Lecţia 9

Pachete

În prima lucrare s-a accentuat faptul că un program orientat pe obiecte este compus din obiecte care interacţionează între ele prin apeluri de metode sau, altfel spus, prin transmitere de mesaje. După cum probabil s-a remarcat deja, aceasta e o viziune dinamică asupra unui program orientat pe obiecte, mai exact o viziune asupra a ceea ce se întâmplă la execuţia programului. Din punct de vedere static programul, sau mai exact codul său sursă, este organizat sub forma unui set de clase care prezintă implementarea obiectelor. Odată ce dimensiunea unui sistem software creşte, numărul de tipuri distincte de obiecte creşte şi deci şi numărul de clase/interfeţe. Astfel, se pune problema organizării sau grupării eficiente a claselor/interfeţelor. În această lucrare vom prezenta anumite aspecte de principiu legate de această organizare şi facilităţile puse la dispoziţie de Java pentru a o pune în practică.

9.1 Modularizarea

9.1.1 Arhitectura programelor orientate pe obiecte

În general, pentru a putea înţelege şi construi uşor un sistem complex de orice natură, acesta e văzut ca fiind compus fizic dintr-un ansamblu de subsisteme interconectate între ele. Într-o viziune similară, un sistem software de mari dimensiuni este văzut ca fiind compus fizic dintr-un ansamblu de subsisteme software sau *module*. Este important de reţinut că modulele sunt utilizate pentru a reda *arhitectura fizica* a sistemului software, pentru a reda părţile sale constituente.

Pentru a exemplifica ce înseamnă arhitectura fizică a unui sistem să ne gândim la sistemele de calcul (PC-urile) pe care le avem noi acasă. Care sunt părţile lor constituente? Dintre modulele cuprinse fizic de un calculator putem aminti procesorul, placa de bază, placa de reţea, placa video, placa de sunet. Evident există multe astfel de module într-un calculator.

Dar ce este efectiv un modul software? Ce conţine un astfel de modul? Răspunsul la această întrebare depinde din păcate de limbajul de programare la care ne raportăm. În Java, un modul poate fi implementat printr-un *pachet* care trebuie văzut ca un container în care se depune un grup de clase şi interfeţe. Spre deosebire de modulele care descriu arhitectura fizică a unui program, clasele şi interfeţele descriu *arhitectura logică* a modulelor ce le conţin şi deci implicit a sistemului software.

În exemplul anterior am spus că un calculator este compus fizic din mai multe module. Un astfel de modul este, de exemplu, placa de sunet. Pe placa de sunet găsim mai multe obiecte: tranzistoare, rezistenţe, condensatoare, circuite digitale, etc. Toate acestea interacţionează între ele urmând o anumită logică pentru a îndeplini principalele funcţii ale unei plăci de sunet: redare sunet, înregistrare sunet, comunicare cu placa de bază, etc. Într-un mod similar, clasele dintr-un modul descriu implementarea şi interacţiunile obiectelor care dau naştere funcţionalităţii sau comportamentului respectivului modul.

În acest context putem defini activitatea de modularizare. Principial modularizarea reprezintă procesul de structurare al unui sistem software în module. Este foarte important de menţionat că această structurare NU se face ad-hoc, ci pe baza unor principii de modularizare. Structurarea unui program orientat pe obiecte în module este o problemă poate la fel de dificilă ca identificarea setului corespunzător de obiecte care va exista în respectivul program şi depăşeste cu mult scopul acestei lecţii. Noi ne vom rezuma aici doar la anumite aspecte elementare. Astfel, putem vorbi de modularizare doar dacă rezultatul acestei activităţi prezintă aşa numita proprietate de modularitate.

9.1.2 Proprietatea de modularitate

Proprietatea de modularitate este un concept de bază în domeniul tehnologiei orientate pe obiecte şi nu numai.

Definiţie 2 *Modularitatea este proprietatea unui sistem care a fost descompus într-un set de module coezive şi slab cuplate.*

În definiţia de mai sus apar doi termeni foarte importanţi în contextul orientării pe obiecte a programelor: *cuplajul* şi *coeziunea*. Spunem că două module sunt cuplate atunci când ele sunt "conectate" între ele. Două module sunt slab cuplate atunci când gradul lor de interconectare este redus sau altfel spus, atunci când modificări aduse unui modul nu implică modificarea celuilalt. Cât priveşte cea de-a doua noţiune, spunem că un modul e coeziv dacă elementele conţinute de el, în cazul nostru clasele, au puternice legături între ele care să justifice gruparea lor la un loc.

De multe ori aţi auzit spunându-se că sistemele de calcul pe care le avem noi acasă (PC-urile) sunt modulare. De ce sunt ele modulare? Pentru că ele sunt fizic compuse dintr-un set de module coezive şi slab cuplate. Dintre aceste module amintim placa de baza, procesorul, placa video, placa de reţea, etc. Faptul că un modul este coeziv înseamnă că absolut tot ceea ce se găseşte în el este utilizat pentru îndeplinirea funcţiilor specifice modulului. De exemplu, toate componentele electronice de pe o placă video sunt destinate îndeplinirii funcţiilor de afişare pe ecranul monitorului. Faptul că două module sunt cuplate înseamnă că îndeplinirea funcţiilor proprii de către un modul depinde de utilizarea celuilalt. De exemplu, placa video şi placa de reţea nu sunt cuplate pentru că fiecare poate să-şi îndeplinească funcţiile independent una de alta. Pe de altă parte, placa video şi placa de bază sunt cuplate pentru că nici una nu poate funcţiona corect fără cealaltă. Totuşi este important de menţionat că placa video şi placa de bază sunt slab cuplate, deoarece este uşor să înlocuim un model de placă video cu un alt model ceva mai performant fără a trebui să schimbăm şi placa de bază.

9.1.3 Scopul modularizării

În dicuţia de până acum am prezentat conceptul de modularizare. Dar care este scopul? De ce vrem noi să modularizăm programele în general sau programele orientate pe obiecte în particular?

Imaginaţi-vă ce s-ar întâmpla de exemplu dacă pentru a testa o singură clasă într-un sistem ar trebui să compilăm întregul său cod, chiar şi acele părţi care nu au logic nici o legătură cu clasa noastră. Ar fi cam ineficient mai ales dacă compilarea durează câteva ore. Mai rău, ar fi foarte neplăcut ca pentru a testa aceeaşi clasă ar trebui să înţelegem o mare parte din restul sistemului. Aceasta ar necesita discuţii cu alte echipe de programatori care s-au ocupat de implementarea diferitelor alte părţi din sistem lucru care ar putea fi foarte ineficient mai ales dacă echipele sunt pe continente diferite (şi nici deplasarea până în căminele vecine nu e prea plăcută dacă are loc de 50 de ori pe zi).

Ei bine, scopul general este unul simplu: reducerea complexităţii sistemului şi implicit a costurilor de construcţie permiţând modulelor să fie proiectate, implementate şi revizuite în mod independent. Această independenţă poate permite modulelor să fie compilate şi testate individual. Prin urmare, este posibil ca diferite module să fie construite de echipe diferite de programatori care să lucreze independent. Mai mult, o echipă va lucra mult mai eficient deoarece ea poate să se concentreze exclusiv asupra funcţionalităţii oferite de modulul de care se ocupă fără a fi nevoită să înţeleagă sistemul în ansamblul său.

Calculatorul de acasă a fost gândit să fie modular. Astfel, placa de bază poate fi construită separat, placa de reţea separat, placa video separat, etc. Ce s-ar întâmpla dacă pentru a construi o placă de reţea ar trebui (contraintuitiv) să se cunoască în detaliu cum funcţionează placa video? Probabil nu ar fi prea mulţi producători de plăci de reţea şi acestea ar fi cam scumpe.

9.1.4 De la vorbe la fapte

Din discuţia de până acum reiese importanţa modularizării. Dar cum atingem acest scop? După cum am mai spus, o discuţie despre acest lucru depăşeşte scopul acestei lecţii. Totuşi, pentru a înţelege rolul anumitor construcţii de limbaj din Java vom menţiona un set restrâns de reguli de modularizare:

- implementarea unui modul trebuie să depindă numai de interfeţele altor module.
- interfaţa unui modul trebuie să cuprindă aspecte care sunt puţin probabil a se modifica.
- implementarea unui modul trebuie ascunsă de celelalte module.

Explicarea acestor reguli se face cel mai bine printr-un exemplu. Astfel, prima regulă spune că modul în care este implementată funcţionalitatea unei plăci de bază nu trebuie să depindă de modul de implementare a unei plăci video. Ea trebuie să depindă numai de interfaţa unei plăci video. Aşa stau lucrurile într-un calculator: placa de bază a fost construită ştiind doar că placa video are o interfaţă de comunicare PCI. A doua regulă spune că interfaţa unei plăci video trebuie să fie stabilă. Cu alte cuvinte, modul de comunicare al plăcii video nu trebuie să se schimbe des. Efectul acestor două reguli e simplu: într-un calculator putem schimba un model de placă video cu alt model mai performant fără a schimba placa de bază! Evident, dacă interfaţa plăcii video se schimbă, de exemplu e o placă cu interfaţă AGP, trebuie să schimbăm şi placa de bază (presupunând că ea nu are o magistrală AGP). Ultima regulă este un efect al aplicării principiului ascunderii informaţiei la nivelul modulelor. Dacă un modul depinde sau este interesat doar de interfaţa modulelor cu care comunică înseamnă că el nu e interesat de implementarea lor. Ca urmare, aplicând principiul ascunderii informaţiei la nivelul modulelor, trebuie să ascundem implementarea modulelor de orice alt modul.

9.2 Modularizarea programelor Java

Limbajul Java oferă suport pentru modularizarea programelor prin noţiunea de *pachet*. Din punct de vedere fizic, un program Java este compus dintr-o colecţie de pachete care la rândul lor sunt compuse fizic dintr-un număr de *unităţi de compilare*. Într-un sistem de calcul obişnuit, unui pachet îi corespunde un director, în timp ce unei unităţi

de compilare îi corespunde un fişier cod sursă. Este bine ca toate fişierele ce aparţin unui pachet să fie amplasate în directorul ce corespunde respectivului pachet.

Important Directorul părinte al directorului ce corespunde unui pachet va fi numit pe parcursul acestei lecţii *directorul de lucru* sau directorul *src*.

Toate clasele şi interfeţele declarate în unităţile de compilare care aparţin unui pachet sunt membri ai respectivului pachet. În continuare vom discuta câteva aspecte legate de lucrul cu clase şi interfeţe în contextul pachetelor.

9.2.1 Declararea pachetelor

După cum am spus înainte, un pachet este fizic compus din mai multe fişiere sursă, acestea din urmă conţinând declaraţiile claselor şi interfeţelor membre ale pachetului respectiv. Prin urmare, este logic să putem specifica apartenenţa unui fişier la un pachet. Acest lucru se realizează prin clauza *package* care poate fi plasată *numai pe prima* linie a fişierului.

```
//Acest cod este continut de un singur fisier, sa spunem fisier1.java
//si trebuie plasat intr-un director numePachet, mai exact src/numePachet
package numePachet;

class ClasaA {
    ...
}

class ClasaB {
    ...
}

class ClasaC {
    ...
}
```

Să considerăm exemplul de mai sus. Clauza *package* de la începutul fişierului indică faptul că *fisier1.java* aparţine pachetului *numePachet* şi prin urmare clasele *ClasaA*, *ClasaB* şi *ClasaC* sunt membre ale acestui pachet. Modul de reprezentare UML al pachetelor şi al apartenenţei unei clase la un pachet este exemplificat în Figurile 9.1 şi 9.2

Important Clauza *package* poate apare numai pe prima linie dintr-un fişier. Prin urmare este clar că un fişier sursă NU poate conţine clase ce sunt membre ale unor pachete diferite. Pe de altă parte, clasele declarate în fişiere diferite pot fi membre ale aceluiaşi pachet dacă respectivele fişiere au pe prima linie o clauză *package* cu nume de pachet identic.

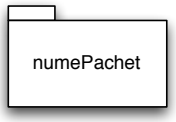


Figura 9.1: Reprezentarea UML a pachetelor.

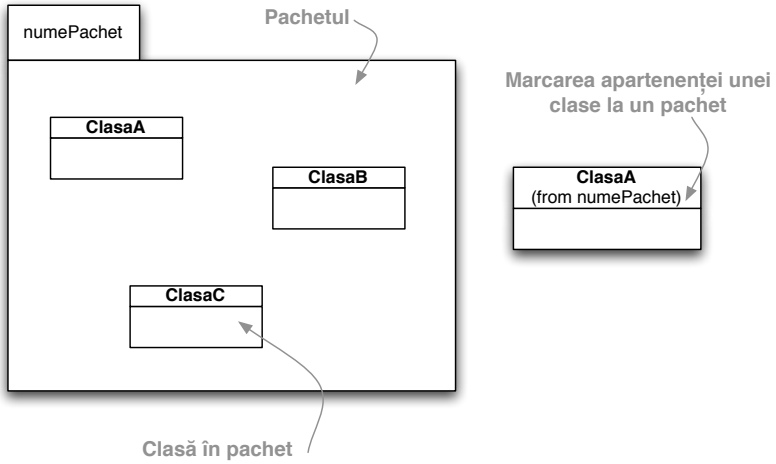


Figura 9.2: Modalităţi de reprezentare a claselor din pachete.

Până acum am scris programe Java fără să utilizăm clauza *package*. Cărui pachet aparţineau clasele declarate? Dacă nu se utilizează clauza *package* într-un fişier, se consideră implicit că toate declaraţiile conţinute de fişier aparţin unui pachet ce nu are nume sau care e anonim. Fizic, el corespunde directorului de lucru pe care l-am mai numit src.

9.2.2 Subpachetele. Numele complet al unui pachet

Java permite ca un pachet să fie fizic descompus în mai multe pachete (sau subpachete), subpachetele în alte pachete (sau subsubpachete), etc. Datorită acestui fapt putem avea două pachete cu acelaşi nume conţinute în pachete diferite. Prin urmare, pentru a identifica complet un pachet, nu este suficient să utilizăm numele său ci numele său complet. Acesta e format din numele "mic" al pachetului şi *numele complet* al pachetului ce-l conţine.

```
package numePachet.numeSubPachet.numeSubSubPachet;
```

Ca şi exemplu, plasând clauza *package* de mai sus într-un fişier, toate clasele şi interfeţele respectivului fişier vor fi membre ale pachetului *numeSubSubPachet* care e conţinut de pachetul *numeSubPachet* care la rândul său e conţinut de pachetul *numePachet* care la rândul său e conţinut de pachetul anonim.

Ce director îi va corespunde pachetului *numeSubSubPachet*? Dacă pachetului *numePachet* îi corespunde un director src/numePachet e natural ca pachetului *numeSubPachet* să-i corespundă un director src/numePachet/numeSubPachet. Urmând aceeaşi logică, pentru pachetul *numeSubSubPachet* vom avea un director src/numePachet/numeSubPachet/numeSubSubPachet.

9.2.3 Numele complet al claselor

În programele scrise până acum nu am făcut uz de facilităţile de modularizare puse la dispoziţie de Java. Mai exact, toate clasele erau membre ale pachetului anonim. Prin urmare, o clasă era complet identificată prin numele său. În contextul pachetelor, o clasă nu mai poate fi complet identificată doar prin numele său deoarece două clase pot avea acelaşi nume dacă sunt membre ale unor pachete diferite. Ca şi în cazul subpachetelor, o clasă e complet identificată prin numele ei *complet* care e format din numele ei "mic" şi numele complet al pachetului ce o conţine.

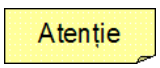
```
//Tot exemplul este continut intr-un singur fisier, sa zicem fisier2.java
//si trebuie plasat in directorul src/numePachetulNostru/numePachetulMeu
package numePachetulNostru.numePachetulMeu;

class ClasaD {

    public static void main(String argv[]) {
        ...
    }
}
```

Ca şi exemplu, clasa de mai sus este complet identificată prin următoarea succesiune de identificatori separaţi prin punct.

```
numePachetulNostru.numePachetulMeu.ClasaD
```



Să presupunem că dorim să lansăm în execuţie programul care începe în metoda *main* din exemplul de mai sus. Cum procedăm? Evident, folosind

numele complet al clasei *ClasaD*. Mai exact vom utiliza comanda *java numePachetulNostru.numePachetulMeu.ClasaD* aflându-ne în directorul de lucru (src).

9.2.4 Vizibilitatea conţinutului pachetelor

Aşa cum am amintit în discuţia de la începutul acestei lecţii, implementarea unui pachet trebuie ascunsă de alte pachete şi doar interfaţa unui pachet trebuie să fie vizibilă altor pachete. Cu alte cuvinte, un pachet trebuie să permită accesarea din exteriorul lui doar a claselor şi interfeţelor ce ţin de interfaţa pachetului, în timp ce clasele şi interfeţele ce ţin de implementarea lui trebuie ascunse. Java permite specificarea apartenenţei unei clase/interfeţe la interfaţa unui pachet prin utilizarea specificatorului de access *public* înaintea declaraţiei respectivei clase.

```
//Tot exemplul este continut intr-un singur fisier,
//ClasaDinInterfataPachetului.java
//care trebuie plasat in directorul src/unPachet
package unPachet;

public class ClasaDinInterfataPachetului {
    ...
}
```

Utilizând *public* în exemplul de mai sus specificăm faptul că respectiva clasă aparţine interfeţei pachetului şi poate fi accesată sau utilizată în exteriorul pachetului *unPachet*. Dacă specificatorul *public* ar fi lipsit, clasa de mai sus ar fi fost considerată ca ţinând de implementarea pachetului şi nu ar fi fost accesibilă din afara pachetului *unPachet*.

În programele de până acum nu am folosit niciodată specificatorul *public* înaintea claselor pe care le-am declarat. De ce puteam totuşi să le utilizăm? Foarte simplu. Pentru că noi nu am utilizat nici clauza *package* pentru a le distribui în pachete. Ca urmare, toate clasele făceau practic parte din acelaşi pachet (pachetul anonim), iar în interiorul unui pachet putem accesa sau utiliza orice clasă sau interfaţă declarată în acel pachet.

9.2.5 Accesarea conţinutului pachetelor

Până acum am văzut cum implementăm efectiv gruparea claselor şi a interfeţelor în pachete şi regulile de vizibilitate a claselor şi interfeţelor la nivelul pachetelor. Dar cum accesăm efectiv aceste entităţi? Evident acest lucru depinde de "locul" din care dorim să le accesăm. În pricipiu, clasele dintr-un pachet pot fi accesate din acelaşi pachet prin numele lor "mic" iar din afara pachetului "doar" prin numele lor complet (evident ele trebuie să fie accesibile sau, altfel spus, declarate publice).

```
//Tot exemplul este continut intr-un singur fisier, A1.java
//care trebuie plasat in directorul src/pachet1
package pachet1;

public class A1 {
    ...
}
```

```
//Tot exemplul este continut intr-un singur fisier, sa zicem fisier1.java
//care trebuie plasat in directorul src/pachet1
package pachet1;

class B1 {

    public void metodaB1() {
        A1 ob = new A1();
        //Se acceseaza clasa A1 membra a pachetului pachet1
        //Accesul se face prin numele ''mic''
    }
}
```

```
//Tot exemplul este continut intr-un singur fisier, sa zicem fisier2.java
//care trebuie plasat in directorul src/pachet2
package pachet2;

class A2 {

    public void metodaA2() {
        pachet1.A1 ob = new pachet1.A1();
        //Se acceseaza clasa A1 membra a pachetului pachet2
        //Accesul se face prin numele complet
    }
}

class B2 extends pachet1.A1 {
    ...
}
```

Important Dacă am accesa clasa A1 în *fisier2.java* utilizând doar numele ei "mic", compilatorul ar semnala o eroare.

Totuşi, necesitatea utilizării numelui complet al unei clase/interfeţe publice pentru a o accesa dintr-un alt pachet este destul de incomod. Acestă "deficienţă" poate fi eliminată

prin utilizarea clauzelor *import*. Ele pot apare numai la începutul unui fişier sursă şi pot fi precedate doar de o eventuală clauză *package*.

Astfel, în cadrul celui de-al treilea fişier din exemplul de mai sus, putem să accesăm clasa *A1* din pachetul *pachet1* utilizând numai numele clasei (adică *A1*) dacă introducem imediat după clauza *package* o clauză *import* de forma:

```
//Aceasta forma a clauzei import se mai numeste single type import
import pachet1.A1;
```

Mai mult, clauza *import* ar putea avea şi forma:

```
//Aceasta forma a clauzei import se mai numeste type import on demand
import pachet1.*;
```

Dacă folosim a doua formă a clauzei *import*, putem accesa în acelaşi fişier orice clasă (sau interfaţă) publică din pachetul *pachet1*, folosind doar numele lor "mic". Utilizând clauze *import*, conţinutul celui de-al treilea fişier va putea avea forma de mai jos, fără să apară nici o eroare de compilare.

```
//Tot exemplul este continut intr-un singur fisier, fisier2.java
//care trebuie plasat in directorul src/pachet2
package pachet2;
import pachet1.*;

class A2 {

    public void metodaA2() {
        A1 ob = new A1();
    }
}

class B2 extends A1 {
    ...
}
```

Toate clasele predefinite din Java sunt distribuite în pachete. În particular, clasele *String*, *Math*, clasele înfăşurătoare, etc. pe care le-am utilizat deja aparţin unui pachet denumit *java.lang*. Totuşi, noi nu am utilizat clauze *import* pentru a putea accesa aceste clase din alte pachete (în particular, din pachetul anonim cu care am lucrat până acum). Ei bine, pachetul *java.lang* este tratat într-un mod special de compilatorul Java, considerându-se implicit că fiecare fişier sursă Java conţine o clauză *import java.lang.*;*

Atenţie

Atenţie la eventuale conflicte de nume! Astfel, putem avea două pachete diferite care conţin câte o clasă cu acelaşi nume. Pentru detalii suplimentare legate de utilizarea clauzei import în astfel de situaţii consultaţi specificaţiile limbajului Java.

În exemplul de mai sus *pachet2* utilizează clase grupate logic în *pachet1*. Se spune că *pachet2* depinde de *pachet1*. Figura 9.3 exemplifică modul de reprezentare UML a relaţiei de dependenţă între două pachete.

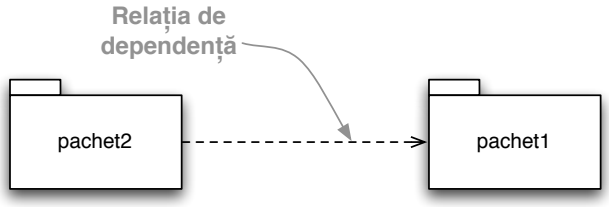


Figura 9.3: Reprezentarea UML a dependenţei între pachete.

9.2.6 Vizibilitatea membrilor claselor

În lecţiile anterioare am văzut că drepturile de access la membrii unui clase pot fi specificaţi explicit prin specificatori de access. Tot atunci am discutat regulile de vizibilitate date de specificatorii *private*, *public* şi *protected*. În continuare vom extinde aceste reguli în contextul pachetelor.

Atunci când nu se specifică explicit specificatorul de access la membrii unei clase (câmpuri sau metode) se consideră implicit că accesul la respectivul membru este de tip *package*. Aceasta înseamnă că respectivul membru este accesibil de oriunde din interiorul pachetului ce conţine clasa membrului respectiv, dar numai din interiorul respectivului pachet. Este important de menţionat că NU există un specificator de access *package*. Dacă dorim ca accesul la membrul unei clase să fie de tip *package*, pur şi simplu nu punem nici un specificator de access.

Pe de altă parte, în lecţia legată de relaţia de moştenire am discutat regulile de vizibilitate date de specificatorul de acces *protected*. Acolo am spus că *în general* un membru *protected* al unei clase este "asemănător" unui membru *private* dar care este accesibil şi din toate clasele ce moştenesc clasa membrului respectiv. Am accentuat "în general" pentru că acesta este comportamentul teoretic al specificatorului *protected*. În plus faţă de această regulă, în Java un membru declarat *protected* este accesibil de oriunde din interiorul pachetului ce conţine clasa membrului respectiv.

Sfat Încercaţi să evitaţi această particularitate a limbajului Java.

Prin urmare, regulile de acces ale unei clase B la membrii unei clase A se extind în modul următor:

- Dacă clasele A şi B sunt conţinute în acelaşi pachet atunci B are acces la toţi membrii clasei A ce nu sunt declaraţi *private*.
- Dacă clasele A şi B sunt conţinute de pachete distincte şi B nu moşteneşte A atunci B are acces la toţi membrii clasei A ce sunt declaraţi *public*.
- Dacă clasele A şi B sunt conţinute de pachete distincte şi B moşteneşte A atunci B are acces la toţi membrii clasei A ce sunt declaraţi *public* sau *protected*.

Atenţie În cazul în care clasa A nu este declarată ca aparţinând interfeţei pachetului ce o conţine, atunci ea nu este accesibilă din afara respectivului pachet şi implicit nici un membru al respectivei clase nu poate fi accesat din exteriorul pachetului în care ea se află.

Aceste reguli de vizibilitate sunt exemplificate în porţiunile de cod de mai jos.

```
//Tot exemplul este continut intr-un singur fisier, A1.java
//care trebuie plasat in directorul src/pachet1
package pachet1;

public class A1 {

    private int x;
    public int y;
    protected int z;
    int t;
}
```

```
//Tot exemplul este continut intr-un singur fisier, fisier1.java
//care trebuie plasat in directorul src/pachet1
package pachet1;

class B1 {
```

```
    public void metodaB1() {
        A1 ob = new A1();
        ob.x = 1; //Eroare
        ob.y = 1; //Corect
        ob.z = 1; //Corect
        ob.t = 1; //Corect
    }
}
```

```
//Tot exemplul este continut intr-un singur fisier, fisier2.java
//care trebuie plasat in directorul src/pachet2
package pachet2;

class A2 {

    public void metodaA2() {
        pachet1.A1 ob = new pachet1.A1();
        ob.x = 1; //Eroare
        ob.y = 1; //Corect
        ob.z = 1; //Eroare
        ob.t = 1; //Eroare
    }
}

class B2 extends pachet1.A1 {

    public B2() {
        x = 1; //Eroare
        y = 1; //Corect
        z = 1; //Corect
        t = 1; //Eroare
    }

    public void metodaB2() {
        pachet1.A1 ob = new pachet1.A1();
        ob.x = 1; //Eroare
        ob.y = 1; //Corect
        ob.z = 1; //Eroare (Da! E eroare!)
        ob.t = 1; //Eroare
    }
}
```

Atenție Din punctul de vedere al regulilor de vizibilitate, un pachet şi un subpachet al său se comportă ca şi cum ar fi două pachete oarecare!

Poate vă întrebaţi de ce în exemplul de mai sus nu e posibil să accesăm câmpul *protected z* din clasa *pachet1.A1* în metoda *metodaB2*. Doar *B2* extinde clasa *pachet1.A1*. Aceasta pentru că, din punct de vedere pur teoretic, un membru *protected* poate fi accesat din clasa sa şi din orice subclasă a clasei sale dar în al doilea caz doar pentru instanţa *this*. Dacă am pune toate clasele de mai sus în acelaşi pachet accesul ar fi posibil. Aceasta pentru că specificatorul *protected* din Java diferă de specificatorul *protected* din teorie permiţând accesul, de oriunde din interiorul pachetului ce conţine clasa respectivului membru. În cazul nostru, membrul *z* e *protected* şi ar putea fi accesat din orice clasă ce aparţine aceluiaşi pachet ca şi clasa sa (repetăm: dacă am pune toate clasele în acelaşi pachet).

În UML, vizibilitatea membrilor ce au politică de access de tip package se marchează cu simbolul ˜. Figura 9.4 exemplifică modul de reprezentare a clasei *A1* din exemplul anterior.

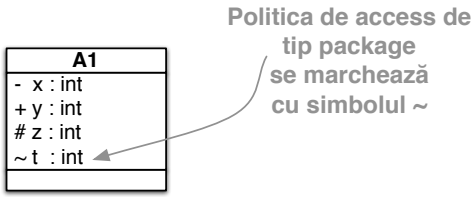
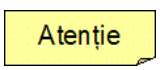


Figura 9.4: VIZIBILITATEA DE TIP PACKAGE ÎN UML.

9.2.7 Gestionarea fişierelor .java şi .class

Până acum am discutat modul în care realizăm distribuirea claselor şi a interfeţelor pe care le declarăm într-o aplicaţie în diferite pachete. Dar cum repartizăm declaraţiile claselor şi interfeţelor pe fişiere sursă? La prima vedere am fi tentaţi să spunem că din moment ce repartizarea pe pachete se face la nivel de fişier sursă, am putea declara toate clasele şi interfeţele unui pachet în acelaşi fişier sursă. Ei bine acest lucru nu e posibil totdeauna. Motivul e simplu: orice clasă sau interfaţă publică dintr-un pachet trebuie declarată într-un fişier al cărui nume e format din numele clasei/interfeţei urmat de terminaţia *.java*.

Nerespectarea acestei reguli duce la o eroare de compilare. Acesta e motivul pentru care, în exemplele de până acum, toate clasele publice erau declarate într-un fişier a cărui nume respecta regula anterioară.



Într-un fişier putem declara mai multe clase şi interfeţe, dar datorită regulii de mai sus într-un fişier putem avea maxim o clasă sau o interfaţă

publică. Restul claselor/interfeţelor nu vor putea fi publice şi implicit nu vor putea fi accesate din afara pachetului.

Pe de altă parte, odată ce compilăm un fişier cu cod sursă, vom obţine câte un fişier *.class* pentru fiecare clasă şi interfaţă din fişierul sursă compilat. În mod obligatoriu aceste fişiere *.class* trebuie amplasate într-o structură de directoare care să reflecte direct pachetul din care fac ele parte. Să considerăm un exemplu.

```
//Tot exemplul este continut intr-un singur fisier A1.java
package pachet1;

public class A1 {
    ...
}

class B1 {
    ...
}
```

```
//Tot exemplul este continut intr-un singur fisier cu nume oarecare
//din moment ce nu avem declaratii publice
package pachet2;

class A2 {
...
}
```

```
//Tot exemplul este continut intr-un singur fisier cu nume oarecare
//din moment ce nu avem declaratii publice
package pachet2.subpachet21;

class B21 {
    ...
}
```

După ce compilăm aceste fişiere cu cod sursă, fişierele *.class* obţinute trebuie repartizate după cum urmează:

- Fişierele *A1.class* şi *B1.class* într-un director denumit *pachet1*.
- Fişierele *A2.class* într-un director denumit *pachet2*.
- Fişierele *B21.class* într-un director denumit *subpachet1* care este subdirector al directorului *pachet2*.

Această distribuire a fişierelor rezultate în urma compilării nu trebuie însă făcută manual. Ea este realizată automat de compilator dacă se utilizează un argumet al compilatorului Java ca în exemplul de mai jos. Argumentul *-d numeDirector* îi spune compilatorului să distribuie automat fişierele *.class* generate într-o structură de directoare corespunzătoare a cărei rădăcină este directorul *numeDirector*.

```
javac -d numeDirector nume_fisier_compilat
```

Pentru a compila toată aplicaţia va trebui să compilăm toate fişierele sursă ce o compun. În acest scop, putem da ca argumente comenzii *javac* o listă de fişiere, listă care poate fi creată utilizând şi nume generice (*.java). Pe de altă parte am putea compila toată aplicaţia compilând pe rând fiecare pachet. În acest caz foarte importante sunt opţiunile de compilare *-sourcepath numeDirector(e)* şi *-classpath numeDirector(e)*.

Astfel, în momentul în care compilatorul găseşte în pachetul compilat un acces la o clasă din afara pachetului, va trebui cumva să ştie ce conţine acea clasă, mai exact, are nevoie de fişierul *.class* asociat ei. Opţiunea *-classpath numeDirector(e)* îi furnizează compilatorului o listă de directoare [1] [2] care conţin pachete (directoare) cu fişiere *.class*. Pe baza acestei liste şi pe baza numelui complet al clasei referite compilatorul poate localiza fişierul *.class* necesar.

Problemele apar în momentul în care există dependenţe circulare: pachetul *alpha* trebuie compilat înainte de *beta* pentru că utilizează o clasă din al doilea pachet, iar al doilea pachet trebuie compilat înainte de primul pentru că el utilizează o clasă din pachetul *alpha*. Ce e de făcut? Deşi aceste dependenţe circulare indică probleme serioase de modularizare (ambele pachete trebuie compilate simultan) compilatorul Java ne ajută prin opţiunea *-sourcepath numeDirectoare(e)*. Ea indică o listă de directoare ce conţin pachete (directoare) cu fişiere sursă. Astfel, dacă de exemplu compilăm tot pachetul *alpha*, în momentul în care compilatorul are nevoie de fişierul *.class* al unei clase din *beta* (care nu a fost încă compilat!), compilatorul va căuta pe baza listei *sourcepath* şi pe baza numelui complet al clasei referite fişierul sursă care conţine acea clasă şi-l va compila şi pe el.

Dacă compilatorul nu poate obţine de nicăieri fişierul *.class* al unei clase utilizate în pachetul compilat va genera o eroare de compilare.

[1] lista poate conţine şi fişiere .jar, acestea reprezentând de fapt un director împachetat folosind utilitarul jar

[2] în Windows separatorul elementelor listei este ; iar în Linux :

Din discuţia de mai sus s-ar părea că e destul de complicat să compilăm individual fiecare pachet al unei aplicaţii şi ar fi mai bine să compilăm totul odată. Răspunsul e simplu: e complicat dar necesar mai ales când echipe diferite de programatori dezvoltă individual pachete diferite ale unei aplicaţii uriaşe. Java a introdus pachetele pentru a ajuta la dezvoltarea de aplicaţii mari permiţând diferitelor echipe printre altele să-şi compileze "individual partea" lor de aplicaţie.

Compilarea individuală a unui pachet e posibilă doar dacă sistemul a fost modularizat corespunzător (vezi regulile de modularizare). Dacă, de exemplu, interfaţa unui pachet nu e stabilă atunci e foarte probabil să apară conflicte (şi la propriu şi la figurat) între echipele ce dezvoltă respectivul modul şi cele care dezvoltă module ce depind de respectiva interfaţă. Practic, ele nu vor mai fi echipe independente. La fel se întâmplă şi în cazul dependenţelor circulare. De aici se poate deduce o altă regulă de modularizare: nu permiteţi cicluri în graful de dependenţă între module.

9.3 Exerciţii

1. Fie codul de mai jos. Presupunem că dorim să mutăm doar clasa *B* în *pachet2* şi, în continuare, *B* moşteneşte *A*. Arătaţi şi explicaţi ce modificări trebuie efectuate pentru ca programul să fie în continuare compilabil.

```
//Continut fisier x.java
package pachet1;
class A {}
```

```
//Continut fisier y.java
package pachet1;
class B extends A {}
```

2. Să se scrie o clasă ce modelează conceptul de stivă. Elementele ce pot aparţine stivei la un moment dat sunt nişte figuri geometrice. Fiecare figură e caracterizată print-un punct de coordonate O(x,y) şi de alte elemente specifice funcţie de tipul figurii. Concret, putem avea figuri de tip Cerc şi Pătrat.

   Un obiect de tip Cerc are punctul de coordonate O(x,y) care reprezintă originea cercului; are un atribut pentru stocarea razei cercului; este egal cu un alt obiect Cerc dacă cele două obiecte au aceeaşi origine şi razele de lungimi egale; e afişat sub forma - "Cerc:" urmat de coordonatele originii şi de rază.

   Un obiect de tip Pătrat are punctul de coordonate O(x,y) care reprezintă colţul din

stânga-sus al pătratului; are punctul de coordonate P(x,y) care reprezintă colţul din stânga-jos al pătratului; este egal cu un alt obiect Pătrat dacă cele două obiecte au aceeaşi arie; e afişat sub forma - "Patrat:" urmat de coordonatele stânga-sus şi de latura pătratului.

Singura modalitate pentru setarea atributelor figurilor descrise mai sus e prin intermediul constructorilor!!!

Dimensiunea stivei este setată prin intermediul unui constructor ce primeşte ca parametru un *int*. Dacă parametrul primit este negativ atunci acest constructor va genera o eroare "neverificată". Clasa care modelează stiva pune la dispoziţia unui client:

(a) o metodă pentru introducerea unui element în stivă. Această metodă primeşte ca parametru elementul ce se doreşte a fi introdus. Dacă nu mai există loc în stivă pentru introducerea unui nou element, se va genera o excepţie prin care clientul unui obiect de acest tip (deci, clientul stivei) va fi informat de faptul că momentan nu mai pot fi introduse elemente. Întrucât nu vrem ca să existe în stivă mai multe figuri identice, încercarea de a introduce o figură care e deja stocată va genera o excepţie ce va furniza mesajul "Elementul" + (parametrul primit) + "este deja stocat".
(b) o metodă pentru returnarea şi ştergerea ultimului element introdus în stivă. Dacă stiva este goală, acest lucru va fi semnalat tot printr-o excepţie.
(c) o metodă pentru afişarea conţinutului stivei. Elementele vor fi afişate în ordinea UltimulStocat...PrimulStocat.

În vederea creării de figuri geometrice se vor defini mai multe pachete, unele pentru citire şi altele pentru scriere.

(a) pachetul *reader.input* - va conţine o interfaţă ce va avea cel puţin două metode: una pentru citirea atributelor unui obiect de tip Figura şi alta pentru citirea unei opţiuni (metoda care citeşte o figură va returna un obiect de tip Figura iar metoda pentru citirea unei opţiuni va returna un int, 1, 2, 3 sau 4).
(b) pachetul *reader.output* - va conţine o interfaţă cu o singură metodă ce va primi ca parametru un obiect stivă; implementările acestei metode nu vor face altceva decât să afişeze corespunzător stiva pe ecran sau într-un fişier.
(c) pachetul *reader.input.text* - va avea o clasă care va implementa interfaţa din pachetul *reader.input* astfel încât citirile să se facă de la tastatură.
(d) pachetul *reader.output.text* - va avea o clasă care va implementa interfaţa din pachetul *reader.output* astfel încât afişarea stivei să se facă pe ecran, în mod text.

Se va construi cel puţin un pachet care va conţine clasele din prima parte a problemei. Se va scrie o metodă *main* în care se va instanţia o stivă precum şi două obiecte

de tipul interfeţelor definite în pachetele *reader.input*, respectiv *reader.output*. Prin intermediul unui obiect ce implementează interfaţa din pachetul *reader.input*, se vor citi opţiuni (până la citirea opţiunii 4) şi în funcţie de opţiunile citite se vor apela cele trei metode ale stivei.

Bibliografie

1. Grady Booch, *Object-Oriented Analysis And Design With Applications*, Second Edition, Addison Wesley, 1997.
2. Martin Fowler. *UML Distilled, 3rd Edition.* Addison-Wesley, 2003.
3. Sun Microsystems, *Java Language Specification.* http://java.sun.com/docs/books/jls/, Capitolul 9, 2000.