML
  - Relaţii între clase
    - Asociere
    - Agregare/Compoziţie
    - Clase imbricate
  - Liste şi iteratori
    - Listă simplu înlănţuită
    - Iterator exterior
    - Iterator interior

Analiză şi proiectare orientată pe obiecte (APOO)

- Abordare a ingineriei informaţiei care modelează sistemele ca un grup de obiecte care interacţionează
- AOO este o descriere a *ceea ce* sistemul trebuie să facă, sub forma unui model conceptual
  - Cazuri de utilizare
  - Diagrame de clase
  - Diagrame de interacţiune
- 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 acţiona actorii?
    - Ce probleme pot să apară?
  - Cum se va construi?
    - Numele claselor
    - Responsabilităţile claselor: ce ar trebui să facă
    - Colabărările între clase: cum vor interacţiona clasele?
  - Construcţia nucleului
  - Iterarea cazurilor de utilizare
  - Evoluţia

Limbajul UML

- UML
  - Unified Modelling Language
  - Limbaj standard pentru specificarea şi proiectarea artefacturilor unei aplicaţii 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)
  - activităţile
  - scheme ale bazelor de date
  - reutilizabilitatea componentelor

Diagrame UML

- Tipologie
  - Diagrame de comportament → pt. înţelegerea cerinţelor de funcţionare a sistemului
    - Diagrama cazurilor de utilizare
    - Diagrama de secvenţe
    - Diagrama de colaborare
    - Diagrama activităţilor
    - Diagrama stărilor
  - Diagrame de structură → pt. organizarea obiectelor şi stabilirea relaţiilor între ele
    - Diagrama de clase
    - Diagrama de obiecte
    - Diagrama de componente
    - Digrama de desfăşurare
  - Diagrame de organizare a modelului → pt. a descrie cum şi unde sunt implementate obiectele
    - Diagrama de pachete
    - Diagrama de subsisteme
    - Diagrama modelului

Diagrame UML

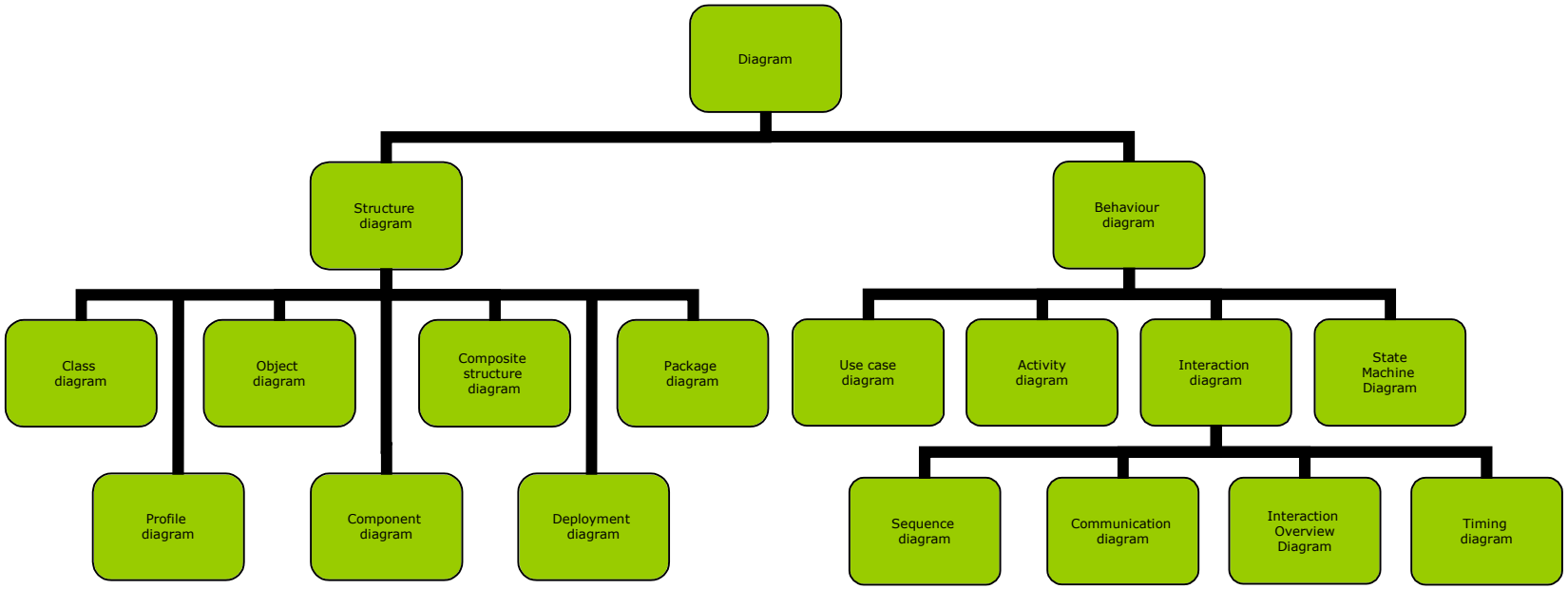
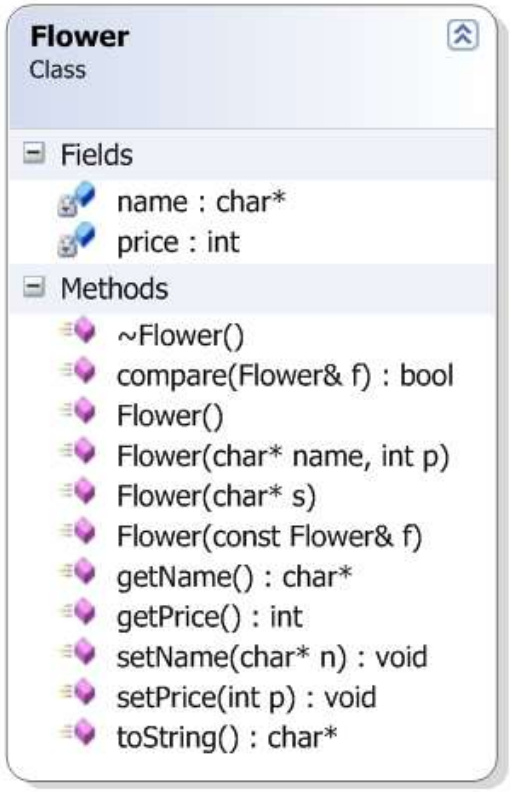


Diagrama de clasă

- Specificarea unei clase

Numele clasei
Secţiunea de date - protecţie - numele datelor - tipul datelor
Secţiunea de metode - protecţia - 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



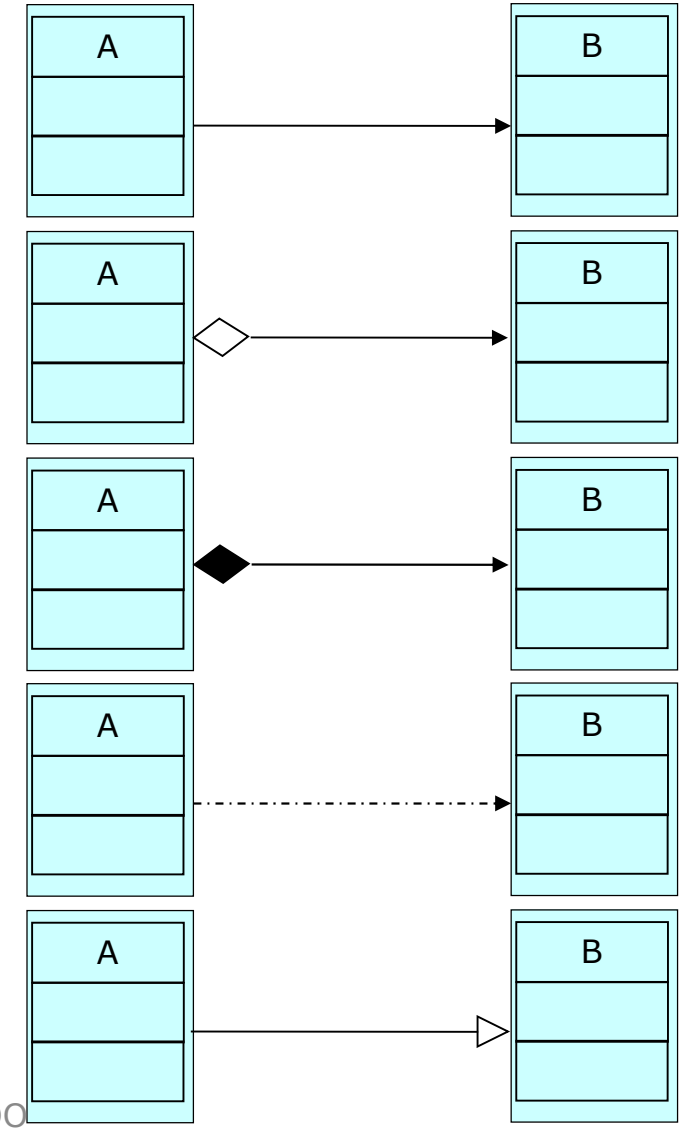
- Protecţia:
  - + - public
  - – - private
  - # - protected

UML

- Tipuri de date predefinite în UML:
  - Integer
    - Size : Integer
    - Elements : Integer[]
  - Real
  - Boolean
  - String
  - char

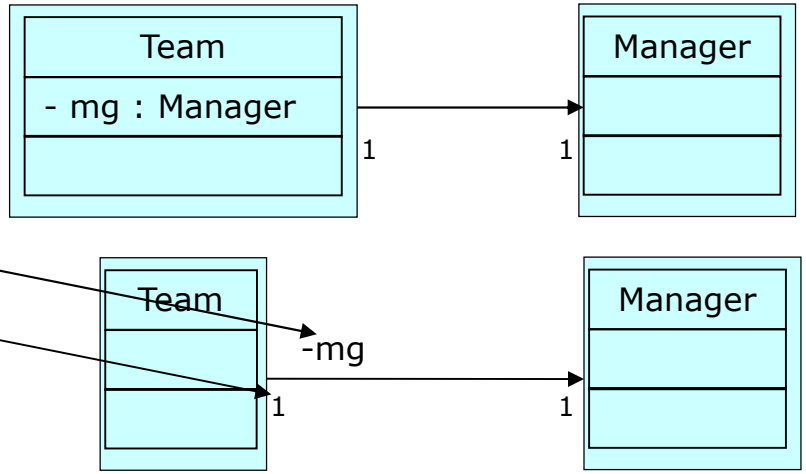
Diagrama de clase

- Relaţii între clase
  - asociere (colaborare)
    - A utilizează B
    - **Grădinarul** utilizează **Stropitoarea**
  - agregare
    - A conţine 1/mai multe B-uri
    - B există fără A
    - **Gradina** conţine **Flori**
  - compoziţie
    - A conţine 1/mai multe B-uri
    - B este creat de către A
    - **Floarea** este compusă din mai multe **Petale**
  - dependenţă
    - A depinde (într-un anumit fel) de B
    - **Forma** depinde de un **ContextDeDesenare**
  - moştenire
    - A este un B
    - **Floarea** este o **Plantă**



Asocierea (colaborarea)

- presupune două elemente între care există o relaţie
- implementată, de obicei, ca instanţă a unei clase (în alta clasă)
- poate conţine:
  - numele rolului la fiecare capăt,
  - cardinalitatea,
  - direcţia,
  - constrângeri



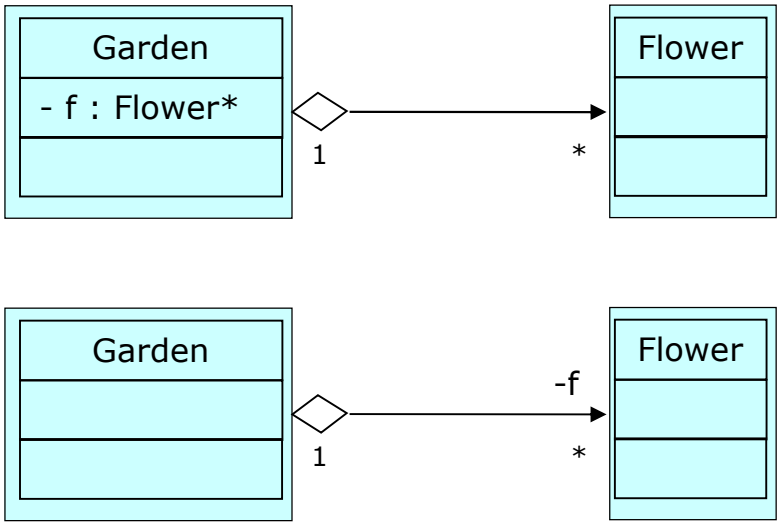
- O **echipă** are un **manager**

Exemplu de relație de asociere

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

Agregarea

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

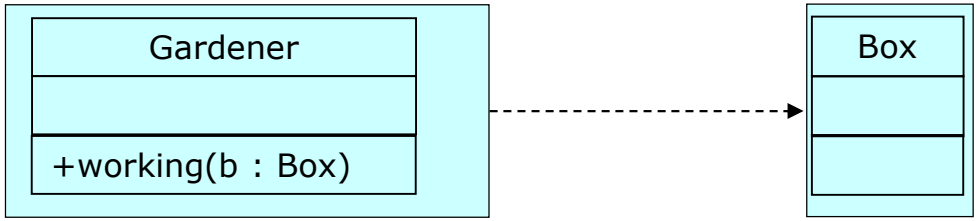


Exemplu de relație de agregare

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

Dependența

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

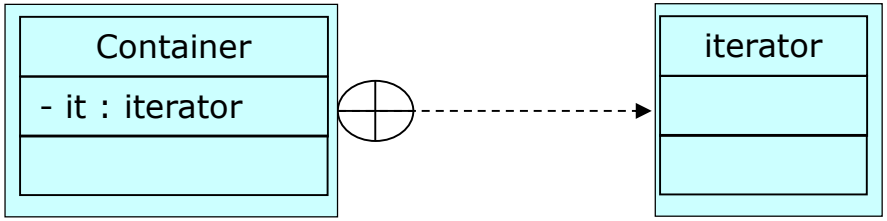


- Exemplu – a se consulta directorul 05/dependency
  - *Box.h, Box.cpp*
  - *Gardener.h, Gardener.cpp*
  - *test.cpp*

Imbricarea

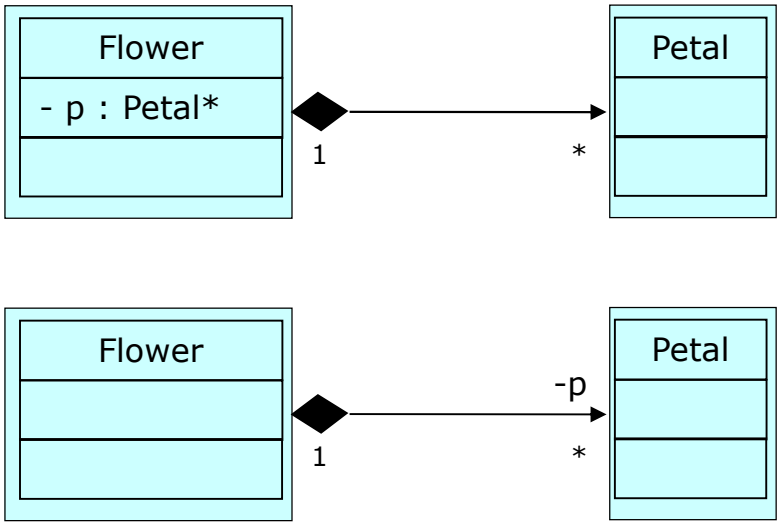
- arată că elementul sursă este imbricat în elementul destinaţie

- clase imbricate (interioare)



Compoziție

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



Exemplu de relație de compoziție

- Listă simplu înlănţuită
  - Nod
    - Clasă exterioara Listei
    - Clasă interioară Listei
  - Iterator
    - Clasă exterioara Listei
    - Clasă interioară Listei

TAD Listă Simplu Înlănțuită

1. Specificare TAD
   - Domeniu

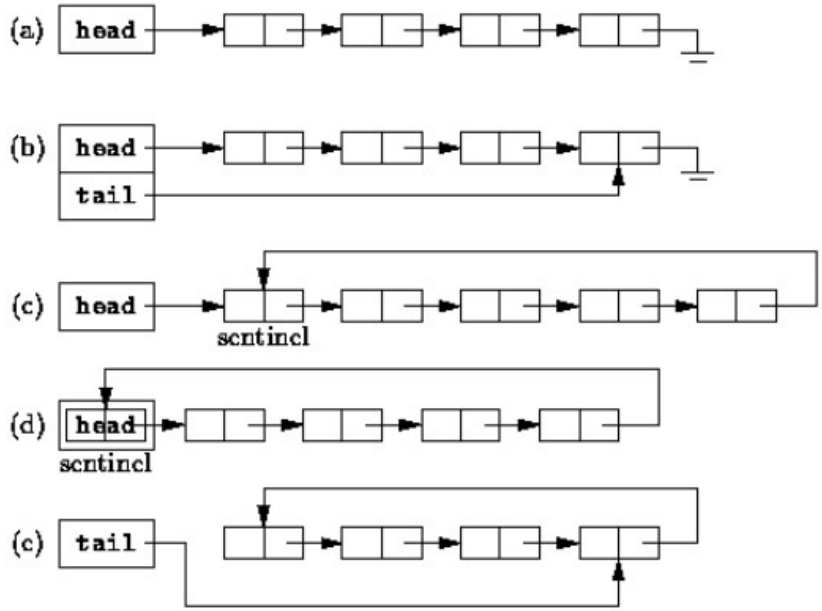
     D = {l | l = (el_1, el_2, …), unde el_i, i=1,2,3… sunt de acelaşi tip TE}
   - Operaţii:
     - create
     - addElem
     - removeElem
     - getElem
     - getLength
     - ….

TAD Listă Simplu Înlănțuită

- Specificarea operaţiilor
  - create
    - Data: -
    - Precond: true
    - Results: l
    - Postcond: l ϵ D, l este vidă
  - addElem
    - Data: l, el
    - Precond: l ϵ D, e ϵ TE, l =(el1, el2, …, eln)
    - Results: l'
    - Postcond: l' ϵ D, l' =(el1, el2, …, eln, el)
  - removeElem
    - Data: l, el
    - Precond: l ϵ D, e ϵ TE, l =(el1, el2, …, eln)
    - Results: l'
    - Postcond: l' ϵ D, l' =(el1, el2, …, eln) without el if el ϵ l
      l' = l, altfel
  - getElem
    - Data: l, pos
    - Precond: l ϵ D, pos ϵ **Z**, l =(el1, el2, …, eln)
    - Results: el
    - Postcond:el ϵ TE, el = elpos
  - getLength
    - Data: l
    - Precond: l ϵ D, l =(el1, el2, …, eln)
    - Results: n
    - Postcond:e n ϵ **Z**

TAD Listă Simplu Înlănțuită

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



   - Operaţiile TAD în pseudo-cod

TAD Listă Simplu Înlănțuită

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

Iterator

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

Iterator

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

Iteratori - tipologie

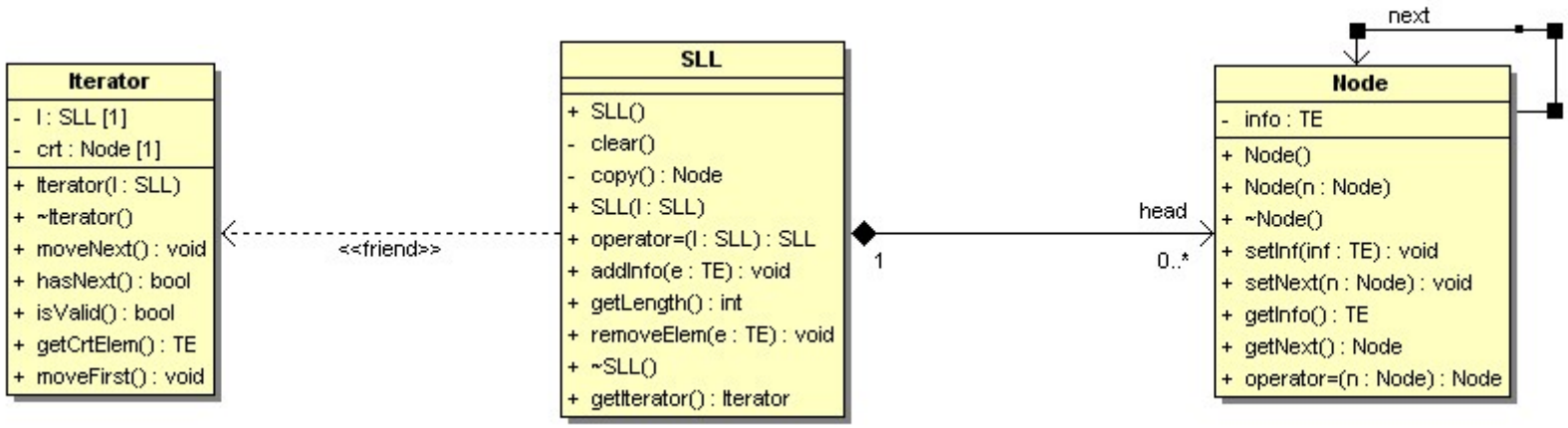
- Locul declarării
  - Iteratori externi
  - Iteratori interni (*true iterators*)

- Capacităţi
  - IO
    - Iteratori de intrare (doar citire, se deplasează înainte)
    - Iteratori de ieşire (doar scriere, se deplasează înainte)
  - Mişcare
    - Înainte (se deplasează doar înainte)
    - Bidirecţionali (se deplasează înainte şi înapoi)
    - Acces aleator (similar unui pointer)

TAD Listă Simplu Înlănțuită

3. Implementare TAD
   - a se consulta subdirectoarele directorului ***UML/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



Temă

- Implementaţi clasa Stivă care conţine:
  - elemente întregi
  - CD-uri (tip, nume, capacitate)

  Scrieţi un program de test pentru utilizarea acestei clase.

- Implementaţi clasa Coadă care conţine:
  - elemente de tip caracter
  - Maşini (tip, putere)

  Scrieţi un program de test pentru utilizarea acestei clase.

Cursul următor

- GUI