

Lecția 6

Polimorfismul

6.1 Ce este polimorfismul?

Prin intermediul magistralei USB (Universal Serial Bus) se pot conecta la un calculator diverse dispozitive periferice precum aparate foto sau camere video digitale, toate produse de diverse firme. Magistrala care conectează dispozitivul periferic la un calculator nu cunoaște tipul dispozitivului periferic ci doar faptul că acesta este un dispozitiv periferic. Ce s-ar întâmpla însă dacă magistrala ar cunoaște diferitele particularități de transfer de date ale fiecărui dispozitiv? Răspunsul este simplu: în loc de o funcție generală pentru transferul datelor între calculator și diverse periferice, aceasta ar trebui să ofere mai multe funcții de transfer specifice pentru fiecare tip de periferic în parte!!! Ca efect, atunci când ar apare un nou tip de periferic, magistrala USB ar trebui modificată astfel încât să fie posibil și transferul între calculator și noul dispozitiv!!! Evident acest lucru nu e convenabil. În cele ce urmează, vom arăta printr-un exemplu simplu cum anume putem modela într-un limbaj orientat pe obiecte o magistrală USB care să poată transfera date între un calculator și un dispozitiv periferic, indiferent de tipul concret sau de particularitățile de transfer de date ale perifericului.

După cum am precizat anterior, o magistrală USB realizează transferul de date dintre un calculator și un dispozitiv periferic. Anterior s-a precizat că această magistrală nu cunoaște exact tipul dispozitivului periferic implicat în transferul de date, ci doar faptul că acesta este un *dispozitiv periferic*. Spuneam în Secțiunea 1.1.1 că “*simplificăm sau abstractizăm obiectele reținând doar aspectele lor esențiale*”. Tot atunci am stabilit că aspectele esențiale ale unui obiect sunt relative la punctul de vedere din care este văzut obiectul. Concret, pentru magistrala USB este important ca dispozitivul implicat în transferul de date să conțină servicii pentru stocarea și furnizarea de date. Evident, pentru un utilizator al unei camere video este important ca aceasta, pe lângă posibilitatea conectării la un calculator în vederea descărcării filmelor, să și filmeze. Având în vedere aspectele esențiale din punctul de vedere al magistrlei USB, definim clasa *Device* de mai jos ale cărei instanțe furnizează servicii de stocare, respectiv furnizare de date.

```
class Device {  
  
    private String information;  
  
    public Device() {  
        information = "";  
    }  
  
    public Device(String information) {  
        this.information = information;  
    }  
  
    public void store(String information) {  
        this.information = information;  
    }  
  
    public String load() {  
        return information;  
    }  
}
```

Orice utilizator al oricărui dispozitiv periferic de genul aparat foto sau cameră video dorește la un moment dat să descarce informația conținută de dispozitiv pe calculator. Datorită acestui fapt e absolut normal ca orice dispozitiv periferic concret să fie modelat de o clasă ce extinde clasa *Device* adăugând în același timp servicii specifice respectivului dispozitiv ca în exemplul de mai jos.

```
class PhotoDevice extends Device {  
  
    public PhotoDevice(String information) {  
        super(information);  
    }  
  
    public void takePicture() {  
        System.out.println("TakePicture...");  
        //String picture = ...  
        //Se va apela this.store(picture) pentru stocarea pozei  
    }  
}
```

```
class VideoDevice extends Device {  
  
    private String producer;
```

```
public VideoDevice(String information, String producer) {
    super(information);
    this.producer = producer;
}

public void film() {
    System.out.println("Film...");
    //String film = ...
    //Se va apela this.store(film) pentru stocarea filmului
}
}
```

Pentru cealaltă parte implicată în transferul de date, și anume calculatorul, definim clasa *PC* de mai jos.

```
class PC {

    private String memory = "";
    private String registry;

    public void store(String information) {
        memory += information;
        registry = information;
    }

    public String load() {
        return registry;
    }
}
```

În fine, clasa *USB* oferă servicii pentru transferul de date bidirecțional între un calculator și un dispozitiv periferic. Codul său e prezentat mai jos.

```
class USB {

    public void transferPCToDevice(PC pc, Device device) {
        String data;
        data = pc.load();
        device.store(data);
        System.out.println("PC -> Device " + data);
    }

    public void transferDeviceToPC(PC pc, Device device) {
        String data;
        data = device.load();
    }
}
```

```

    pc.store(data);
    System.out.println("Device -> PC" + data);
}
}

```

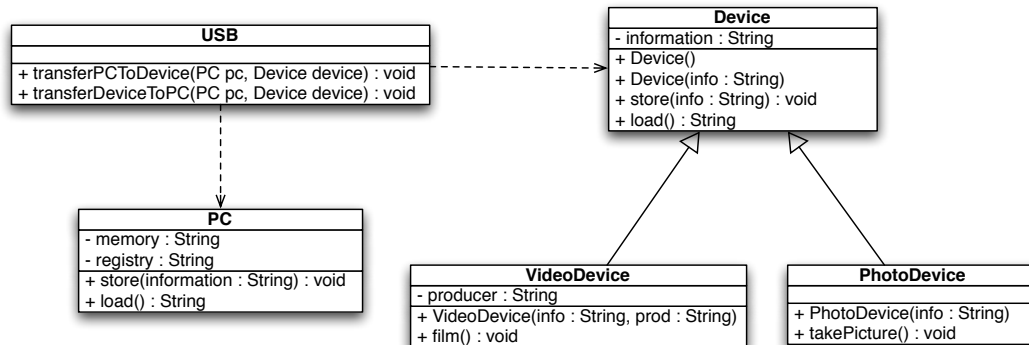


Figura 6.1: DIAGRAMA DE CLASE PENTRU EXEMPLUL DISCUTAT.



Săgețile punctate din figura de mai sus denotă relații de dependență. Relația de dependență este diferită de relația de agregare prezentată în Figura 5.1. După cum se poate observa, clasa *USB* nu conține atribute de tip *PC*, respectiv *Device* ci ea face uz de obiecte de tip *PC*, *Device* prin intermediul parametrilor *pc*, *device* aferenți metodelor *transferPCToDevice*, respectiv *transferDeviceToPC*.



Magistrala USB trebuie să poată transfera date între calculator și orice tip de dispozitiv periferic. E posibil acest lucru având în vedere că interfața magistralei oferă doar două servicii? Răspunsul este DA, deoarece *device* poate referi orice obiect instanță a clasei *Device* sau a oricărei clase ce moștenește clasa *Device*!!!

Să vedem acum un exemplu de utilizare a claselor definite până acum. În Secțiunea 5.5 am văzut că putem utiliza o instanță a unei subclase în locul unui obiect al superclasei sale. Datorită acestui fapt, clientul de mai jos e corect. Putem observa în acest client, că obiectul de tip *USB* este utilizat atât pentru a transfera date de la calculator la un aparat foto, cât și pentru a transfera date de la calculator la o cameră video. Prin urmare, bazându-ne pe moștenirea de tip, am modelat o magistrală ce poate transfera date între un calculator și orice tip concret de dispozitiv periferic.

```
class ClientUSB {  
  
    public static void main(String[] args) {  
  
        Device photo, video;  
        PC pc;  
        USB usb;  
        photo = new PhotoDevice("initialPhotoData");  
        video = new VideoDevice("initialVideoData", "company");  
        pc = new PC();  
        usb = new USB();  
  
        usb.transferPCToDevice(pc, photo);  
        usb.transferDeviceToPC(pc, video);  
    }  
}
```

Din păcate lucrurile nu se opresc aici. Să presupunem acum că în cazul camerelor video se dorește ca la descărcarea filmelor, pe lângă informația stocată de cameră, să se transmită și numele producătorului acesteia. Prin urmare, implementarea metodei *load* din clasa *Device* nu e potrivită pentru camerele video și s-ar părea că magistrala noastră USB ar trebui să trateze într-un mod particular transferul de date de la o cameră video la un calculator. Nu e deloc așa !!!

6.1.1 Suprascrierea metodelor

Definiție 8 *Atunci când într-o subclasă modificăm implementarea unei metode moștenite de la o superclasă spunem că suprascrim sau redefinim metoda respectivă.*

```
class VideoDevice extends Device {  
    ...  
    public String load() {  
        return producer + " " + super.load();  
    }  
    ...  
}
```

În exemplul de mai sus metoda *load* din clasa *VideoDevice*, redefineste metoda *load* din clasa *Device*. Mai mult, noua metodă îndeplinește și cerința de mai sus ca, în cazul camerelor video, pe lângă filmul propriu-zis să se transfere și numele producătorului echipamentului.

Atenție

Redefinirea unei metode dintr-o superclasă într-o subclasă înseamnă declararea unei metode în subclasă cu EXACT aceeași semnătură ca a

metodei din superclasă.

Important

Spunem despre o metodă suprascrisă că este o specializare (rafinare de comportament) a metodei din clasa de bază doar în cazul în care aceasta apelează metoda din superclasă. Este bine ca o metodă suprascrisă să fie o specializare. Implementarea metodei *load* din clasa *VideoDevice* este o specializare deoarece ea apelează metoda *load* moștenită de la clasa *Device* prin instrucțiunea *super.load()*.



În Secțiunea 3.4.2 am spus că este necesară o *modificare* a metodei *toString*. De fapt, modificarea despre care era vorba nu era altceva decât *suprascrierea* unei metode moștenite din clasa *Object*.

6.1.2 Legarea dinamică. Polimorfismul.

În secțiunea anterioară am redefinit metoda *load* în clasa *VideoDevice*. Acum haideti să vedem cum poate fi aceasta apelată. Un prim mod de apel al metodei este cel de mai jos, prin intermediul unei referințe de tip *VideoDevice*.

```
VideoDevice video = new VideoDevice("myVideo","XCompany");  
System.out.println(video.load()); //Se va afisa "XCompany myVideo"
```

Oare ce se va afișa însă la execuția codului de mai jos?

```
Device otherVideo = new VideoDevice("myVideo","XCompany");  
System.out.println(otherVideo.load());
```

La prima vedere am fi tentați să spunem că se va afișa doar “myVideo” deoarece referința *otherVideo* este de tip *Device* și se va executa codul metodei *load* din clasa respectivă. Lucrurile nu stau însă deloc așa iar pe ecran se va afișa “XCompany myVideo”!!! Explicația este că, de fapt, nu se apelează metoda *load* existentă în clasa *Device* ci metoda *load* existentă în clasa *VideoDevice*.

Definiție 9 *Asocierea dintre apelul la un serviciu (apel de metodă) și implementarea acestuia (implementarea metodei ce se va executa la acel apel) se numește legare.*

În limbajele de programare există două tipuri de legare:

Legare statică (early binding) : asocierea dintre un serviciu și implementarea acestuia se realizează la compilare programului. Cu alte cuvinte, se cunoaște din momentul compilării apelului metodei care este implementarea metodei ce se va executa la acel apel.

Legare dinamică (late binding) : asocierea dintre un serviciu și implementarea acestuia se realizează la NUMAI la execuția programului. Cu alte cuvinte, se

va cunoaște doar în momentul execuției apelului care implementare a unei metode se va executa la respectivul apel.

În Java, legarea apelurilor la metode nestatice se realizează în majoritatea cazurilor dinamic!!! Aceasta este explicația tipăririi mesajului “XCompany myVideo” la rularea exemplului de mai sus. La execuția apelului metodei *load*, suportul de execuție Java vede că referința *otherVideo* indică un obiect *VideoDevice* și ca urmare execută implementarea metodei *load* din clasa *VideoDevice*. Dacă referința ar fi indicat o instanță a clasei *Device* atunci s-ar fi apelat implementarea metodei *load* din clasa *Device*.

Să revenim acum la codul din clasei *USB* și mai exact la metoda *trasferDeviceToPC*. Obiectul referit de parametrul *device* este cunoscut magistralei USB doar prin intermediul interfeței sale (totalitatea metodelor publice) definite în cadrul clasei *Device*. Atunci când aceasta solicită operația *load* obiectului referit de *device*, modul în care operația va fi executată depinde de tipul concret al obiectului referit de *device*. Dacă obiectul e instanță a clasei *VideoDevice* se va executa metoda *load* din această clasă și prin urmare se va transfera și numele producătorului echipamentului. Altfel se va executa metoda *load* din clasa *Device*. Prin urmare, clasa *USB* poate fi folosită în continuare nemodificată pentru a transfera date între orice fel de echipament periferic și calculator.

Polimorfismul este ansamblul format de moștenirea de tip și legarea dinamică. El ne permite să tratăm uniform obiecte de tipuri diferite (prin moștenirea de tip) și în același timp în mod particular fiecare tip de obiect dacă e necesar (prin legarea dinamică).

6.1.3 Clase și metode abstracte

Informația stocată de o cameră video este, în general, un film care este rezultatul serviciului de filmare oferit de cameră. Putem transfera filme de pe calculator spre camera video dar scopul pentru care aceasta a fost proiectată este de a filma și nu de a păstra informații. Un dispozitiv periferic care nu poate face nimic în afară de transferul bidirecțional al datelor nu este de nici un folos (chiar și stilourile de memorie, în afară de transferul bidirecțional al datelor, mai oferă și serviciul de scriere protejată). Orice dispozitiv trebuie să fie *ceva*, în cazul nostru o cameră video sau foto.

Clasa *Device* a fost creată de fapt pentru a stabili o *interfață comună* pentru toate subclasele sale. Întrucât un obiect al clasei *Device* nu este de fapt de nici un folos, trebuie ca operația de instanțiere a acestei clase să nu fie posibilă.

Definiție 10 *O clasă abstractă este o clasă care nu poate fi instanțiată.*

Declarăm o clasă ca fiind abstractă dacă prefixăm cuvântul cheie *class* de cuvântul cheie *abstract*, ca mai jos:

```
abstract class Device {  
    ...  
}
```



În exemplul de mai sus compilatorul nu ar fi semnalat o problemă dacă clasa *Device* nu ar fi fost declarată abstractă. Faptul că ea a devenit abstractă împiedică instanțierea ei accidentală și mărește gradul de înțelegere al codului; de fiecare dată când vedem o clasă abstractă știm că ea este o interfață comună pentru toate subclasele sale.

Să considerăm un alt exemplu. Cercul, pătratul și romb sunt obiecte figuri geometrice și fiecare din aceste figuri știe să-și calculeze aria. Prin urmare clasele lor ar putea fi derivate dintr-o clasă abstractă *FiguraGeometrica*. Mai mult, întrucât fiecare figură geometrică are o arie, este normal ca *FiguraGeometrica* să conțină metoda *public float arie()*. Se pune însă problema ce implementare va avea această metodă în clasa de bază deoarece fiecare figură geometrică concretă are un mod propriu de calcul al ariei? O primă variantă este prezentată mai jos.

```
abstract class FiguraGeometrica {  
  
    public float arie() {  
        return 0;  
    }  
    ...  
}
```

Din păcate nici o figură geometrică nu poate avea aria 0, deci metoda *arie* de mai sus nu va fi specializată de nici o subclasă a clasei *FiguraGeometrica*. Mai exact, fiecare subclasă a clasei *FiguraGeometrica* va reimplementa total metoda *arie*. Cu alte cuvinte, această implementare a metodei *arie* este inutilă și nu folosește nimănui. În plus, este posibil ca în interiorul unei subclase să uităm să suprascriem metoda *arie* și atunci în mod cert aria nu va fi calculată corect.

Varianta corectă în acest caz este ca metoda *arie* să fie abstractă, ea neavând nici o implementare în clasa de bază *FiguraGeometrica*.

```
abstract class FiguraGeometrica {  
  
    public abstract float arie();  
  
    ...  
}
```


Atenție

Spuneam în Secțiunea 3.1 că în anumite condiții corpul unei metode poate lipsi. O metodă abstractă nu are niciodată o implementare, deci corpul acesteia întotdeauna lipsește. Datorită acestui fapt compilatorul ar fi semnalat o eroare dacă am fi încercat să-i dăm o implementare metodei *arie*.

Atenție

O clasă poate fi abstractă chiar dacă aceasta nu conține nici o metodă abstractă.

Atenție

Dacă o clasă conține cel puțin o metodă abstractă, atunci respectiva clasă va trebui și ea declarată ca fiind abstractă, în caz contrar semnalându-se o eroare la compilare.

Atenție

O subclasă a unei clase abstracte trebuie să implementeze toate metodele abstracte ale supercalsei sale, în caz contrar fiind necesar ca și subclasa să fie abstractă. Acest fapt este normal fiindcă nu pot fi instanțiate clase care conțin servicii neimplementate!



Într-o diagramă de clase UML, clasele și metodele abstracte se evidențiază ca în Figura 6.2. Evident, când desenăm de mână o diagramă, este cam greu să scriem italic. În astfel de cazuri, se practică marcarea explicită a clasei/metodei abstracte folosind notația {abstract}.

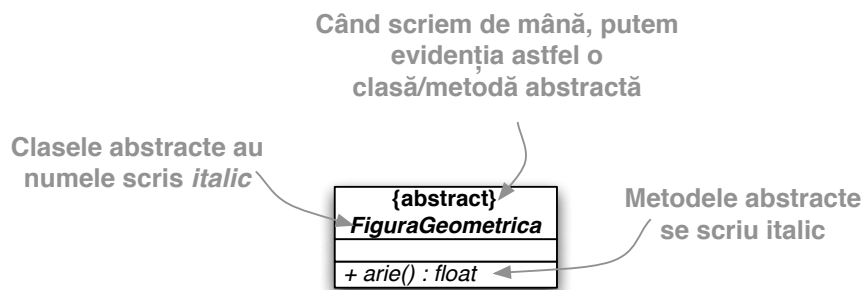


Figura 6.2: REPREZENTAREA UNEI CLASE/METODE ABSTRACTE ÎN UML.

6.2 Principiul de Proiectare Închis-Deschis

Atunci când vrem să realizăm un sistem software, nu putem trece direct la faza de scriere a codului sursă. Procesul de realizare al unui program este compus din mai multe faze, printre care și așa numita fază de proiectare. Proiectarea unui sistem software se face

pe baza unor principii iar *Principiul Închis-Deschis*, sau *The Open-Closed Principle*, este unul dintre cele mai importante.

Definiție 11 *Principiul OCP ne sugerează să proiectăm un program astfel încât entitățile software (clase, module, funcții, etc) să fie deschise pentru extensii dar închise pentru modificări [3].*

Modulele care se conformează cu acest principiu au două caracteristici principale:

Deschise pentru extensii - în acest caz ce se poate extinde nu este altceva decât comportamentul modulelor.

Închise pentru modificări - extinderea comportamentului modulelor nu duce la modificări asupra codului respectivelor modulelor.

Clasa *USB* a fost proiectată ținând cont de acest principiu. Comportamentul său poate fi extins prin adăugarea de noi dispozitive (de exemplu introducând o subclasă *Imprimanta* a clasei *Device*) în sensul că ea poate fi utilizată atunci pentru a transfera informații între un calculator și un nou tip de periferic (adică o imprimantă). În același timp, clasa *USB* va rămâne nemodificată la nivel de cod sursă. Haideți să vedem ce s-ar fi întâmplat dacă nu ar fi fost respectat principiul OCP. Avem mai jos o variantă a clasei *USB* care nu respectă acest principiu (din economie de spațiu am reprodus doar metodele de transfer de la calculator la periferic și nu și invers).

```
class USB {  
  
    public void transferPCToDevice(PC pc, PhotoDevice device) {  
        String data;  
        data = pc.load();  
        device.store(data);  
        System.out.println("PC -> Device " + data);  
    }  
  
    public void transferPCToDevice(PC pc, VideoDevice device) {  
        String data;  
        data = device.load();  
        pc.store(data);  
        System.out.println("PC -> Device " + data);  
    }  
}
```

Problema acestei variante a clasei *USB* constă în faptul că ea cunoaște diferitele tipuri de dispozitive periferice existente iar în momentul introducerii de noi subclase a clasei *Device*, de exemplu *NewDevice*, clasei *USB* va trebui să i se adauge metoda:

```
public void transferPCToDevice(PC pc, NewDevice device) {  
    ...  
    System.out.println("PC -> Device " + data);  
}
```



Și totuși, care este marea problemă? Doar nu este așa de greu să adăugăm o metodă cu același conținut ca și restul, având doar un parametru schimbat! Ei bine, problema este mult mai gravă decât pare la prima vedere. Gândiți-vă că de fiecare dată când achiziționați un nou dispozitiv ce trebuie să comunice prin intermediul magistralei USB cu propriul calculator, trebuie să mergeți întâi la un service de calculatoare pentru a modifica magistrala USB!!! Exact așa stau lucrurile și aici: la fiecare creare a unei clase derivate din clasa *Device* clasa *USB* va trebui modificată și recompilată!!!

Se poate observa că implementările celor două metode *transferPCToDevice* sunt identice. Spunem că între cele două metode există o *duplicare de cod*. Duplicarea de cod existentă în cadrul unui sistem este o sursă serioasă de probleme. Presupunem că la un moment dat în cadrul transferului se dorește să se afișeze pe ecran în loc de "PC -> Device" un alt șir de caractere. Este evident că va trebui modificat codul în mai multe locuri, nu doar în unul singur.

Să vedem acum o altă variantă a clasei *USB* în care implementarea metodei *transferPCToDevice* arată în felul următor:

```
class USB {  
  
    public void transferPCToDevice(PC pc, Device device) {  
        String data;  
        data = pc.load();  
        device.store(data);  
        if (device instanceof PhotoDevice)  
            System.out.println("PC -> PhotoDevice " + data);  
        else if (device instanceof VideoDevice)  
            System.out.println("PC -> VideoDevice " + data);  
    }  
}
```

Evident că această implementare nu respectă OCP, la adăugarea unui nou tip de dispozitiv fiind necesară modificarea metodei.

Sfat

Încercați să scrieți programe astfel încât acestea să nu conțină duplicare de cod și nici secvențe if-then-else care verifică tipul concret al unui obiect. Duplicarea de cod și secvențele lungi if-then-else de acest fel sunt semne că undeva nu utilizați polimorfismul deși ar trebui.

6.3 Supraîncărcarea versus suprascrierea metodelor

În exemplul de mai jos, în clasa *D*, în loc să suprascriem metoda *oMetoda* am supraîncărcat-o.

Important

În literatură *supraîncărcarea* metodelor este cunoscută sub numele de *overloading* iar *suprascrierea* sub numele de *overriding*.

```
class B {  
    public void oMetoda(Object o) {  
        System.out.println("BBBB.oMetoda");  
    }  
    ...  
}  
  
class D extends B{  
    public void oMetoda(B b) {  
        System.out.println("DDDD.oMetoda");  
    }  
    ...  
}
```

Din punctul de vedere al compilatorului acest fapt nu reprezintă o problemă dar comportamentul metodei supraîncărcate s-ar putea să nu fie acela dorit de noi.

```
B b = new B();  
B d = new D();  
d.oMetoda(b); //Se va afișa BBBB.oMetoda
```

Poate ce am fi dorit noi să se afișeze este *DDDD.oMetoda* dar acest fapt nu se întâmplă deoarece clasa *B* nu conține nici o metodă cu semnătura *oMetoda(B)*, în acest caz apelându-se metoda *oMetoda(Object)* din clasa de bază.



Pentru ca o metodă dintr-o clasă derivată să se apeleze polimorfic **OBLIGATORIU** trebuie ca aceasta să *suprascrie* și nu să *supraîncarce* o metodă din clasa de bază.

6.4 Constructorii si polimorfismul

Presupunând că definim clasele de mai jos, oare ce se va afișa (și ce am dori să se afișeze) în urma execuției următoarei secvențe?

```
SubClasa ss = new SubClasa();
System.out.println("Valoarea este " + ss.getValoare());
```

```
class SuperClasa {

    protected int valoareSuperClasa;

    public SuperClasa() {
        valoareSuperClasa = valoareImplicita();
    }

    public int valoareImplicita() {
        return 10;
    }

    public int getValoare() {
        return valoareSuperClasa;
    }

}

class SubClasa extends SuperClasa {

    private int valoareSubClasa;

    public SubClasa() {
        valoareSubClasa = 20;
    }

    public int valoareImplicita() {
        return valoareSubClasa;
    }

}
```

Poate că noi ne-am dori ca efectul să fie tipărirea șirului de caractere “Valoarea este 20”, numai că în loc de acesta se va tipări “Valoarea este 0”!!!



Dacă în constructorul unei superclase se apelează o metodă care este suprascrisă în subclasă, la crearea unui obiect al subclasei există riscul ca metoda respectivă să refere câmpuri încă neinițializate în momentul apelului.

Pentru exemplul de mai sus să urmărim ce se întâmplă la instanțierea unui obiect al clasei *SubClasa*.

1. se inițializează câmpurile cu valorile implicite corespunzătoare tipurilor lor, deci în

cazul nostru cu 0.

2. se apelează constructorul *SubClasa* care apelează constructorul *SuperClasa*.
3. constructorul *SuperClasa* apelează metoda suprascrisă *valoareImplicita*; datorită faptului că metoda este suprascrisă, implementarea apelată este cea din *SubClasa* și nu cea din *SuperClasa* iar fiindcă atribuirea *valoareSubClasa = 20* încă nu s-a realizat metoda va întoarce valoarea 0!!!
4. se revine în constructorul *SubClasa* și abia acum se execută atribuirea din acest constructor.

6.5 Tablourile și polimorfismul

În exemplul de mai jos am creat un tablou ale cărui elemente sunt referințe de tipul *Device*. Faptul că am scris *new Device[10]* NU înseamnă că am creat 10 obiecte de tip *Device* ci doar că am instanțiat un tablou de dimensiune 10, fiecare intrare a tabloului fiind o referință de tip *Device*.

```
Device[] devices = new Device[10];
```

În acest moment, intrările tabloului nu referă obiecte și se pune problema inițializării lor. Ce fel de elemente putem adăuga în tablou, având în vedere că *Device* este o clasă abstractă (presupunând că am declarat-o așa)?

```
devices[0] = new VideoDevice("video","company");
```

Răspunsul la problemă este simplu. Ținând cont de moștenirea de tip, și în acest caz putem utiliza o instanță a unei subclase în locul unei instanțe a superclasei sale și deci, codul de mai sus este absolut corect.

6.6 Exercițiu rezolvat

Joc de strategie

Într-un joc de strategie simplificat există mai multe feluri de unități de luptă. Indiferent de felul său concret, pe un obiect unitate de luptă se poate apela din exterior:

- o metodă denumită *rănire* ce primește ca parametru o valoare întreagă
- o metodă denumită *lovește* care primește ca parametru o referință la o unitate de luptă de orice fel
- o metodă denumită *esteVie* care întoarce un boolean prin care se poate testa dacă unitatea mai este sau nu în viață

Implementarea acestor metodelor depinde de felul concret al obiectului unitate de luptă după cum se detaliază în cele ce urmează. Felurile concrete de unități de luptă sunt:

- Arcaș - când un arcaș este rănit (adică și mai exact, se apelează pe el metoda `rănire`) atunci, dacă e viu viața sa se decrementează cu valoarea dată ca argument metodei. Dacă e mort nu se întâmplă nimic (nu mai poate muri odată și deci nu mai are rost decrementarea). Viața inițială a unui arcaș este de 100 și el devine mort când viața sa este mai mică sau egală cu 0. Când un arcaș lovește o unitate de luptă și el nu e mort, efectul constă în rănirea unității date ca parametru metodei lovește cu valoarea 10. Dacă e mort rănirea se face cu valoarea 0.
- Călăreț - când un călăreț este rănit atunci, dacă e viu viața sa se decrementează cu valoarea dată ca argument metodei `rănire`. Dacă e mort nu se întâmplă nimic. Viața inițială a unui călăreț este de 200 și el devine mort când viața sa este mai mică sau egală cu 0. Când un călăreț lovește o unitate de luptă și el nu e mort, efectul constă în rănirea unității date ca parametru metodei lovește cu valoarea 15. Dacă e mort rănirea se face cu valoarea 0.
- Pluton - care reprezintă o înșiruire de unități de luptă de orice fel (arcași, călăreți, alte plutoane) într-un număr nelimitat și în orice combinație. Când un pluton este rănit efectul constă în rănirea unei unități vii din pluton (prima găsită), iar dacă nu există nici o unitate vie în pluton atunci nu se întâmplă nimic. Când un pluton lovește o unitate de luptă efectul constă în lovirea unități date ca parametru metodei de către fiecare unitate a plutonului (nu are importanță dacă membrul e viu sau mort pentru că oricum, dacă e mort, rănirea va fi în final 0). Un pluton se consideră mort dacă toate unitățile din el sunt moarte, iar dacă nu conține nici un membru se consideră implicit viu. Clasa mai definește o metodă denumită `adaugă` care primește ca parametru o referință la o unitate de luptă de orice fel și o adaugă în plutonul corespunzător. Dacă unitatea ce se dorește adăugată e moartă, metoda de adăugare întoarce false și adăugarea la pluton nu se face. Dacă plutonul la care se dorește adăugarea e mort, metoda întoarce false iar adăugarea la pluton nu se face. Altfel, metoda de adăugare întoarce true.

Se cere:

1. Diagrama de clase detaliată cuprinzând toate clasele descrise și orice alte clase considerate necesare.
2. Implementarea tuturor claselor descrise, inclusiv a altora considerate necesare.
3. Considerând că la moartea unui călăreț moare și calul, introduceți și implementați corespunzător în una din clase o metodă care să întoarcă numărul de cai ce au murit de la începutul rulării jocului/programului (aceste elemente nu trebuie surprinse în diagrama de clase).
4. Exemplificați, într-o metodă `main`, lupta dintre un pluton format din patru arcași și un alt pluton format dintr-un călăreț și un pluton format din alți doi arcași. După aceea tipăriți pe ecran numărul de cai decedați.

Rezolvare

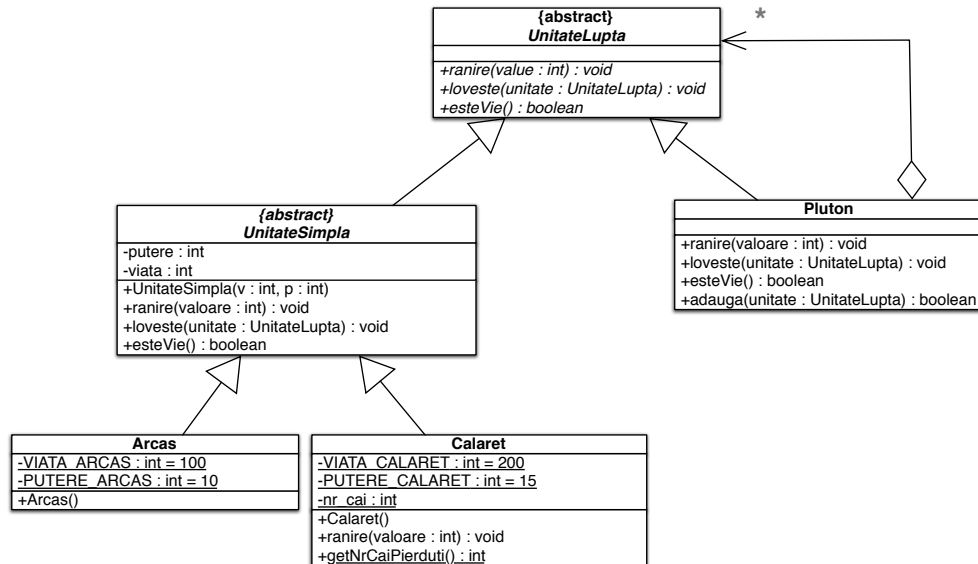


Figura 6.3: DIAGRAMA DE CLASE.

Din specificațiile problemei putem observa că plutonul va fi compus din mai multe unități de luptă de diverse feluri. Astfel, va trebui să putem avea plutoane formate din călăreți, va trebui să putem avea plutoane formate din arcași, va trebui să putem avea plutoane formate din arcași și călăreți. Mai mult, din specificații reiese că vom putea avea plutoane formate din alte (sub) plutoane, călăreți și alte plutoane și așa mai departe.

Pentru a putea rezolva simplu această explozie de combinații posibile ne vom baza pe polimorfism, adică pe faptul că putem trata uniform diversele feluri de unități de luptă. Mai exact, dorim să putem declara variabile referință ce să poată referi uniform atât călăreți cât și arcași și plutoane. Prin urmare, toate aceste obiecte vor trebui să aibă un supertip comun pentru a putea declara astfel de variabile referință. În consecință vom defini clasa *UnitateLupta* ce va fi superclasă pentru toate clasele ce modelează obiecte care reprezintă entități luptătoare.

O problemă care s-ar putea ridica aici e următoare. Dintr-o lecție anterioară știm că *Object* este superclasă pentru toate clasele din Java. De ce nu folosim variabile referință declarate de tip *Object* pentru a referi uniform orice fel de unitate de luptă?

Motivul e simplu: din specificațiile problemei reiese că, pe lângă dorința de referire uniformă, dorim și să putem apela un set de metode (adică *ranire*, *esteVie* și *loveste*) în manieră uniformă, fără a ține cont de felul concret al unei unități luptătoare. În consecință, supertipul nostru comun trebuie să conțină ori să moștenească declarațiile acestor metode. Cum *Object* nu are cum conține aceste declarații, nu ne putem baza pe referințe de acest tip (de câte ori am dori să facem un apel la metodele de mai sus, fără a ști cu ce fel de unitate de luptă avem de interacționat, ar trebui să determinăm felul concret al unității folosind operatorul *instanceof* și va trebui să facem un *cast* corespunzător).

Acesta este și motivul pentru care clasa *UnitateLupta* conține declarațiile metodelor *ranire*, *esteVie* și *loveste*. Deoarece la acest nivel de abstractizare nu există funcționalitate similară între călăreți, arcași și plutoane, toate aceste metode vor fi declarate abstracte. În același timp, la acest nivel de abstractizare nu avem nici date comune pentru absolut toate felurile de unități de luptă și, prin urmare, nu avem ce câmpuri să includem în această clasă.

```
abstract class UnitateLupta {  
  
    public abstract void ranire(int value);  
  
    public abstract void loveste(UnitateLupta unitate);  
  
    public abstract boolean esteVie();  
  
}
```

Arcașii și călăreții au o mare parte din caracteristici și funcționalități similare. De exemplu, ambele feluri de unități de luptă au o valoare curentă pentru viața lor, au o valoare pentru puterea lor, se comportă la fel când sunt rănite și așa mai departe. Pentru a nu duplica o mare parte din implementarea călăreților și arcașilor, ne vom baza pe moștenirea de clasă și vom da factor comun aceste elemente incluzându-le în clasa abstractă *UnitateSimpla*, urmând ca aceasta să fie extinsă de clasa arcașilor și de cea a călăreților.

De ce este clasa *UnitateSimpla* abstractă ? Deoarece folosim clasa doar pentru a factoriza cod și nu are sens să instanțiem vreodată această clasă. Nu avem obiecte care sunt doar unități simple: avem fie arcași, fie călăreți, fie plutoane.

```
abstract class UnitateSimpla extends UnitateLupta {  
  
    private int viata, putere;
```

```
public UnitateSimpla(int viata, int putere) {
    this.viata = viata;
    this.putere = putere;
}

public void ranire(int valoare) {
    if(esteVie()) {
        viata = viata - valoare;
    }
}

public void loveste(UnitateLupta unitate) {
    if(esteVie()) {
        unitate.ranire(putere);
    }
}

public boolean esteVie() {
    if(viata > 0) {
        return true;
    } else {
        return false;
    }
}
}
```

Am ajuns în sfârșit să implementăm arcașii. Toate datele și funcționalitățile arcașilor sunt moștenite de la clasa *UnitateSimpla*, singura sarcină care cade pe umerii clasei *Arcas* fiind inițializarea corespunzătoare a vieții și puterii unui arcaș. Cum superclasa cere ca acest lucru să fie realizat prin constructorul ei, clasa *Arcas* va trebui să realizeze inițializarea prin apelarea acestui constructor dintr-un constructor propriu.

```
class Arcas extends UnitateSimpla {

    private static final int VIATA_ARCAS = 100;
    private static final int PUTERE_ARCAS = 10;

    public Arcas() {
        super(VIATA_ARCAS, PUTERE_ARCAS);
    }

}
```

Din punctul de vedere al inițializării vieții și puterii unui călăreț, clasa călăreților are aceleași sarcini ca și clasa arcașilor.

În plus însă, problema cere să avem o metodă prin care să determinăm oricând câți cai au decedat de la începutul rulării jocului. Momentul decesului unui cal poate fi determinat în momentul rănirii unui călăreț. În consecință, este clar că această clasă reprezintă locul ideal de implementare a funcționalității cerute.

Astfel, clasa călăreților redefinesc metoda *rănire*, determinând starea călărețului înainte și după rănirea propriu-zisă: dacă înainte de rănire călărețul e viu și după rănire nu mai e viu, înseamnă că tocmai a decedat atât el cât și calul său. În acest caz vom incrementa o variabilă statică (definită în această clasă). Ea trebuie să fie statică deoarece trebuie să determinăm numărul *total* al cailor decedați, indiferent de ce cal și implicit obiect călăreț dispăre.

În cele din urmă, pentru a pune la dispoziție metoda cerută, în clasa *Calaret* definim metoda statică *getNrCaiPierduti* care întoarce valoarea variabilei statice menționate anterior. Metoda va fi statică pentru că ea nu caracterizează un călăreț anume: ea caracterizează clasa călăreților spunând câte din instanțele sale și-au pierdut caii (mai exact câte din instanțele sale au decedat).

```
class Calaret extends UnitateSimpla {

    private static final int VIATA_CALARET = 200;
    private static final int PUTERE_CALARET = 15;

    private static int nr_cai = 0;

    public static int getNrCaiPierduti() {
        return nr_cai;
    }

    public Calaret() {
        super(VIATA_CALARET, PUTERE_CALARET);
    }

    public void ranire(int valoare) {
        boolean inainte_de_ranire = this.esteVie();
        super.ranire(valoare);
        boolean dupa_ranire = this.esteVie();
        if((inainte_de_ranire == true) && (dupa_ranire == false)) {
            nr_cai++;
        }
    }
}
```

Cea mai interesantă parte din clasa *Pluton* constă în memorarea membrilor plutonului. Pentru acest lucru definim un tablou de *UnitateLupta* pentru ca elementele sale să

poată referi conform cerințelor orice fel concret de unitate de luptă. Inițial tabloul are alocate 10 poziții.

Pe de altă parte, cerințele precizează că un pluton poate avea un număr nelimitat de membri. Prin urmare, în cadrul metodei *adauga*, vom determina dacă mai există poziții neocupate în tablou. În cazul în care nu mai există poziții libere (adică numărul curent de membri este egal cu dimensiunea alocată tabloului), vom crea un tablou de dimensiune mai mare, vom copia toate referințele din vechiul tablou în noul tablou, urmând ca noul tablou să fie folosit în locul celui vechi.

Evident, clasa pluton va trebui să implementeze comportamentul său corespunzător metodelor declarate în clasa *UnitateLupta*. Această implementare este extrem de simplă odată ce rolul supertipului comun a fost înțeles și, prin urmare, nu o vom mai discuta pe larg.

```
class Pluton extends UnitateLupta {  
  
    private UnitateLupta[] membri = new UnitateLupta[10];  
  
    private int nr_membri = 0;  
  
    public void ranire(int valoare) {  
        for(int i = 0; i < nr_membri; i++) {  
            if(membri[i].esteVie()) {  
                membri[i].ranire(valoare);  
                break;  
            }  
        }  
    }  
  
    public void loveste(UnitateLupta unitate) {  
        for(int i = 0; i < nr_membri; i++) {  
            membri[i].loveste(unitate);  
        }  
    }  
  
    public boolean esteVie() {  
        if(nr_membri == 0) {  
            return true;  
        }  
        for(int i = 0; i < nr_membri; i++) {  
            if(membri[i].esteVie()) {  
                return true;  
            }  
        }  
    }  
}
```

```

        return false;
    }

    public boolean adauga(UnitateLupta unitate) {
        if(!unitate.esteVie() || !this.esteVie()) {
            return false;
        }
        if(nr_membri == membri.length) {
            UnitateLupta[] tmp = new UnitateLupta[membri.length + 10];
            for(int i = 0; i < membri.length; i++) {
                tmp[i] = membri[i];
            }
            membri = tmp;
        }
        membri[nr_membri] = unitate;
        nr_membri++;
        return true;
    }
}

```

Clasa *Main* implementează metoda *main* care realizează operațiile cerute în ultima parte a cerințelor problemei. Astfel, metoda construiește un pluton ce conține patru arcași (referit de *pluton1*), apoi un pluton ce conține doi arcași (referit de *pluton3*) și în fine, un pluton (*pluton2*) ce conține un călăreț și plutonul anterior (*pluton3*).

Pentru exemplificarea luptei, fiecare pluton lovește succesiv pe celălalt pluton, până în momentul în care unul din plutoane nu mai e viu. Pentru a fi mai interesantă simularea, se va determina aleator care pluton lovește prima dată. În final se tipărește pe ecran starea plutoanelor și învingătorul, și apoi se tipărește numărul cailor decedați.

```

public class Main {

    public static void main(String argv[]) {

        Pluton pluton1, pluton2, pluton3;

        pluton1 = new Pluton();
        pluton1.adauga(new Arcas());
        pluton1.adauga(new Arcas());
        pluton1.adauga(new Arcas());
        pluton1.adauga(new Arcas());
    }
}

```

```

    pluton3 = new Pluton();
    pluton3.adauga(new Arcas());
    pluton3.adauga(new Arcas());
    pluton2 = new Pluton();
    pluton2.adauga(new Calaret());
    pluton2.adauga(pluton3);

    //Pentru a determina aleator care pluton loveste primul
    //folosim metoda statica random din clasa Math ce intoarce
    //un numar aleator intre 0 si 1.
    boolean loveste_primul = (Math.random() > 0.5);
    while(pluton1.esteVie() && pluton2.esteVie()) {
        if(loveste_primul) {
            System.out.println("Loveste Pluton1");
            pluton1.loveste(pluton2);
        } else {
            System.out.println("Loveste Pluton2");
            pluton2.loveste(pluton1);
        }
        loveste_primul = !loveste_primul;
    }
    System.out.println("Pluton1 este viu:" + pluton1.esteVie());
    System.out.println("Pluton2 este viu:" + pluton2.esteVie());
    System.out.println("A castigat:" + (pluton1.esteVie() ? "pluton1" :
        pluton2.esteVie() ? "pluton2" : "nimeni"));

    System.out.println("Numar cai decedati:"
        + Calaret.getNrCaiPierduti());

}

}

```

6.7 Exerciții

1. Modificați ultima implementare a metodei *transferPCToDevice* din Secțiunea 6.2 astfel încât aceasta să respecte principiul OCP iar apelul acesteia să producă același efect. Puteți modifica oricare din clasele *Device*, *PhotoDevice* și *VideoDevice*. Care sunt beneficiile modificării?
2. Modificați corespunzător clasa *B* din Secțiunea 6.3 astfel încât apelul metodei *oMetoda* din exemplul dat să se facă polimorific.
3. La ghișeu de încasări a taxelor locale se prezintă un contribuabil. Operatorul de la ghișeu caută contribuabilul (după nume sau CNP), îi spune cât are de plătit pentru anul curent în total pentru toate proprietățile după care poate încasa bani (o sumă

totală sau parțială). Fiecare contribuabil poate deține mai multe proprietăți: clădiri și/sau terenuri. Fiecare proprietate e situată la o adresă (o adresă are stradă și număr). Suma datorată în fiecare an pentru fiecare tip de proprietate se calculează în felul următor:

- pentru clădire: $500 * \text{suprafața clădirii}(m^2)$
- pentru teren: $350 * \text{suprafața terenului}(m^2) / \text{rangul localității în care se află terenul}$. Rangul unei localități poate fi 1, 2, 3 sau 4.

Contribuabilul, indiferent dacă plătește sau nu, poate solicita un fluturaș cu toate proprietățile pe care le deține alături de suma pe care trebuie să o plătească în anul curent pentru o proprietate (fluturașul arată la fel indiferent dacă pentru anul în curs contribuabilul a achitat ceva sau nu). Fluturașul are următoarea structură:

```
Contribuabil: Ion Popescu

Proprietati
  Cladire: Strada V Parvan Nr. 2
          Suprafata: 20
          Cost: 10000

  Teren:   Strada V. Parvan Nr. 2
          Suprafata: 10, Rang: 1
          Cost: 3500

  Cladire: Strada Lugoj Nr. 4
          Suprafata: 25
          Cost: 12500

Suma totala: 26000
```

Se cere:

- să se construiască diagrama UML pentru clasele necesare la realizarea operațiilor descrise anterior.
 - să se implementeze o parte din clasele identificate mai sus astfel încât să poată fi executată operația: operatorul, după ce a găsit contribuabilul Ion Popescu, afișează fluturașul corespunzător acestui contribuabil. În metoda *main* se instanțiază clasa ce modelează conceptul de contribuabil, se setează proprietățile aferente acestuia după care se face afișarea lor.
4. Se cere să se modeleze o garnitură de tren. Se va defini în acest scop o clasă *Tren*. Un obiect de tip *Tren* conține mai multe referințe spre obiecte de tip *Vagon* care sunt păstrate într-un tablou. Un vagon poate fi de 3 tipuri: *CalatoriA*, *CalatoriB* și *Marfa*. Despre garnitura de tren și vagoane mai cunoaștem următoarele:
- un tren poate conține maxim 15 vagoane, indiferent de tipul vagoanelor. Vagoanele sunt atașate trenului la crearea lui.

- un vagon de tip CalatoriA
 - are capacitatea de 40 pasageri și 300 colete poștale.
 - furnizează două servicii pentru deschiderea, respectiv închiderea automată a ușilor.
- un vagon de tip CalatoriB
 - are capacitatea de 50 pasageri și 400 colete poștale.
 - furnizează două servicii pentru deschiderea, respectiv închiderea automată a ușilor.
 - fiind un vagon mai nou, furnizează un serviciu automat pentru blocarea geamurilor.
- un vagon de tip Marfa
 - are capacitatea de 400 colete poștale.
 - nu furnizează servicii pentru deschiderea, respectiv închiderea automată a ușilor, aceste operații executându-se manual. Atenție: se interzice existența metodelor pentru deschiderea, respectiv închiderea automată a ușilor în clasa ce modelează acest tip de vagon.

Se cere:

- să se construiască diagrama UML pentru clasele identificate pe baza descrierii anterioare.
 - să se implementeze clasa care modelează conceptul de vagon împreună cu eventualele sale clase derivate. Se menționează că apelurile serviciilor pentru deschiderea, respectiv închiderea ușilor, blocarea geamurilor vor afișa pe ecran un mesaj corespunzător, spre exemplu, apelul serviciului pentru blocarea geamurilor ar putea tipări “Geamurile s-au blocat”. Implementarea se va face astfel încât valorile asociate numărului de pasageri, colete să nu fie stocate în diverse câmpuri ale vagoanelor.
 - să se implementeze o metodă pentru determinarea egalității dintre două trenuri, presupunându-se că două trenuri sunt egale dacă pot transporta același număr de colete, precum și o metodă pentru afișarea tipurilor de vagoane existente într-un tren (ATENȚIE: Tipul unui vagon trebuie determinat prin apeluri polimorfice).
 - să se scrie o metodă *main* pentru exemplificarea apelurilor celor 2 metode definite la punctul precedent.
5. Considerăm o colecție de greutăți în cadrul căreia elementele sunt reținute sub forma unui tablou. Fiecare greutate are o anumită capacitate care poate fi obținută apelând metoda *public int capacitate()* pe care o are fiecare greutate. Greutățile pot fi de următoarele tipuri:
- simple, a căror capacitate este setată prin constructor.

- duble, adică formate din 2 greutateți ce sunt stocate în două câmpuri de tip *Greutate*. Aceste greutateți sunt setate prin constructor dar pot să fie modificate pe parcursul existenței obiectelor de acest tip prin intermediul a două metode accesore (*public void setGreutate1(Greutate g)*, *public void setGreutate2(Greutate g)*). Capacitatea acestui tip de greutate e egală cu suma capacităților celor două greutateți conținute. Capacitatea acestui tip de greutate nu va fi reținută într-un atribut, ci va fi calculată de fiecare dată când unui obiect de acest tip i se va solicita serviciul *capacitate()*.
- multiple, care reprezintă o înșiruire de greutateți simple, duble, și/sau eventual alte greutateți multiple. Cu alte cuvinte, o greutate multiplă reprezintă o înșiruire de greutateți. Capacitatea unei greutateți de acest tip este egală cu suma capacităților greutateților componente. Componentele acestui tip de greutate se setează prin constructorul clasei, dar se poate alege și o altă modalitate de inserare a componentelor. Ca și în cazul clasei descrise anterior, capacitatea acestui tip de greutate nu va fi reținută într-un atribut, ci va fi calculată de fiecare dată când unui obiect de acest tip i se va solicita serviciul *capacitate()*.

Sistemul mai cuprinde și clasa *ColectieGreutati* care conține un tablou de greutateți (acestea reprezintă conținutul efectiv al colecției). Clasa *ColectieGreutati* va conține următoarele metode:

- *public void adauga(Greutate g)*: are rolul de a adăuga elemente în tabloul de greutateți. Presupunem că o colecție de greutateți are o capacitate maximă de greutateți care se setează prin intermediul constructorului.
- *public double medie()*: returnează greutatea medie a colecției (capacitate/numar de greutati).

Se cere:

- diagrama UML pentru clasele prezentate mai sus.
- implementarea claselor prezentate în diagramă.
- o metodă *main* în care se va crea un obiect *ColectieGreutati*, câteva greutateți simple, duble și multiple care vor fi adăugate colecției de greutateți. Se va afișa greutatea medie a colecției.

Bibliografie

1. Bruce Eckel. *Thinking in Java, 4th Edition*. Prentice-Hall, 2006. Capitolul Polymorphism.
2. Radu Marinescu, Carmen De Sabata. *Ingineria Programării 1. Îndrumător de laborator*. Casa Cărții de Știință, 1999. Lucrarea 6, Interfețe și polimorfism.
3. Robert C. Martin. *Agile Software Development. Principles, Patterns and Practices*. Prentice Hall, 2003. Capitolul 9, OCP: The Open-Closed Principle.