Reprezentarea UML a claselor.

Mihai Gabroveanu

Ce este UML?

- **UML** acronim pentru ***U**nified **M**odeling **L**anguage*
- Este un limbaj vizual de modelare utilizat pentru specificarea, construirea şi documentarea sistemelor de aplicaţii orientate obiect şi nu numai.

De ce UML?

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

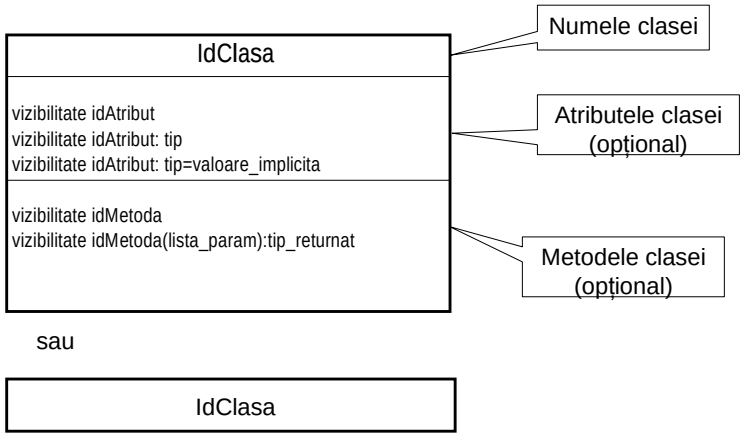
Diagrame UML

- O **diagrama** este o prezentare grafică ale unui set de elemente, cel mai adesea exprimate ca un graf de noduri (elementele) si arce (relatiile).
- Tipuri de diagrame UML
  - **Diagrame ale Cazurilor de Utilizare (Use Case)**
  - **Diagrame de clase**
  - **Diagrame de obiecte**
  - **Diagrame de secvenţă**
  - **Diagrame de stare**
  - **Diagrame de colaborare**
  - **Diagrame de activitate**

Diagramele de Clase

- Diagramele de clase sunt folosite în modelarea orientată obiect pentru a descrie structura statică a sistemului, modului în care este el structurat.
- Oferă o notaţie grafică pentru reprezentarea:
  - **claselor** - entităţi ce au caracteristici comune
  - **relațiilor** - relațiile dintre două sau mai multe clase

Reprezentarea unei clase



Specificarea Atributelor

- Sintaxa specificării atributelor din compartimentul atributelor este urmatoarea:

  vizibilitate idAtribut : tip = valoare_implicitia

  unde:
  - *vizibilitate* reprezintă protecția atributului și poate să aibă una din următoarele valori
    - **+** = public,
    - **-** = privat si
    - # = protected (optional)
  - *idAtribut* este identificatorul atributului
  - *tip* – este tipul acestuia
  - *valoare_implicita* reprezintă valoarea inițială a atributului (optională)

Specificarea Metodelor

- Sintaxa specificarii metodelor sau operatiilor din compartimentul metodelor este urmatoarea:

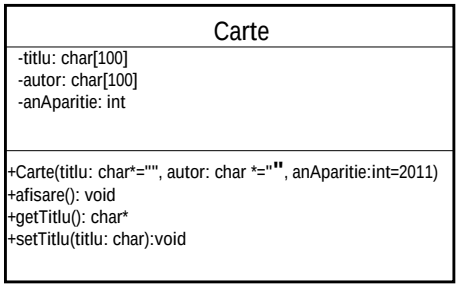
  vizibilitate idMetoda(idP1:tip1, ..., idPn:tipn) : tip_returnat

  unde:
  - *vizibilitate* reprezintă protecția metodei. Poate să aibă una din următoarele valori
    - **+** = public,
    - **-** = privat si
    - # = protected (optional)
  - *idMetoda* este identificatorul metodei
  - *idP1,..., idPn* sunt parametri metodei
  - *tip1, ..., tipn* sunt tipurile parametrilor
  - *tip_returant* reprezintă tipul valorii returnate de metodă

Relațiile

- Toate relaţiile din cadrul diagramei de clase sunt reprezentate grafic printr-o succesiune de segmente orizontale sau verticale care leagă o clasă de alta.
- *Relaţiile* pot fi:
  - asocieri
  - generalizari
  - dependenţe
  - relaţii de rafinare.
- ***Asocierea*** este o conexiune între clase, ceea ce înseamnă o legătură semantică între obiectele claselor implicate în relaţie.

Diagrame de Clase. Exemplu




```
class Carte{
  private:
   char titlu[100];
   char autor[100];
   int anAparitie;
  public:
   Carte(char *titlu="", char *autor="", int an=2011);
   void afisare();
   char* getTitlu();
   void setTitlu(char *titlu);
};
```


Reutilizarea Codului

Reutilizarea Codului

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

Agregarea

- **Agregare** = Definirea unei noi clase ce include una sau mai multe date membre care au ca tip clase deja definite.
- Ne permite construirea de clase complexe pornind de la clase mai simple

Agregare.Constructori (I)

- Dacă o clasă A conține ca date membru obiecte ale altor clase (C1, …, Cn) atunci constructorul clasei A va avea suficienti parametri pentru a inițializa datele membru ale claselor C1,…, Cn.
- La crearea unui obiect al clasei A se vor apela mai întâi constructori claselor C1, … ,Cn pentru a se initializa obiectele incluse în obiectul clasei A și apoi se vor executa intrucțiunile 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_param1){
 instructiuni
}
```

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

Tipuri de agregare

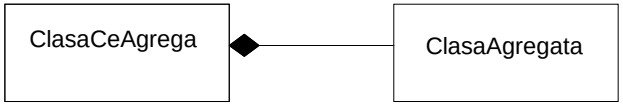
- Există două variante de agregare diferențiate de relația dintre agregat și subobiecte
  - **Compoziția –** subobiectele agregate aparțin exclusiv agregatului din care fac parte, iar durata de viață coincide cu cea a agregatului
  - **Agregarea propriu-zisă** - subobiectele au o existență independentă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

Agregare.Destructori

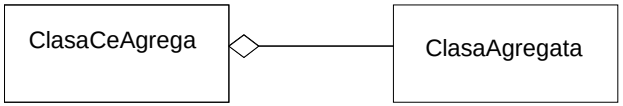
- La distrugerea unui obiect al clasei A se va executa mai destructorul clasei A apoi se vor apela destructori claselor C1, … ,Cn în ordinea inversa apelurilor constructorilor.

Agregarea. Reprezentare UML

- Compozitia



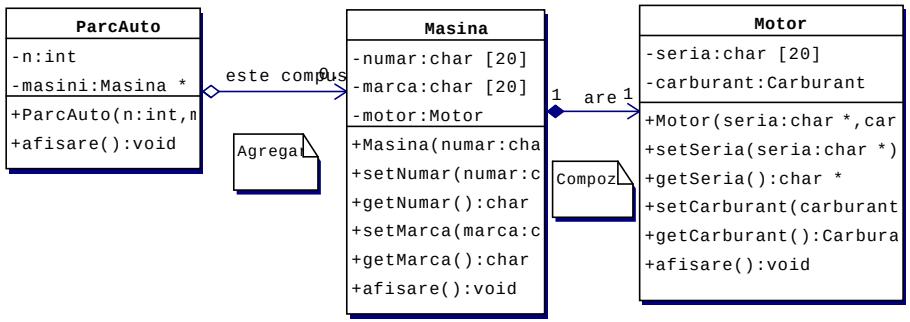
- Agregarea propriu-zisă



Exemplu

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

Diagrama UML



Clasa Motor

```
enum Carburant { benzina=0, motorina};


class Motor {

  private:
    char seria[20];
    Carburant carburant;

  public:
    Motor(char *seria, Carburant carburant);
    void setSeria(char *seria);
    char* getSeria();
    void setCarburant(Carburant carburant);
    Carburant getCarburant();
    void afisare();
};

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

```
void Motor::setSeria(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(char *numar="", char *marca="", char
*seria="", Carburant carburant=benzina);
     void setNumar(char *numar);
     char* getNumar();
     void setMarca(char *marca);
     char* getMarca();
     void afisare();
};

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

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

void Masina::setMarca(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();
   getch();
}
```