Programare orientată pe obiecte

1. Dezvoltarea aplicaţiilor OO
2. Diagrame de clase şi obiecte în UML
3. Aserţiuni

Ciclul de viaţă al software

- Cuprinde toate activităţile de la analiza iniţială până când nu se mai foloseşte – e învechit
- Procesul formal pentru dezvoltarea de software
  - Descrie fazele procesului de dezvoltare
  - Oferă linii de ghidare pentru modul în care trebuie desfăşurate fazele
- Procesul de dezvoltare
  - Analiză
  - Proiectare
  - Implementare
  - Testare
  - Desfăşurare sistematică (deployment)

Analiza

- Se decide ce anume trebuie să facă proiectul
- NU se gândeşte cum îşi va îndeplini programul sarcinile
- Ce rezultă: documentul care cuprinde cerinţele
  - Descrie ce va face programul o dată terminat
  - Manualul de utilizare: spune cum va opera utilizatorul programul
  - Criterii de performanţă

Proiectare. Implementare

- Proiectare
  - Plănuim cum să implementăm sistemul
  - Descoperim structurile care stau la baza problemei de rezolvat
  - Decidem ce clase şi ce metode sunt necesare
  - Ce rezultă:
    - Descrierea claselor şi a metodelor
    - Diagrame care arată relaţiile dintre clase
- Implementare
  - Scriem şi compilăm codul sursă
  - Codul implementează clase şi metode descoperite în faza de proiectare
  - Ce rezultă: un program finalizat

Testare. Desfăşurare sistematică

- Testare
  - Rulăm teste pentru a verifica faptul că programul funcţionează corect
  - Ce rezultă: un raport asupra testelor şi rezultatelor lor
- Desfăşurare sistematică
  - Utilizatorii instalează programul
  - Utilizatorii folosesc programul în scopul în care a fost construit

Proiectarea orientată pe obiecte

1. Descoperim clasele
2. Determinăm responsabilităţile fiecărei clase
3. Descriem relaţiile dintre clase

Unified Modeling Language (UML)

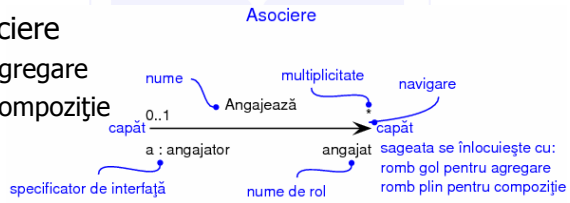
- UML este notaţia internaţională standard pentru analiza şi proiectarea orientată pe obiecte.
- Este definit de Object Management Group (OMG)
- UML 2.0 defineşte treisprezece tipuri de diagrame, împărţite în trei categorii:
  - şase tipuri de diagrame reprezintă structura statică a aplicaţiei;
  - trei reprezintă tipuri generale de comportament;
  - patru reprezintă diferite aspecte ale interacţiunilor
- **Diagramele de structură** cuprind *diagrama de clase, diagrame de obiecte*, diagrama de componente, diagrama de structură compozită, diagrama de pachete şi diagrama de desfăşurare sistematică.

Descoperirea claselor

- O clasă reprezintă un concept util
  - Entităţi concrete: conturi bancare, elipse, produse
  - Concepte abstracte: fluxuri (streams) şi ferestre
- Găsim *clasele* căutând *substantive* în descrierea sarcinii
- Definim comportamentul fiecărei clase
- Găsim *metodele* căutând *verbe* în descrierea sarcinii

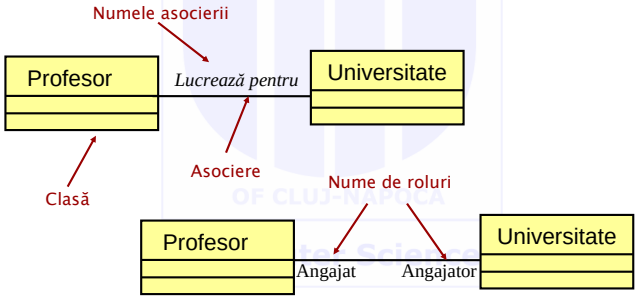
Relaţii între entităţile reprezentate

- Asociere
  - Agregare
  - Compoziţie
- Dependenţă
- Generalizare
- Realizare



Relaţii: Asociere

- Modelează o conexiune semantică între clase

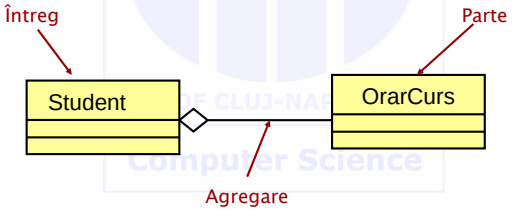


Folosirea asocierilor

- Trei scopuri generale:
  - Pentru a reprezenta o situaţie în care un obiect de o clasă *foloseşte* serviciile unui alt obiect, sau ele îşi folosesc reciproc serviciile – adică un obiect îi trimite mesaje celuilalt sau îşi trimit mesaje între ele.. (În primul caz, navigabilitatea poate fi unidirecţională; în cel de al doilea, ea trebuie să fie bidirecţională.)
  - Pentru a reprezenta agregarea sau compoziţia – unde obiecte de o clasă sunt întregi compuşi din obiecte de cealaltă clasă ca părţi. În acest caz, o relaţie de tip foloseşte este implicit prezentă – întregul foloseşte părţile pentru a-şi îndeplini funcţia, iar părţile pot si ele avea nevoie să folosească întregul.
  - Pentru a reprezenta o situaţie în care obiectele sunt înrudite, chiar dacă nu schimbă mesaje. Aceasta se întâmplă de obicei când cel puţin unul dintre obiecte este folosit în esenţă la stocare de informaţie.

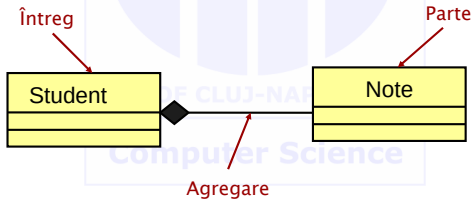
Relaţii: Agregare

- O formă specială de asociere care modelează relaţia parte–întreg între un agregat (întregul) şi părţile sale



Relaţii: compunere

- O formă de agregare cu posesiune *puternică* şi durate de viaţă care coincid
  - Părţile nu pot supravieţui întregului/agregatului



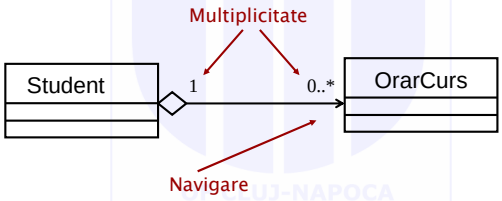
Asociere: Multiplicitate şi navigare

- Multiplicitatea defineşte câte obiecte participă într-o relaţie
  - Numărul de instanţe ale unei clase în raport cu una instanţă a celeilalte clase
  - Specificat pentru fiecare capăt al asocierii
- Asocierile şi agregările sunt implicit bidirecţionale, dar adesea este de dorit să se restrângă navigarea la o singură direcţie
  - Dacă navigarea este restricţionată, se adaugă o săgeată pentru a indica direcţia de navigare

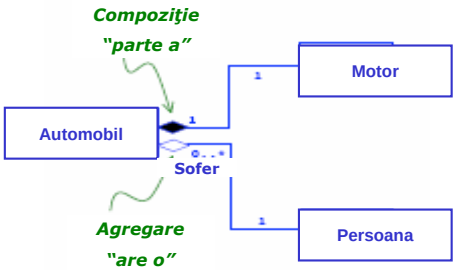
Asociere: multiplicitate

- Nespecificată
- Exact una — 1
- Zero sau mai multe (multe, nelimitat) — 0..* / *
- Una sau mai multe — 1..*
- Zero sau una — 0..1
- Gama specificată — 2..4
- Game multiple, disjuncte — 2, 4..6

Exemplu: multiplicitate şi navigare



Exemple de asocieri



Observaţii

- **Teste pentru relaţii parte-întreg adevărate**
  - » **transitivittate: dacă "A parte a lui B" & "B part a lui C" atunci "A parte a lui C"**
    - "Unghia este parte a degetului, degetul este parte a mâinii; mâna este parte a extremităţii superioare a corpului"
  - **O problemă a unei părţi este o problemă a întregului**
    - O rană la unghie este o rană a corpului
  - "Poziţiile sunt parte a sistemului electric al automobilului. Un defect al poziţiilor este un defect al automobilului"
- **Este-parte-a *e diferit* de**
  - **esteConţinutÎn** : cămăşi, pantaloni,... --- valiza [observaţi ca testul de defectare nu ţine aici: pantalonii stricaţi nu înseamnă valiză stricată]
  - **esteLegatDE** : valiză --- persoană (care o duce)
  - **esteRamurăA** : artera iliacă,... --- aorta
  - **seAflaÎn** : casă... --- stradă

Când să folosim agregarea

- **Ca regulă generală, se poate marca o asociere ca agregare, dacă sunt adevărate următoarele:**
  - Se poate spune că
    - Părţile 'sunt parte' a agregatului
    - sau agregatul 'este compus din' părţi
  - Când ceva deţine sau controlează agregatul, atunci acel ceva deţine sau controlează părţile

Agregare şi compoziţie

Asociere

Obiectele ştiu unul despre altul astfel încât pot lucra împreună

Agregare

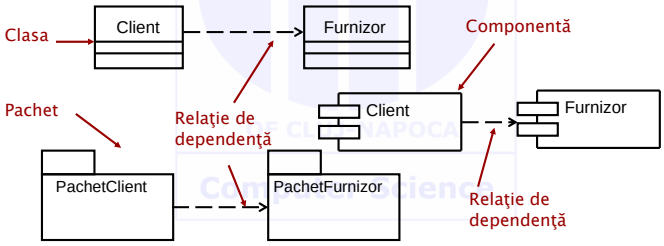
- **Protejează integritatea configuraţiei**
- **Funcţionează ca un singur tot**
- **Controlul se face printr-un obiect– propagarea este în jos**

Compoziţie

Fiecare parte poate fi membru al unui singur obiect agregat

Relaţii: dependenţă

- O relaţie între două elemente ale modelului în care o schimbare în unul dintre elemente **poate** cauza o schimbare în celălalt
- Relaţie nestructurală, de tip "foloseşte"

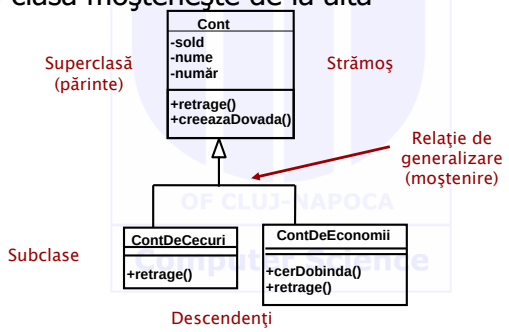


Relaţii: generalizare

- O relaţie între clase în care o clasă partajează structura şi/sau comportamentul uneia sau mai multor clase
- Defineşte o ierarhie de abstracţiuni în care o subclasă moşteneşte de la una sau mai multe superclase
  - Moştenire simplă/de la o singură clasă
  - Moştenire multiplă
- Generalizarea este o relaţie de tipul "este-un-fel-de"

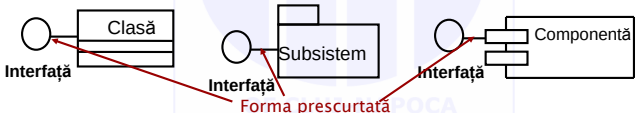
Exemplu: Moştenirea simplă

- O clasă moşteneşte de la alta

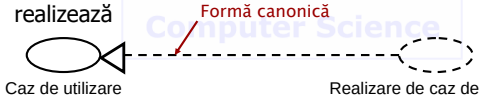


Relaţii: realizare

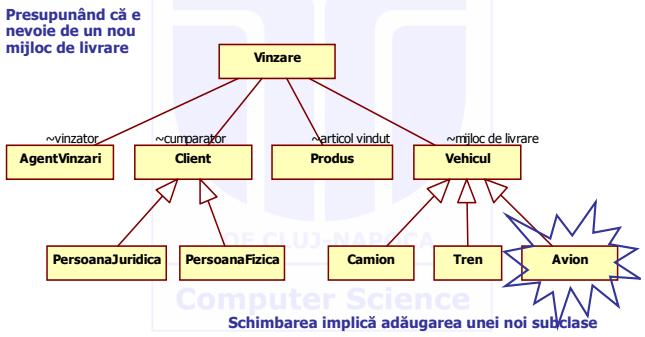
- Un clasificator serveşte pe post de contract pe care celălalt este de acord să-l îndeplinească
- Regăsit între:
  - Interfeţele şi clasificatorii care le realizează



  - Cazurile de folosire (use case) şi colaborările care le realizează

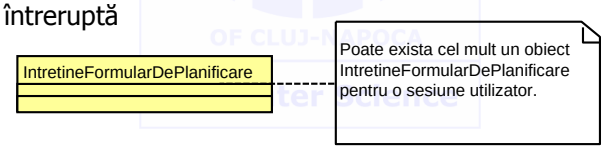


Efectul schimbării cerinţelor



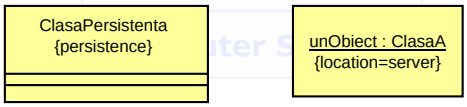
Note (adnotări)

- Se poate adăuga o notă (adnotare) la orice element UML
- Notele se pot adăuga pentru a furniza informaţii suplimentare în diagramă
- Este un dreptunghi cu "ureche de câine"
- Nota poate fi ancorată la un element cu o linie întreruptă



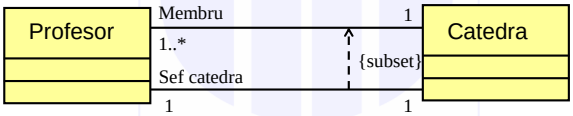
Valori etichetate

- Sunt extensii ale proprietăţilor sau ale anumitor atribute ale unui element UML
- Unele proprietăţi sunt definite de UML
  - Persistenţa
  - Localizarea (d.e., client, server)
- Proprietăţile pot fi create de modelori UML în orice scop



Constrângeri

- Suportă adăugarea de reguli noi sau modificarea regulilor existente



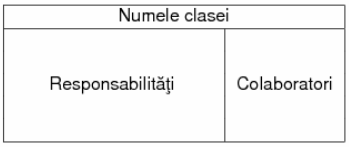
- Această notaţie este folosită pentru a surprinde două relaţii între obiecte de tip Profesor şi obiecte de tip Catedra, unde una dintre relaţii este un subset al celeilalte.
- Arată cum se pot croi diagramele UML pentru a modela corect relaţii exacte

Cartela CRC

- Cartela CRC: descrie o clasă, responsabilităţile şi colaboratorii săi
- Se foloseşte o cartelă pentru fiecare clasă
- Alegem clasa care trebuie să fie răspunzătoare de fiecare metodă (verb)
- Scriem responsabilitatea pe cartela clasei
- Indicăm ce alte clase sunt necesare pentru a îndeplini responsabilitatea (colaboratorii)

Cartela CRC. Relaţii între clase

- Cartela CRC



- Relaţii între clase
  - Moştenire
  - Agregare
  - Dependenţă

Moştenire

- Relaţie de tipul *este-o/un*
- Relaţie între o clasa mai generală (superclasă) şi una mai specializată (subclasă)
- Exemple:
  - Orice cont de economii este un cont bancar
  - Orice cerc este o elipsă (cu lăţime şi înălţime egale)
- Uneori se abuzează de această relaţie
  - Ar trebui să fie clasa **`Anvelopa`** o subclasă a lui **`Cerc`**?
  - Relaţia *are-o* ar fi mai potrivită aici

Agregare

- Relaţie de tipul *are-o/un*
- Obiectele unei clase conţin referinţe la obiectele altei clase
- Folosesc variabile instanţă
  - O anvelopă are un cerc pe post de contur:

```
class Anvelopa {
   . . .
   private String catalogare;
   private Cerc contur;
}
```

- Fiecare automobil are o anvelopă (de fapt are patru)

```
class Automobil extends Vehicul{
   . . .
   private Anvelopa[] anvelope;
}
```


Dependenţă

- Relaţie de tip *foloseşte*
- Exemplu: multe dintre aplicaţiile noastre depind de clasa Scanner pentru a citi intrarea
- Agregarea este o formă mai puternică de dependenţă
- Folosim agregarea pentru a ţine minte un alt obiect între apelurile metodelor

Simboluri UML pentru relaţii

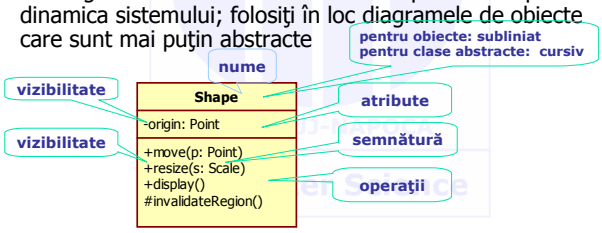
Relaţie	Simbol	Stil de linie	Forma vârfului
Moştenire		Solid	Triunghi
Implementare de interfaţă		Întrerupt	Triunghi
Agregare		Solid	Romb
Dependenţă		Întrerupt	Deschisă

Agregare şi asociere

- Asocierea: o formă mai generală de relaţie între clase
- Se foloseşte de *timpuriu* în faza de proiectare
- O clasă este asociată cu o alta dacă putem naviga de la obiectele unei clase la obiectele celeilalte clase
- Fiind dat un obiect a **`Banca`**, putem naviga la obiecte **`Client`**

Diagramă de clase

- Reprezintă un set de clase, interfeţe, colaborări şi alte relaţii
- Reflectă *proiectul static* al unui sistem
- Poate genera confuzii dacă este folosit pentru a explica dinamica sistemului; folosiţi în loc diagramele de obiecte care sunt mai puţin abstracte

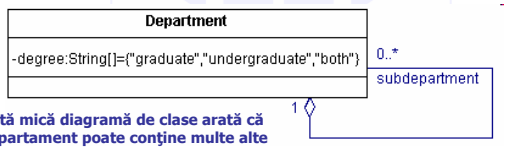


Diagrame de obiecte

- Reprezintă un set de obiecte (instanţe de clase) şi relaţiile dintre acestea.
- Un instantaneu static al unei vederi dinamice a sistemului.
- Reprezintă cazuri reale sau cazuri prototip.
- Foarte utile înainte de dezvoltarea diagramelor de clase.
- Merită salvate ca elaborări ale diagramelor de clase.

Diagrame de obiecte

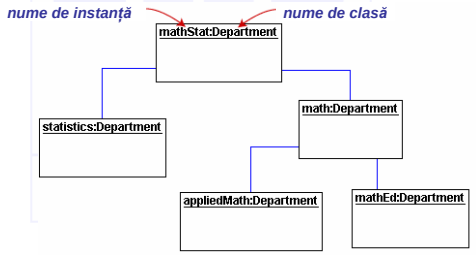
- Utile în explicarea de părţi mici cu relaţii complicate, mai ales în cazul *relaţiilor recursive*
- Fiecare dreptunghi din diagrama de obiecte corespunde unei singure instanţe.
- Numele instanţelor (numele obiectelor) sunt subliniate în diagramele UML.



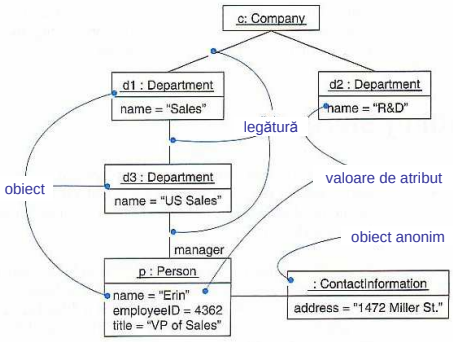
Această mică diagramă de clase arată că un departament poate conţine multe alte subdepartamente.

Diagrame de obiecte

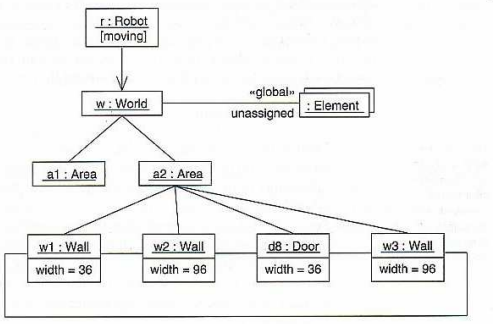
- Diagrama de obiecte de mai jos este o instanţiere a diagramei de clase, clasele fiind înlocuite de exemple concrete
- Numele de clase sau instanţe pot fi omise dacă semnificaţia diagramei rămâne clară.



Alt exemplu de diagramă de obiecte



Alt exemplu de diagramă de obiecte



Procesul de dezvoltare în cinci paşi

- Colectăm cerinţele
- Folosim cartele CRC pentru a determina clasele, responsabilităţile şi colaboratorii
- Folosim diagrame UML pentru a înregistra relaţiile dintre clase
- Folosim **`javadoc`** pentru a documenta comportamentul metodelor
- Implementăm programul

Reguli pentru determinarea claselor

- Între 3 şi 5 responsabilităţi pe clasă
- Nu există clase singuratice
- Feriţi-vă de multe clase mici
- Feriţi-vă de puţine clase mari
- Feriţi-vă de "functoizi" – un functoid este de fapt o funcţie procedurală normală deghizată în clasă.
- Feriţi-vă de clase omnipotente
  - Căutati clase care au "system" sau "controller" în nume!
- Evitaţi arborii de moştenire adânci

Exemplu: factură simplificată

```
                              FACTURA


Maria s.r.l.
str.  Mare nr.  1
Un oraş, 554400



Descriere                                Preţ unitar  Cantitate  Total
Spray XXL                                       8.99          3  26.97
Şervetele Super                                 2.99          4  11.96
Caiet studenţesc                                3.99          2   7.98
Suma de plătit:  46.91
```


Exemplu: factură simplificată

- Clase care vin în minte: **`Factura`**, **`Rind`**, şi **`Client`**
- Este o idee bună să păstrăm o listă de clase candidate
- Folosim brainstorming-ul, pur şi simplu punem toate ideile de clasă în listă
- Le putem tăia pe cel inutile ulterior

Determinarea claselor

- Ţineţi minte următoarele puncte:
  - Clasele reprezintă mulţimi de obiecte cu acelaşi comportament
    - Entităţile cu apariţii multiple în descrierea problemei sunt candidaţi buni pentru obiecte
    - Aflaţi ce au în comun
    - Proiectaţi clasele pentru a surprinde ce este comun
  - Reprezentaţi unele entităţi ca obiecte, iar altele ca tipuri primitive
    - Ar trebui să facem Adresa o clasă sau să folosim un String?
  - Nu toate clasele pot fi descoperite în faza de analiză
  - Unele clase pot exista deja

Tipărirea unei facturi – cerinţe

- Sarcina: tipărirea unei facturi
- Factura: descrie preţurile pentru un set de produse în anumite cantităţi
- Omitem lucrurile mai complicate – aici
  - Date, taxe şi codurile de factură şi de client
- Tipărim factura cu
  - Adresa clientului, toate rândurile, suma de plătit
- Rândul conţine
  - Descriere, preţ unitar, cantitatea comandată, preţul total
- Pentru simplitate nu creăm interfaţa cu utilizatorul
- Programul de test: adaugă rânduri în factură şi apoi o tipăreşte

Tipărirea unei facturi – cartele CRC

- Descoperim clasele
- Substantivele identifică clasele posibile

Factura
Adresa
Rând
Produs
Descriere
Preţ
Cantitate
Total
Suma de plătit

Tipărirea unei facturi – cartele CRC

- Analizăm clasele

```
Factura
Adresa
Rând          // Înregistrează produsul şi cantitatea
Produs
Descriere     // Câmp al clasei produs
Pret          // Câmp al clasei produs
Cantitate     // Nu este un atribut al unui Produs
Total         // Calculat, nu se memorează
Suma de plătit // Calculată, nu se memorează
```

- Clasele după un proces de eliminare

Factura
Adresa
Rând
Produs

Motive pentru rejectarea unei clase candidate

Semnal	Motiv
Clasa cu nume verb (infinitiv sau imperativ)	Poate fi o subrutina, nu o clasa
Clasa cu o singura metodă	Poate fi o subrutina, nu o clasa
Clasă descrisa ca "efectuează" ceva	Poate să nu fie o abstracţiune propriu-zisă
Clasă fără metode	Poate fi o informaţie opacă, nu un TDA, sau poate fi un TDA la care s-au omis rutinele
Clasă cu zero sau foarte puţine atribute (dar care moşteneşte de la părinţi)	Poate fi un caz de "taxomanie"
Clasă care acoperă câteva abstracţiuni	Ar trebui divizată în mai multe clase, câte una pentru fiecare abstracţiune

Cartelele CRC pentru tipărirea unei facturi

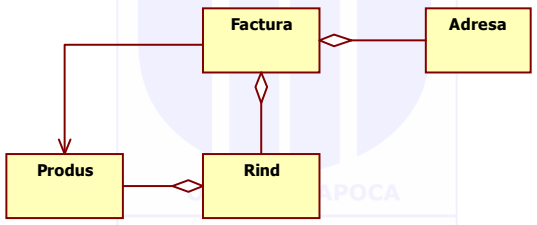
- Atât **Factura** cât şi **Adresa** trebuie să se poată autoformata – responsabilităţi:
  - **Factura** *formatează factura* şi
  - **Adresa** *formatează adresa*
- Adăugăm colaboratori pe cartela facturii: **Adresa** şi **Rind**
- Pentru cartela **Produs** – responsabilităţi: *obţine descrierea, obţine preţul unitar*
- Pentru cartela CRC **Rind** – responsabilităţi: *formatează articolul, obţine preţul total*

Cartelele CRC pentru tipărirea unei facturi

- **Factura** trebuie populată cu produse şi cantităţi:

Factura	
formatează factura *adaugă un produs şi o cantitate*	Adresa Rind Produs

Tipărirea unei facturi – diagrama UML



Instrumente pentru realizarea diagramelor UML

- ArgoUML:
  - http://argouml.tigris.org
  - rulează pe orice platformă Java
- StarUML
  - http://staruml.sourceforge.net/en/
  - proiect "open source"
- Poseidon for UML Community Edition
  - http://gentleware.com/downloadcenter.0.html
- Multe produse comerciale

Tipuri de specificaţii

- Diagrame de clase
- Diagrame de obiecte
- Diagrame de activităţi (diagrame de curgere a controlului)
- Aserţiuni (precondiţii, postcondiţii, invarianţi)
  - Altele
- Reţineţi că primele trei sunt specificaţii incomplete

Specificarea claselor

- O specificaţie software indică sarcina (sau un aspect al sarcinii) care se presupune a fi efectuată la execuţia software respectiv
- O specificaţie de clasă defineşte semantica (comportamentul) unei clase prin:
  - invariant de clasă care descrie ceea ce este întotdeauna adevărat pentru obiectele clasei.
  - specificaţii pentru fiecare dintre metodele clasei.
- Fiecare *specificaţie de metodă* constă din
  - o precondiţie (opţional),
  - o clauză modifică (opţional), şi
  - o postcondiţie.

Specificarea metodelor

- O *precondiţie* declară condiţiile care sunt necesare pentru ca metoda să se execute în mod corespunzător
- O clauză *modifică* reprezintă o listă de obiecte care ar putea fi modificate prin execuţia metodei.
- O *postcondiţie* declară ce este adevărat atunci când metoda şi-a finalizat execuţia

Aserţiune

- O *aserţiune* este o declaraţie a unui fapt care este presupus adevărat relativ la o locaţie/locaţii din cod. Exemplu

```
// assert: str este String  si str.length > 2
char firstChar, secondChar, bigChar;
firstChar = str.charAt(0);
secondChar = str.charAt(1);
if (firstChar > secondChar)   {
  bigChar = firstChar;
} else {
  bigChar = secondChar;
}
/* assert:
  str.length > 2
  and (str.charAt(0) > str.charAt(1)
          implies bigChar == str.charAt(0))
  and (str.charAt(0) ≤ str.charAt(1)
          implies bigChar == str.charAt(1)) */
```


Notarea aserţiunilor

- Aserţiunile sunt bazate pe logică şi anumite notaţii din program (adică referinţe la variabile şi, posibil, apeluri de metode non-void).
- Aserţiunile NU trebuie să conţină verbe de acţiune
- Operatori logici
  not *SubAssertion1* – sub-aserţiunea trebuie să fie falsă.
  SubAssertion1 ***and*** *SubAssertion2* – ambele sub-aserţiuni trebuie să fie adevărate.
  SubAssertion1 ***or*** *SubAssertion2* – una sau amândouă sub-aserţiunile sunt adevărate.
  SubAssertion1 ***implies*** *SubAssertion2* – când prima sub-aserţiune este adevărată şi cea de a doua trebuie să fie adevărată

Notarea aserţiunilor

- Altă notaţie logică, cuantificarea, permite exprimarea aserţiunilor referitoare la structuri de date.
- Cuantificarea universală
  - ***forAll*** (*tip var : condiţieLimită | SubAserţiune*)
  - Exemplu:

forAll (Integer j : 0≤j≤2 | arr1[j] > 0)

semnificaţie: arr1[0] > 0 ***and*** arr1[1] > 0 ***and*** arr1[2] > 0

Notarea aserţiunilor

- Cuantificarea existenţială
  - ***exists*** (*tip var : condiţieLimită | SubAserţiune*)
  - Exemplu:

exists (Integer j : 0≤j≤2 | arr1[j] == 5)

semnificaţie: arr1[0] ==5 ***or*** arr1[1] == 5 ***or*** arr1[2] == 5

Exemple de cuantificări

- Fie două tablouri de double: **a1** şi **a2** şi **a1.length == a2.length == 4**

forAll (Integer r : 0 ≤ r ≤ 3 | a1[r] < a1[r+1])
forAll (Integer w : 0 ≤ w ≤ 3 | a1[w] == a2[w])
exists (Integer k : 0 ≤ k ≤ 3
| a1[k] == 22 ***and*** a2[k] == 22)
exists (Integer k : 0 ≤ k ≤ 3
| (a1[k] < 0
and forAll (Integer j : k < j ≤ 3 | a2[k] == a1[j])))
forAll (j,k : 0 ≤ j,k ≤ 3 ***and*** j != k | a1[j] != a2[k])

Unde se pun aserţiunile

- Locuri posibile
  - Invariantul clasei
  - Postcondiţia metodei
  - Precondiţia metodei
  - Invariantul de buclă

Exemple de aserţiuni

```
/** invariant de clasă
   distanceInMiles > 0  and  timeInSeconds > 0 */
public class LapTime   {
    private double distanceInMiles, timeInSeconds;

    /** pre:  d > 0  and  t > 0
        post: distanceInMiles == d and timeInSeconds == t  */
    public LapTime(double d, double t)   {
        distanceInMiles = d;
        timeInSeconds = t;
    }

    /** post:  distanceInMiles == 60
          and   timeInSeconds == 3600 */
    public void setTo60MPH()   {
       distanceInMiles = 60;
       timeInSeconds = 3600;
    }
}
```


Notaţii speciale pentru postcondiţii

- Valoare returnată (rezultat)

```
// Within LapTime class
/** post: result == distanceInMiles / (timeInSeconds*3600)
 */
public double milesPerHour()   {
    double velocity;
    velocity = distanceInMiles/(timeInSeconds*60*60);
    return velocity
}
```

- Valoare anterioară (@pre)

```
// Within LapTime class
/** post:  distanceInMiles == distanceInMiles@pre * 2 */
public void doubleTheMileage()   {
    distanceInMiles = distanceInMiles * 2;
}
```


Proiectarea prin contract

- Apelantul metodei garantează...
  - precondiţia & invariantul clasei (la momentul apelului metodei)
- Metodei i se cere să asigure...
  - postcondiţia & invariantul clasei (la momentul revenirii din metodă)
- Anexă: o clauză "modifică" poate stipula ce modificări sunt permise

Legături utile

- OMG UML Resource Pages
  - http://www.uml.org
- IBM UML Resource Center
  - http://www-306.ibm.com/software/rational/uml/
- Practical UML: A Hands on introduction for developers
  - http://bdn.borland.com/article/0,1410,31863,00.html
- Robert Martin: UML Class Diagrams for Java Programmers
  - http://www.phptr.com/articles/article.asp?p=336264&seqNum=2&rl=1
- UML by Examples
  - http://www.geocities.com/siliconvalley/network/1582/uml-example.htm
- Holub Associates: UML Reference Card
  - http://www.holub.com/goodies/uml
- Visual Case Tool - UML Tutorial
  - http://www.visualcase.com/tutorials/class-diagram.htm