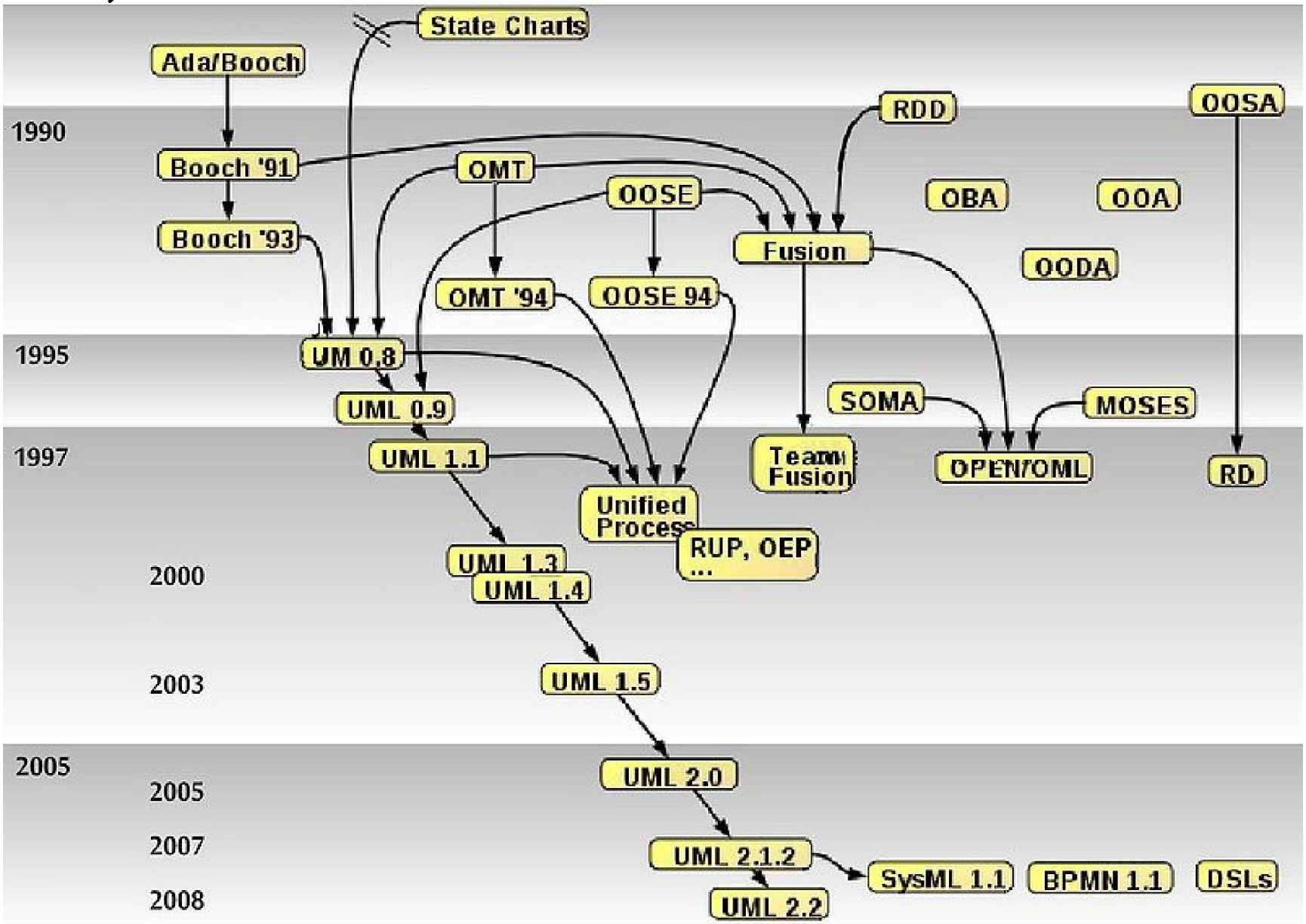
Reprezentarea UML a claselor

Ce este UML?

- **UML** acronim pentru ***U****nified* ***M****odeling* ***L****anguage (limbaj de modelare unificat).*
- Este un limbaj vizual de modelare utilizat pentru *specificarea, construirea şi documentarea sistemelor de aplicații orientate obiect* şi nu numai.
- Un limbaj cu ajutorul căruia se pot construi (descrie) modele.
- Un model surprinde un anumit aspect al unui program.
- Fiecare model poate fi descris la diferite nivele de abstractizare.
- Fiecărui model îi corespunde o diagramă.

Evoluție UML



UML - Istorie

- *Octombrie 1994* – a început în mod oficial dezvoltarea UML prin unificarea limbajelor Booch şi OMT- *Object Modeling Technique* (Rational Software).
- *Octombrie 1995* – apare versiunea preliminară 0.8 a *Unified Method*.
- *Iunie 1996* – se publică versiunea 0.9 a UML (include şi OOSE – *Object Oriented Software Engineering*).
- *Ianuarie 1997* – UML 1.0 a fost propus spre standardizare în cadrul OMG (*Object Management Group*).
- *Noiembrie 1997* – versiunea UML 1.1 a fost adoptată ca standard de către OMG.
- *Iunie 2015* – a fost adoptată ultima versiune UML 2.5.
- Resurse oficiale UML: `http://www.uml.org`

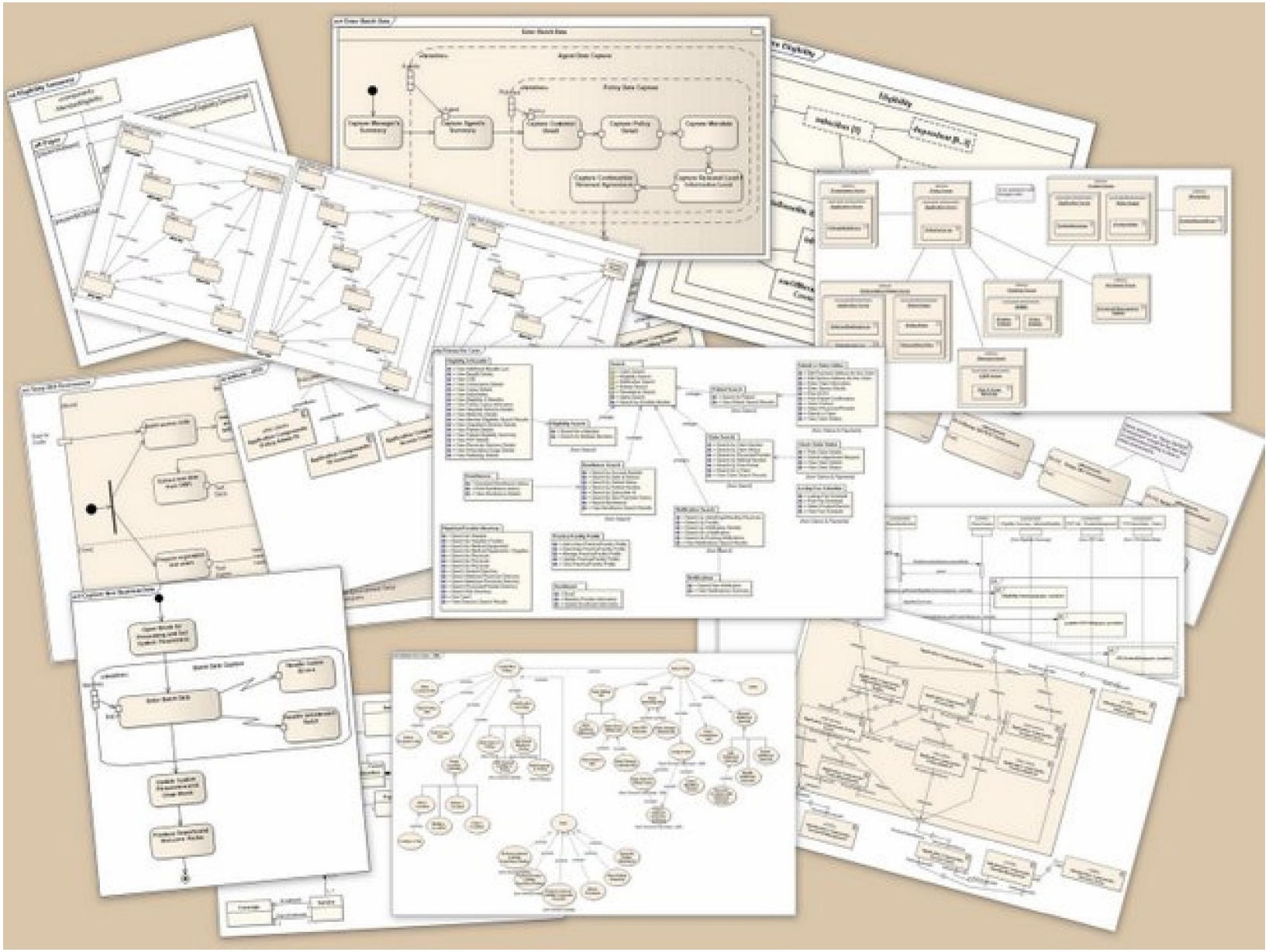
De ce UML?

- este un standard deschis.
- surprinde întreg ciclul de viață al dezvoltării software-ului.
- acoperă multe tipuri de aplicații.

Diagrame UML

- O **diagramă** este o prezentare grafică a unei mulțimi de elemente, cel mai adesea exprimată ca un graf de noduri (*elemente*) şi arce (*relații*).
- Tipuri de diagrame UML:
  - **Diagrame ale cazurilor de utilizare (Use Case);**
  - **Diagrame de clase;**
  - **Diagrame de obiecte;**
  - **Diagrame de secvență;**
  - **Diagrame de stare;**
  - **Diagrame de colaborare;**
  - **Diagrame de activitate.**

Diagrame UML



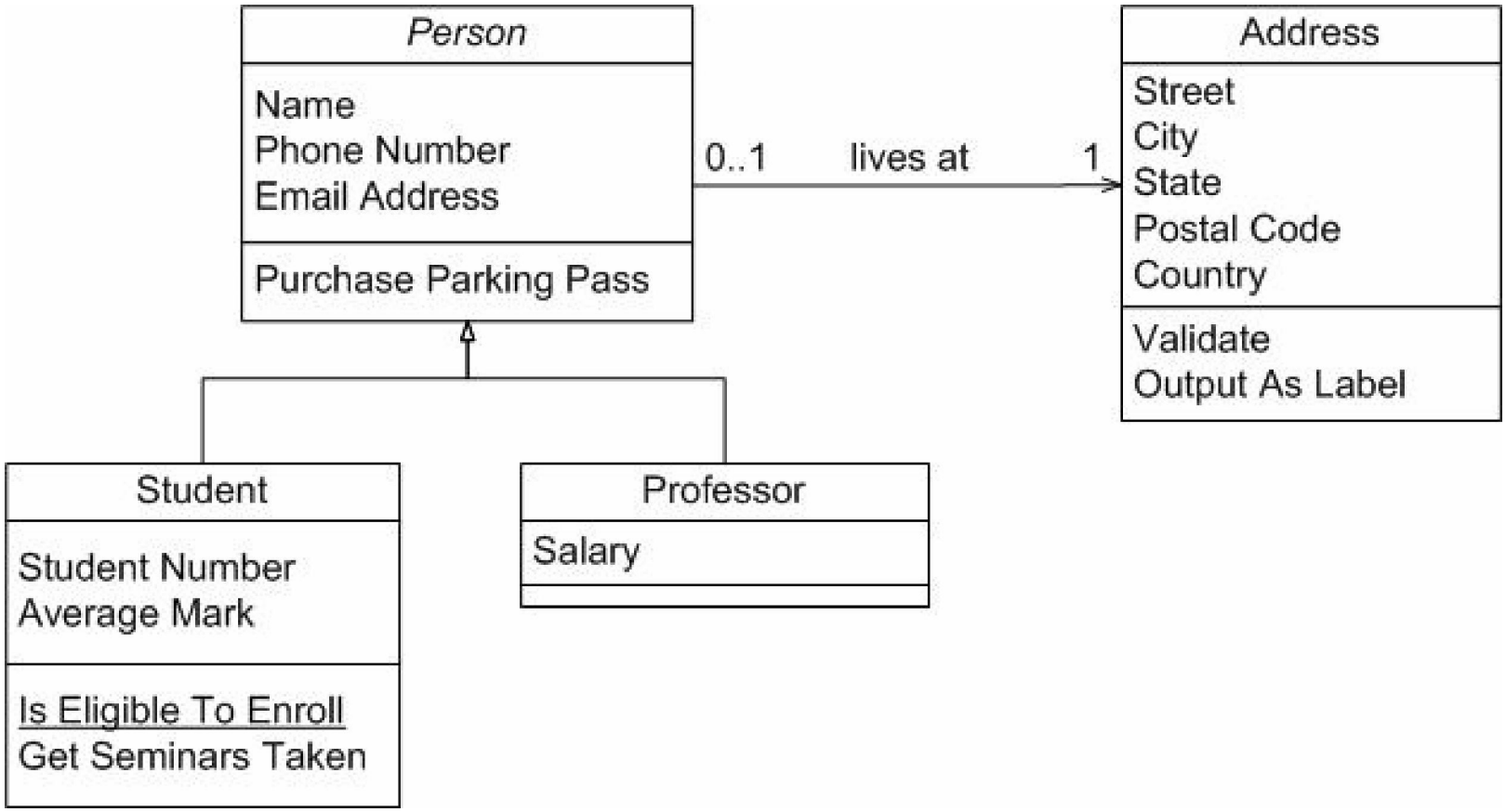
Reprezentarea UML a claselor

Diagrama de clase

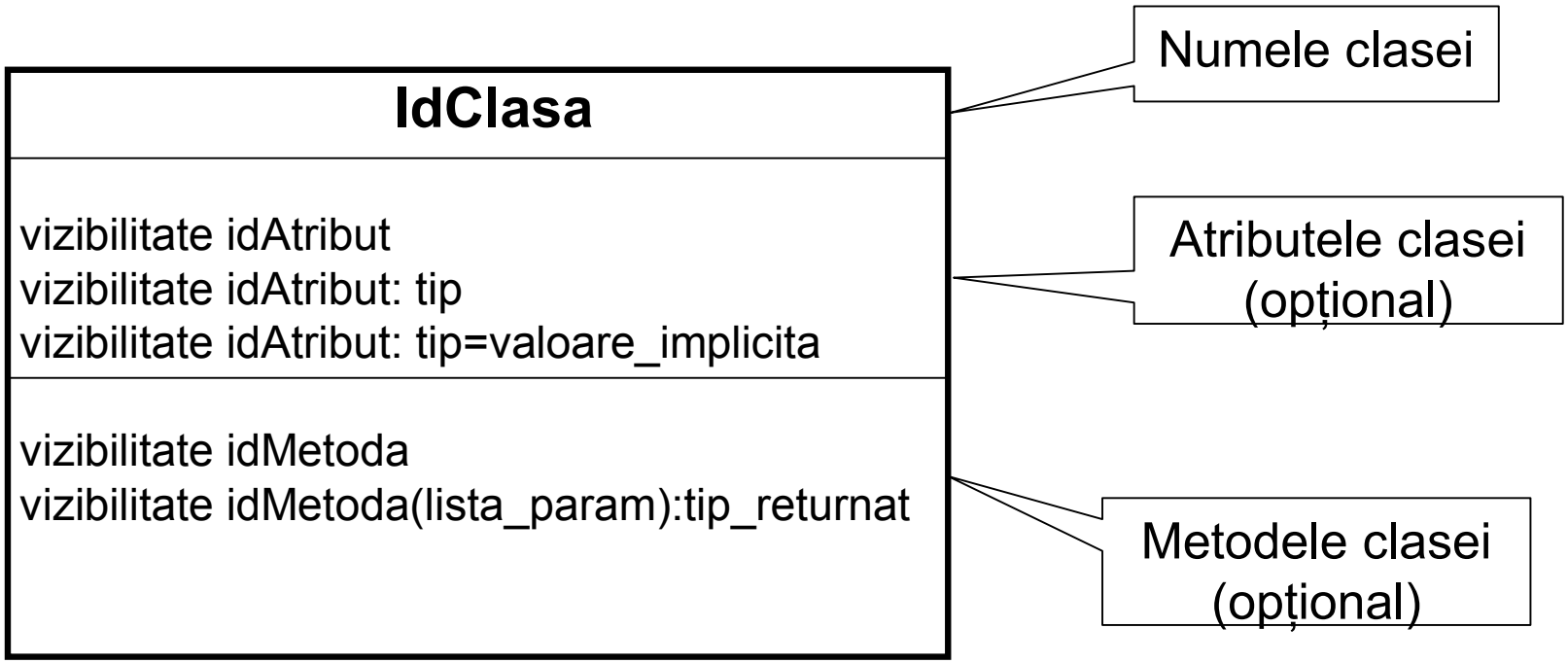
- *Diagramele de clase* sunt folosite în modelarea orientată obiect pentru a descrie structura statică a sistemului, modul în care este el structurat.
- Oferă o notație grafică pentru reprezentarea:
  - **claselor** - entități ce au caracteristici comune;
  - **relațiilor** - relații dintre două sau mai multe clase.
- Modelează vocabularul sistemului ce trebuie dezvoltat.
- Surprinde conexiunile semantice sau interacțiunile ce se stabilesc între elementele componente.
- Este folosită pentru a modela structura unui program.

Diagrama de clase

- cuprinde *clase* (*atribute*, *metode*) şi *relațiile dintre clase*.



Reprezentarea unei clase



sau

Specificarea atributelor

- Sintaxa specificării unui atribut din compartimentul corespunzător este următoarea:

```
vizibilitate idAtribut : tip = valoare_implicita
```

unde:

- *vizibilitate* reprezintă protecția atributului şi poate să aibă una din următoarele valori:
  - **+** = public,
  - **-** = privat şi
  - **#** = protected. (opțional)
- *idAtribut* este identificatorul atributului;
- *tip* – este tipul acestuia;
- *valoare_implicita* reprezintă valoarea inițială a atributului (opțională).

Specificarea metodelor

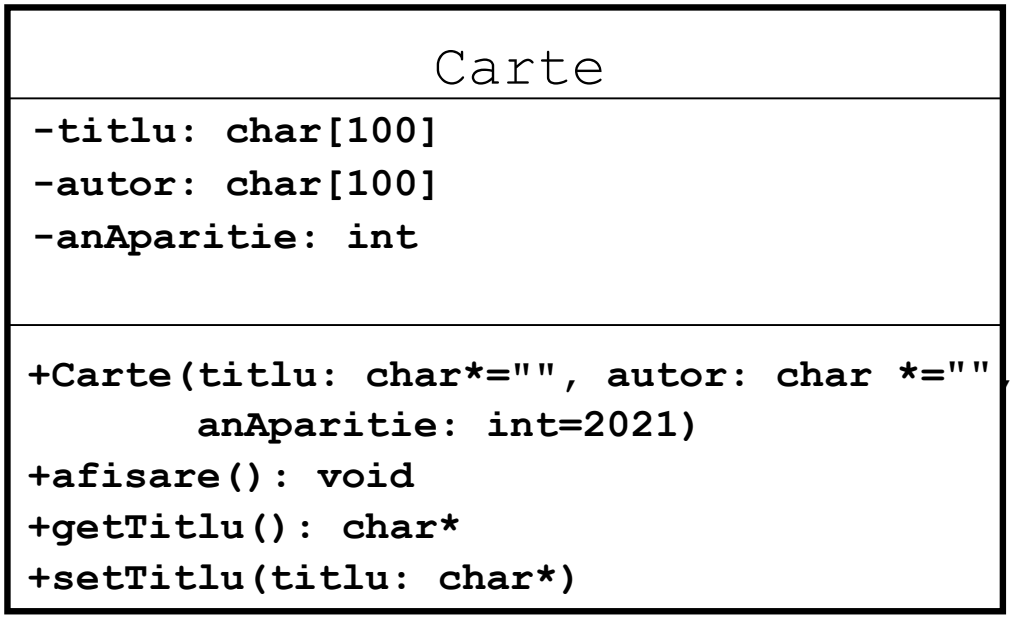
- Sintaxa specificării unei metode sau operații din compartimentul metodelor este următoarea:

```
vizibilitate idMetoda(idP1:tip1, ..., idPn:tipn):tip_returnat
```

unde:

- *vizibilitate* reprezintă protecția metodei. Poate să aibă una din următoarele valori:
  - **+** = public,
  - **-** = privat şi
  - **#** = protected. (opțional)
- *idMetoda* este identificatorul metodei;
- *idP1,..., idPn* sunt parametrii metodei;
- *tip1, ..., tipn* sunt tipurile parametrilor;
- *tip_returnat* reprezintă tipul valorii returnate de metodă.

Diagrama de clase. Exemplu




```
class Carte {
  private:
    char titlu[100];
    char autor[100];
    int anAparitie;
  public:
    Carte(const char *titlu = "",
          const char *autor = "",
          int anAparitie = 2021);
    void afisare();
    char* getTitlu();
    void setTitlu(const char *titlu);
};
```

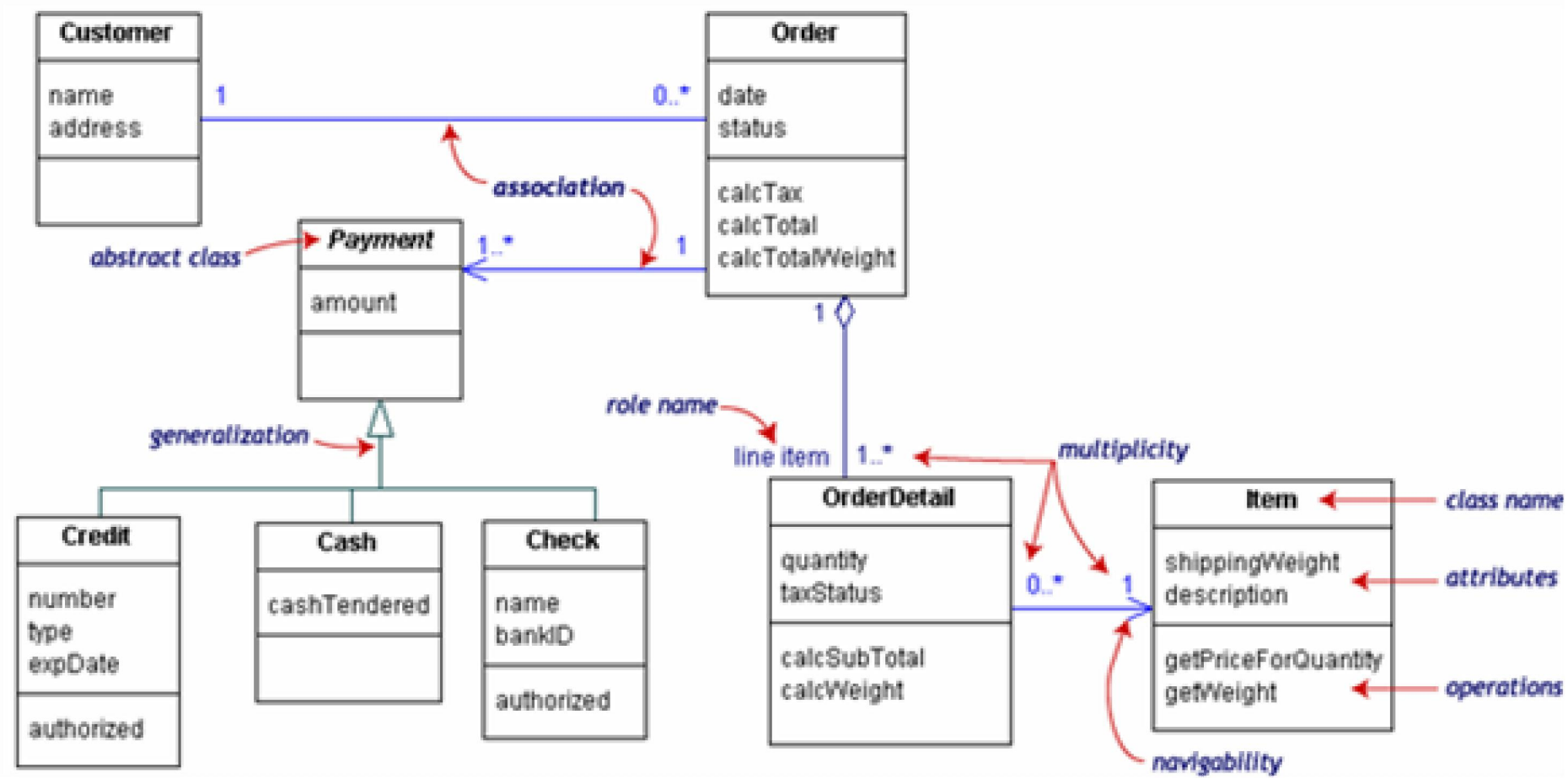
Relații

- Toate relațiile din cadrul diagramei de clase sunt reprezentate grafic printr-o succesiune de segmente orizontale sau verticale ce leagă o clasă de alta.
- *Relațiile* pot fi:
  - asocieri;
  - generalizări;
  - dependențe;
  - relații de rafinare.
- ***Asocierea*** este o conexiune între clase, ceea ce înseamnă o legătură semantică între obiectele claselor implicate în relație.

Relații

- **Asocierea** - este o relație liberă în cadrul căreia obiectele unei clase cunosc obiectele unei alte clase (relație *has-a*).
- **Agregarea** - este o relație de tip parte-întreg (relație *a-part-of*).
- **Compunerea** - o relație similară cu agregarea, însă mai strictă (relație *contains*).
- **Moştenirea** - este o relație de generalizare-specializare (relație *is-a*).

Diagrama de clase. Exemple



Tipuri de relații

- **Relație reflexivă**: obiecte ale aceleiaşi clase se află în relație unele cu altele.
- **Relație directă**: indică o relație direcțională reprezentată printr-o linie orientată cu o săgeată, ce reprezintă o curgere direcțională întreg-parte.

Relația de asociere

- **Asocierea** este identificată prin intermediul expresiei "*has-a*“.
- Asocierea este o relație statică
- Exemple: *O maşină are un şofer. Un student se înscrie la un curs.*
- *Multiplicitatea* - se aplică clasei adiacente şi este independentă de valoarea multiplicității celeilalte părți a asocierii.
- Implementare: *variabile membre*, ca pointeri sau referințe către obiectul asociat.

Notație	Semnificație
1	unul
*	zero sau mai mulți
0..5	între zero şi cinci
0..4,6,10	între zero şi patru, şase sau zece
-	Numai unul (default)

Relația de asociere. Exemplu

- Relația *Student – Disciplină*
  - **`Student`**: un student urmează 0 sau mai multe discipline, cunoaşte disciplinele pe care le urmează.
  - **`Disciplina`**: o disciplină poate fi urmată de mai mulți studenți, **`nu`** cunoaşte studenții care o urmează.

Relația de asociere. Exemplu

- Relația *Disciplină – Profesor*
  - **`Disciplina`**: o disciplină este predată de 0 sau mai mulți profesori, iar o disciplină îşi cunoaşte titularul.
  - **`Profesor`**: un profesor poate preda mai multe discipline, şi cunoaşte disciplinele pe care le predă.

Diagrama de clase. Exemple

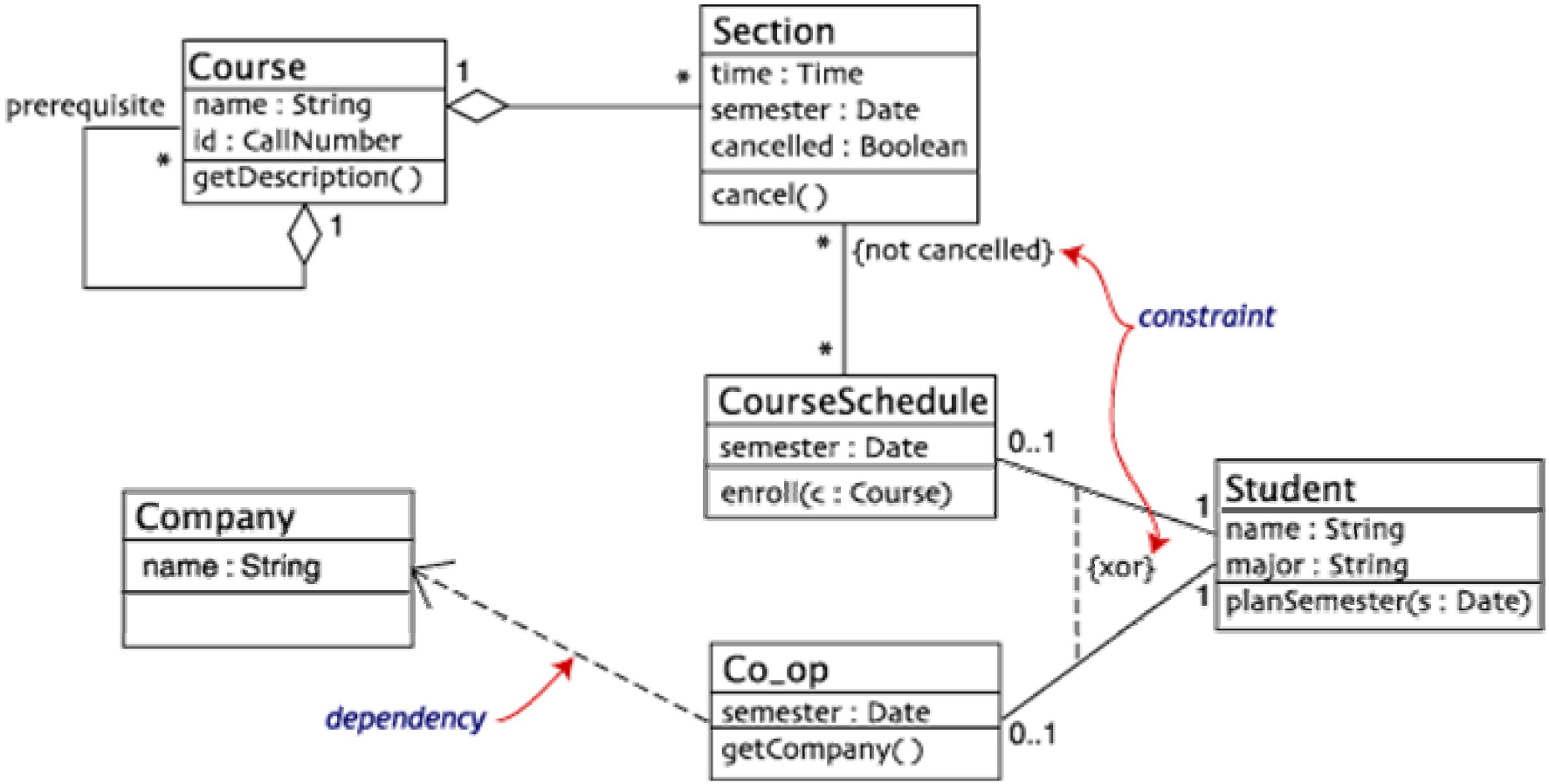
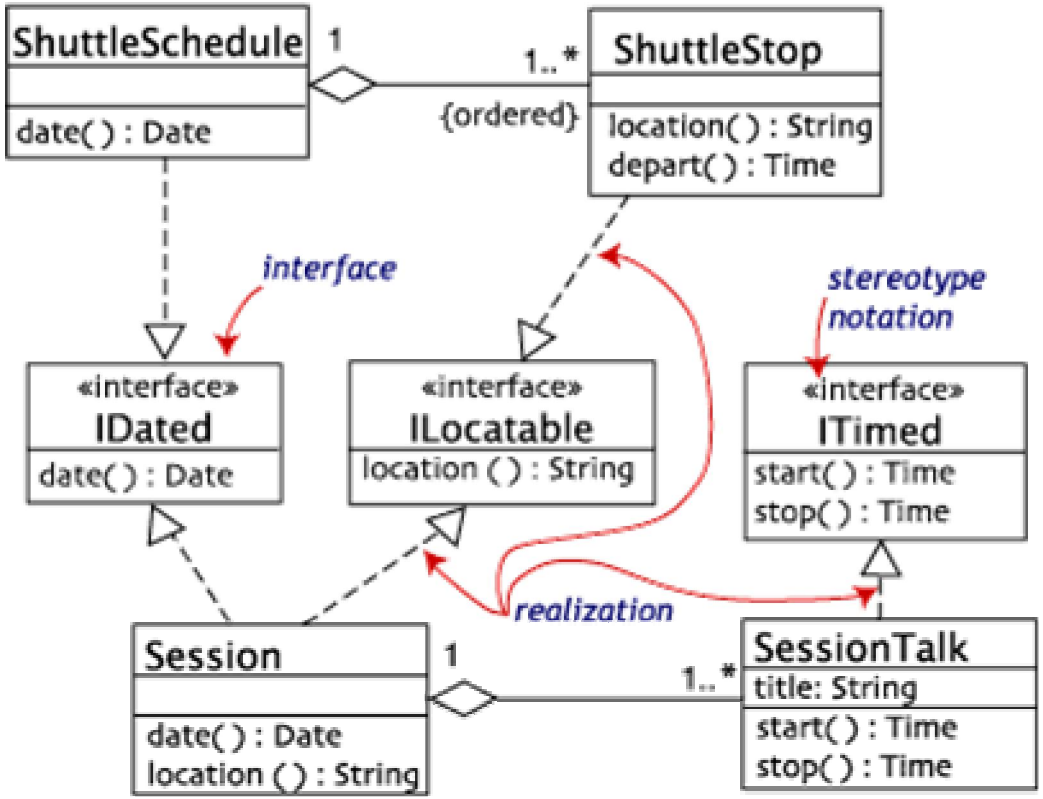
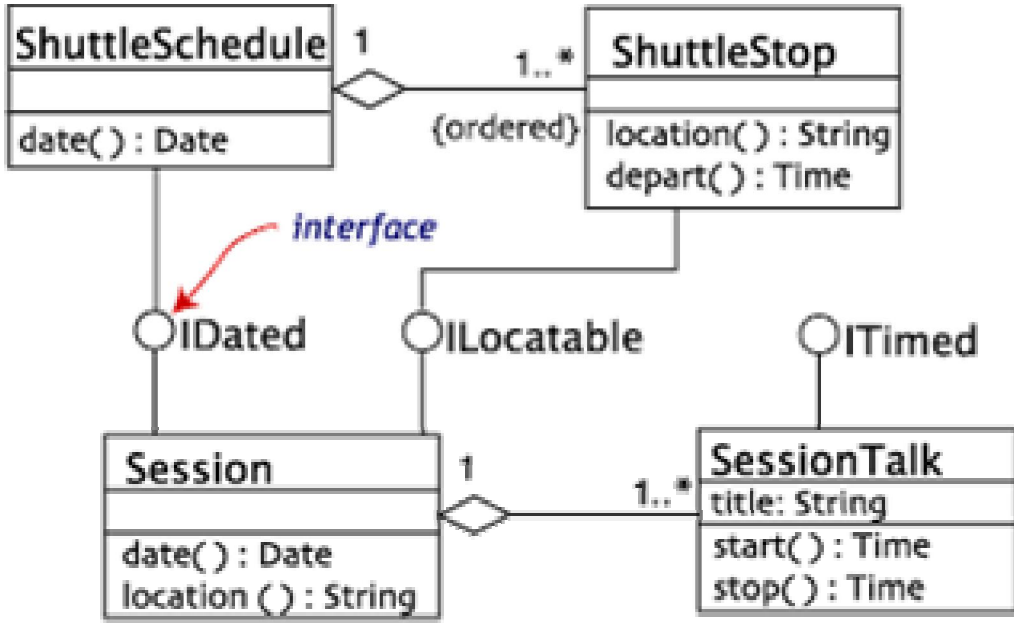


Diagrama de clase. Exemple





Model al domeniului

- Este o reprezentare simplificată:
  - a *conceptelor* prezente în domeniul problemei;
  - a *relațiilor* ce se stabilesc între aceste concepte;
  - a *atributelor* specifice conceptelor.

Identificarea conceptelor

- se face pornind de la *specificații* (*scenarii de utilizare*) şi analizând descrierea problemei (se caută *substantive*).
- construcția modelului se face *incremental.*
- la fiecare pas se consideră un singur scenariu => model parțial care reflectă acel scenariu, nu mai mult.
- se poate întâmpla ca anumite concepte să nu aibă *atribute* (au rol pur *comportamental*, nu şi *informațional*).

Identificarea relațiilor

- A *este o parte fizică/logică* a lui B (`aripa - avion`) => **compoziție**.
- A *este conținut fizic/logic* în B (`pasager - avion`, `ora de curs - orar`) => **agregare**.
- A *comunică* cu B (`agentie de rezervări - pasager`) => **asociere**.
- A *îl cunoaşte* pe B (`student - profesor`) => **asociere**.

Identificarea relațiilor

- A *este o generalizare* a lui B / B *este un tip de* A / B *este o particularizare* a lui A (`forma geometrică -patrat`) => **generalizare** (**moştenire**).
- A *este ca* B / A *este similar cu* B (`mere - pere`: ambele sunt fructe) => conduce la apariția unei **abstracțiuni**.
- A *îl foloseşte pe* B (`program de editat imagini - biblioteca grafica`) => **dependență**.

Observații

- Este necesar să ne concentrăm asupra relațiilor absolut necesare (*need to know*).
- Identificarea conceptelor este mai importantă decât identificarea asocierilor.
- Prea multe relații fac modelul confuz şi greu de înțeles.
- Evitați să arătați relațiile redundante.

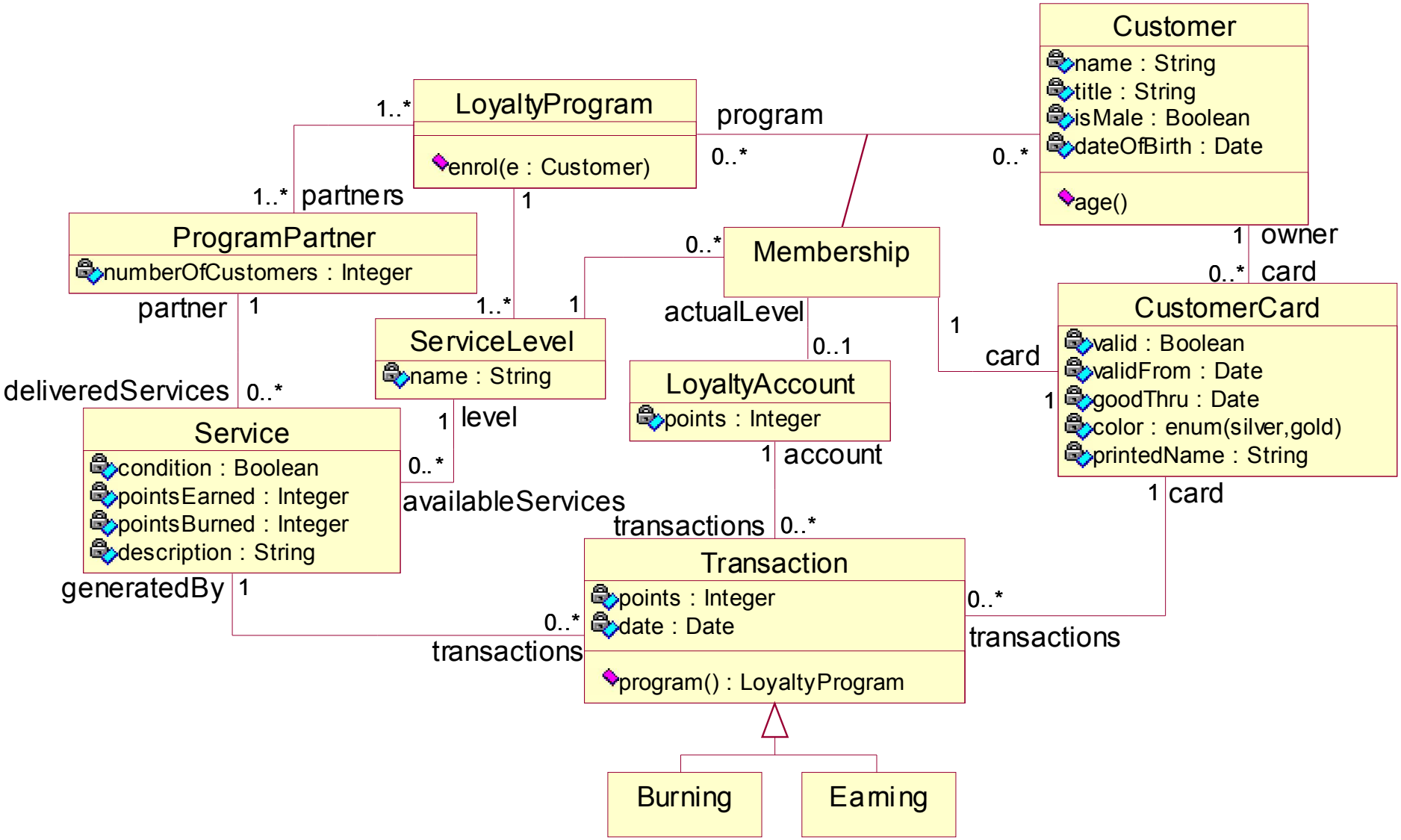
Identificarea atributelor

- sunt informații ce trebuie memorate (se extrag din cerințe).
- ar trebui să aibă tipuri de date simple:
  - predefinite: `Boolean`, `Date`, `Number`, `String`, `Time`;
  - definite de utilizator: `Address`, `PhoneNumber`, `Color`, `Point`, tipuri enumerative.
- *eroare frecventă*: reprezint ceva ca atribut când de fapt ar trebui să fie un concept.

Observații

- Dacă X nu poate fi privit, în domeniul problemei, ca număr sau text, atunci probabil că X este un concept, nu un atribut.
- Acest lucru se întâmplă când:
  - X este compus din secțiuni separate (ex. număr de telefon, numele unei persoane);
  - există operații care sunt în mod uzual asociate cu X, cum ar fi parsare sau validare (ex. CNP, email);
  - X are alte atribute (ex. un preț promoțional poate avea o dată de început şi o dată de sfârşit);
  - X este o cantitate ce posedă o unitate de măsură (ex. plata se face utilizând o anumită valută).

Diagrama de clase - “Royal & Loyal”



Reutilizarea Codului

Reutilizarea codului

- În programarea orientată pe obiecte reutilizarea codului se poate realiza prin:
  - *Compunere* (*Agregare*) – includerea unor obiecte în cadrul altor obiecte;
  - *Moştenire* – posibilitatea de a defini o clasă extinzând o altă clasă deja existentă;
  - *Clase şi funcții template*.

Agregarea

- **Agregare** = definirea unei clase noi ce include una sau mai multe date membre ce au ca tip, clase deja definite.
- Ne permite construirea de clase complexe pornind de la clase mai simple.
- Definițiile pentru agregare şi compunere au în vedere următoarele caracteristici:
  - *Accesibilitate*: obiectul parte este accesibil numai prin intermediul obiectului întreg.
  - *Durată de viață*: obiectul parte este distrus atunci când obiectul întreg este distrus.
  - *Compartimentare*: obiectul întreg este în întregime compus din obiectele parte.

Agregare. Constructori (I)

- Dacă o clasă A conține ca date membru obiecte având ca tip alte clase (`C1, …, Cn`) atunci constructorul clasei A va avea suficienți parametri pentru a inițializa datele membru ale claselor `C1, …, Cn`.
- La crearea unui obiect al clasei A se vor apela mai întâi constructorii claselor `C1, …, Cn` pentru a se inițializa obiectele incluse în obiectul clasei A şi apoi se vor executa instrucțiunile constructorului clasei A.

Agregare. Constructori (II)

- Apelul constructorilor claselor `C1, …,Cn` se poate face:
  - *explicit la definirea constructorului clasei A*

```
class A {
   C1 c1;
   …
   Cn cn;
   …
   A(lista de parametri);
};
A::A(lista de parametri):c1(sub_lista_param1), ...,
   cn(sub_lista_paramn) {
  instructiuni
}
```

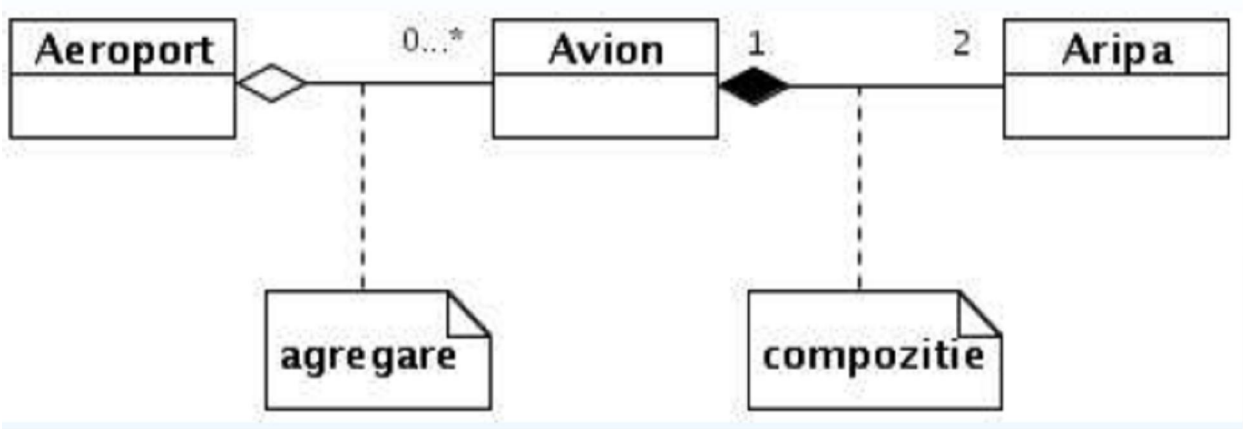
  - *automat de către compilator* – în acest caz se apelează constructorul implicit al acelor clase.

Agregare. Destructori

- La distrugerea unui obiect al clasei A se va executa mai întâi destructorul clasei A, apoi se vor apela destructorii claselor `C1, …,Cn` în *ordinea inversă* apelurilor constructorilor.

Tipuri de agregare

- Există două variante de agregare, diferențiate de relația dintre agregat şi subobiecte:
  - **Compoziția** – subobiectele agregate aparțin exclusiv agregatului din care fac parte, iar durata de viață coincide cu cea a agregatului.
  - **Agregarea propriu-zisă** – subobiectele au o existență independentă de agregatele ce le partajază. Mai mult, un obiect poate fi partajat de mai multe agregate.

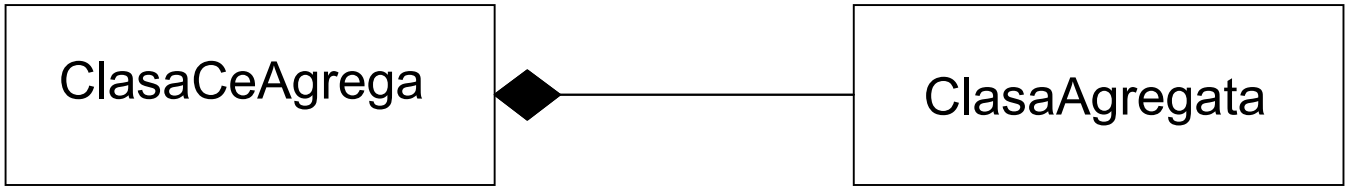


Agregare propriu-zisă

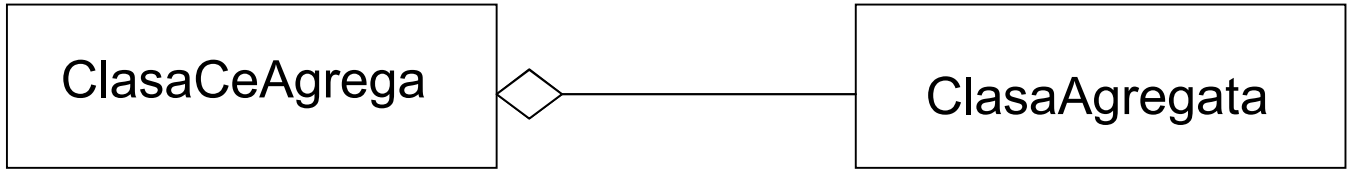
- Agregarea propriu-zisă este *o relație între parte şi întreg* în care o parte poate fi independentă de întreg şi/sau o parte poate fi comună între două instanțe ale aceleiaşi clasei întreg.
- Agregarea se poate identifica prin "*is part of*" (*este o parte a lui*).
- Agregarea nu poate fi *circulară*: un obiect nu poate fi parte a lui însuşi.

Agregare. Reprezentare UML

- **Compoziția**



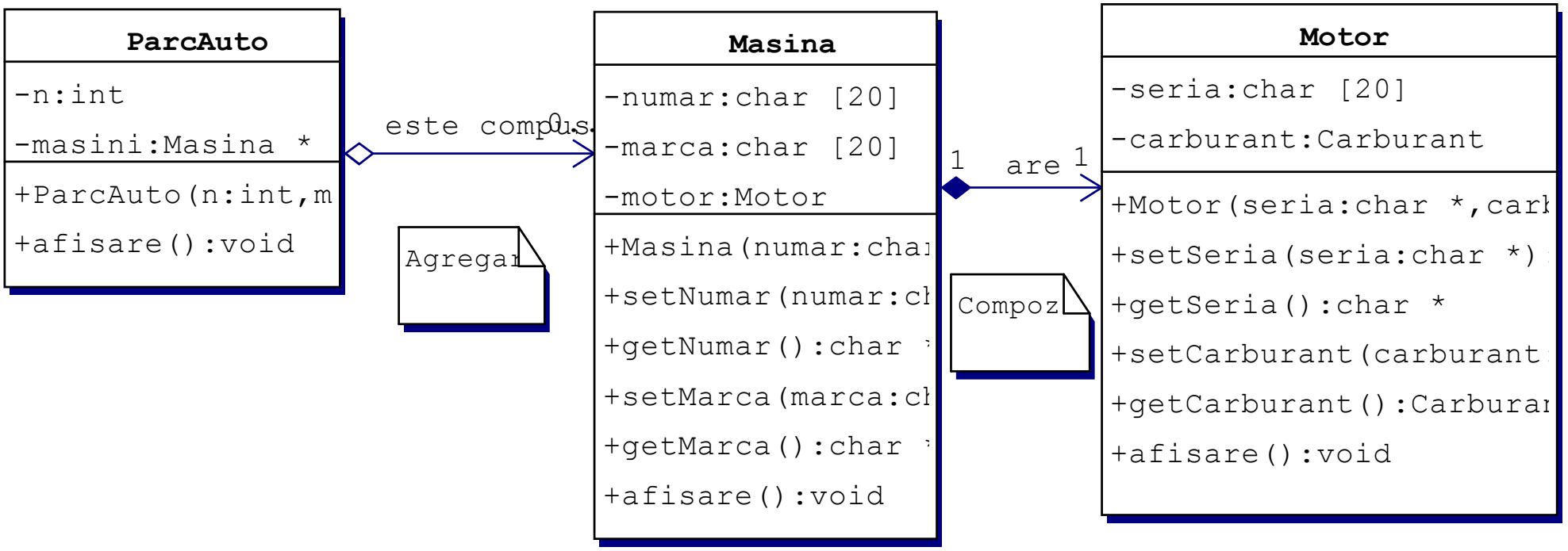
- **Agregarea** propriu-zisă



Exemplu

- Considerăm următorul enunț: *Un **parc auto** este format dintr-o mulțime de maşini. Pentru fiecare **maşină** din parc se cunosc următoarele detalii: numărul de înmatriculare, marca, seria **motor**ului şi tipul de carburant utilizat de acesta.*
- Să se modeleze utilizând diagrame UML.

Diagrama UML



Clasa Motor

```
#include <iostream>
using namespace std;

enum Carburant {benzina = 0, motorina};

class Motor {
  private:
    char seria[20];
    Carburant carburant;
  public:
    Motor(const char *seria,
          Carburant carburant);
    void setSeria(const char *seria);
    char* getSeria();
    void setCarburant(Carburant carburant);
    Carburant getCarburant();
    void afisare();
};

Motor::Motor(const char *seria,
             Carburant carburant) {
  setSeria(seria);
  setCarburant(carburant);
}
```

```
void Motor::setSeria(const char *seria) {
  strcpy(this->seria, seria);
}

char* Motor::getSeria() {
  return seria;
}

Carburant Motor::getCarburant() {
  return carburant;
}

void Motor::setCarburant(Carburant carburant) {
  this->carburant = carburant;
}

void Motor::afisare() {
  cout << "Serie Motor:" << seria <<endl;
  cout << "Tip Combustibil:" <<
((carburant==motorina)?"motorina":"benzina");
  cout << endl;
}
```

Clasa Masina

```
class Masina {
  private:
    char numar[20];
    char marca[20];
    Motor motor;
  public:
    Masina(const char *numar = "",
           const char *marca = "",
           const char *seria = "",
           Carburant carburant = benzina);
    void setNumar(const char *numar);
    char* getNumar();
    void setMarca(const char *marca);
    char* getMarca();
    void afisare();
};

Masina::Masina(const char *numar,const char *marca,
           const char *seria, Carburant carburant):
                       motor(seria, carburant) {
  setNumar(numar);
  setMarca(marca);
}
```

```
void Masina::setNumar(const char *numar) {
  strcpy(this->numar, numar);
}

char* Masina::getNumar() {
  return numar;
}

void Masina::setMarca(const char *marca) {
  strcpy(this->marca, marca);
}

char* Masina::getMarca() {
  return marca;
}

void Masina::afisare() {
  cout << "Numar:" << numar << endl;
  cout << "Marca:" << marca << endl;

  motor.afisare();
}
```

Clasa ParcAuto

```
class ParcAuto {
    int n;
    Masina *masini;
  public:
    ParcAuto(int n, Masina masini[]);
    void afisare();
};

ParcAuto::ParcAuto(int n, Masina masini[]) {
   this->n = n;
   this->masini = new Masina[n];
   for(int i = 0; i < n; i++) {
     this->masini[i] = masini[i];
   }
}
```

```
void ParcAuto::afisare() {
  for(int i = 0; i < n; i++) {
    masini[i].afisare();
  }
}

int main() {
   Masina m[] ={Masina("DJ-01-UCV","Audi", "1234",
motorina), Masina("DJ-02-UCV","Logan", "2121",
benzina)};

   ParcAuto parc(2, m);
   parc.afisare();

   return 0;
}
```