Lecţia 12

Probleme frecvente în cod. Javadoc. Jar

12.1 Câteva probleme frecvente în cod

Martin Fowler prezintă în *Refactoring: Improving the Design of Existing Code* 22 de carenţe de proiectare ce pot să apară în cadrul sistemelor orientate pe obiecte (cunoscute şi sub numele de *bad smells in code*). În această secţiune vom prezenta câteva dintre ele.

12.1.1 Cod duplicat

Prin cod duplicat înţelegem existenţa într-o aplicaţie a mai multor porţiuni de cod identice. Duplicarea codului îngreunează într-o mare măsură dezvoltarea şi modificarea ulterioară a aplicaţiei.

Să presupunem că definim clasa *Masina* de mai jos. În cadrul metodei *porneste* a fost introdusă accidental o eroare deoarece în cadrul ei se depăşesc limitele tabloului referit de *usi*.

```
class Masina {

    protected Motor motor;
    protected Usa[] usi;

    public Masina(Motor motor, Usa[] usi) {
        this.motor = motor;
        this.usi = usi;
    }

    public void porneste() {
```

```
        for (int i=0; i<=usi.length; i++)
            usi[i].blocheaza();
        motor.demareaza();
    }
    ...
}
```

Să presupunem acum că înainte de a testa clasa *Masina* se scrie o altă clasă, *MasinaABS* iar în cadrul metodei suprascrise *porneste* se face o copiere a corpului metodei din clasa de bază. Evident, se copiază şi eroarea!!!

```
class MasinaABS extends Masina {

    public MasinaABS(Motor motor, Usa[] usi) {
        super(motor, usi);
    }

    public void porneste() {
        System.out.println("Masina ABS");
        for (int i=0; i<=usi.length; i++)
            usi[i].blocheaza();
        motor.demareaza();
    }
    ...
}
```

Este clar faptul că eliminarea erorii din clasa derivată ar putea să nu aibe loc în momentul în care ea e detectată şi eliminată din clasa *Masina*, pentru că pur şi simplu nu se mai ştie de unde, ce, şi cât cod a fost duplicat.

Duplicarea de cod dintr-un program măreşte artificial dimensiunea codului, măreşte efortul necesar eliminării erorilor (o eroare nefiind localizată într-un singur loc) precum şi efortul necesar pentru introducerea de noi cerinţe în cadrul aplicaţiei. Mecanismul *copy-paste* rezolvă superficial probleme! În cadrul exemplului anterior, un simplu apel *super.porneste()* în metoda *porneste* din clasa *MasinaABS* elimină duplicarea.

12.1.2 Listă lungă de parametri

O listă lungă de parametri la o metodă este greu de înţeles şi poate deveni inconsistentă şi greu de utilizat când este subiectul unor frecvente modificări.

```
class Sertar {

    private int lungime, inaltime, adancime;

    public Sertar(int lungime, int inaltime, int adancime) {
        this.lungime = lungime;
        this.inaltime = inaltime;
        this.adancime = adancime;
    }

    public int getLungime() {
        return lungime;
    }

    public int getInaltime() {
        return inaltime;
    }

    public int getAdancime() {
        return adancime;
    }
}
class Etajera {

    private Sertar[] sertare = new Sertar[2];

    public Etajera(int lungime, int inaltime, int adancime) {
        sertare[0] = new Sertar(lungime, inaltime, adancime);
        sertare[1] = new Sertar(lungime, inaltime, adancime);
    }

    //Returneaza o singura lungime, sertarele fiind suprapuse
    public int getLungime() {
        return sertare[0].getLungime();
    }

    public int getInaltime() {
        return sertare[0].getInaltime() + sertare[1].getInaltime();
    }

    public int getAdancime() {
        return sertare[0].getAdancime();
    }

    public Sertar[] getSertare() {
        return sertare;
    }
}
```

Presupunem că o etajeră este formată din două sertare, ambele sertare având aceleaşi dimensiuni. Implementările pentru clasele *Sertar* şi *Etajera* sunt prezentate mai sus. Dorim să definim o metodă statică care să verifice dacă o etajeră încape într-un spaţiu oarecare de dimensiuni L x A x H. Metoda definită, în loc să aibe patru parametri (dimensiunile L x A x H precum şi o referinţă spre un obiect etajeră), are 5 parametrii, dimensiunile L x A x H precum şi două referinţe spre sertarele incluse de etajeră. Au fost trimise două referinţe spre obiecte de tip sertar fiindcă, în fond, o etajeră este alcătuită din două sertare suprapuse.

```
class Verifica {

    public static boolean incapeEtajera(int L, int A, int H,
                                        Sertar s1, Sertar s2) {
        int LS, AS, HS;

        //Calculeaza dimensiunile etajerei

        LS = s1.getLungime();
        AS = s1.getAdancime();
        HS = s1.getInaltime() + s2.getInaltime();

        return ((L>LS) && (A>AS) && (H>HS));
    }
}
```

După cum se observă, în interiorul metodei *incapeEtajera* se calculează dimensiunile etajerei în care sunt dispuse cele două sertare referite de *s1* şi *s2*. Acest fapt implică o cunoaştere a modului în care sunt dispuse cele două sertare, adică a unei particularităţi de realizare (implementare) a unei etajere.

Pe de o parte, schimbarea implementării etajerei (spre exemplu, dispunerea pe orizontală a celor două sertare) necesită şi modificarea implementării metodei *incapeEtajera* pentru obţinerea unei validări corecte. Din păcate, calcule asemănătoare cu cele din cadrul metodei *incapeEtajera* pot exista în mai multe locuri în cadrul aplicaţiei şi e posibil să uităm să efectuăm peste tot modificările necesare, în acest mod fiind facilitată introducerea de erori în aplicaţie.

Pe de altă parte, componenţa unei etajere ar putea fi modificată, în sensul că am putea avea etajere formate din trei sertare. Este evident faptul că trebuie modificată lista de parametri a metodei *incapeEtajera* precum şi toate entităţile din cadrul aplicaţiei care apelează metoda. Dacă însă am fi transmis metodei noastre o singură referinţă spre un obiect de tip *Etajera* iar în loc de a calcula dimensiunile etajerei, le-am fi obţinut direct prin intermediul acestei referinţe, metoda *incapeEtajera* precum şi entităţile apelante nu ar fi trebuit să fie modificate la schimbarea structurii etajerei.

Trimiterea la apelul unei metode de referinţe spre atributele unui obiect în loc de o referinţă spre obiectul căruia aparţin atributele este o practică defectuasă pentru aplicaţiile orientate pe obiecte şi se manifestă de multe ori prin liste lungi de parametri.

12.1.3 Instrucţiuni switch şi if-else-if

Mai jos e definită excepţia *TablouException* ce e generată în diferite situaţii anormale ce pot apare la apelarea metodelor clasei *Tablou*. Această clasă permite adăugarea de întregi într-o colecţie de dimensiune limitată precum şi obţinerea unui element de pe o anumită poziţie. Evident, parametrii metodelor clasei *Tablou* pot avea valori eronate pentru obiectul asupra căruia se cere efectuarea respectivelor servicii, acest fapt fiind semnalat prin emiterea unei excepţii. După cum reiese din definiţia clasei, există mai multe motive pentru care se poate emite excepţia.

```
class TablouException extends Exception {

    private String message;

    public TablouException(String message) {
        this.message = message;
    }

    public String toString() {
        return message;
    }
}
```

```
class Tablou {

    private int[] tablou;
    private int nrElem = 0;

    public Tablou(ind dim) {
        tablou = new int[dim];
    }

    public void addElement(int el) throws TablouException {
        if(nrElem < tablou.length)
            tablou[nrElem++] = el;
        else
            throw new TablouException("Tablou Plin");
    }
```

```
    public int getElementAt(int pos) throws TablouException {
        if(pos >= tablou.length)
            throw new TablouException("Pozitie Invalida");
        if(nrElem <= pos)
            throw new TablouException("Prea Putine Elemente");
        return tablou[pos];
    }
}
```

Totuşi, uneori este necesară execuţia anumitor instrucţiuni în funcţie de motivul concret care a generat excepţia. Evident, motivul concret al excepţiei poate fi aflat doar din mesajul ei. Acest fapt face ca întotdeauna când trebuie să fie luate decizii în sistem legate de acest aspect să apară o înşiruire de instrucţiuni *if-else-if*.

```
public static oMetoda(Tablou t) {
    int pos = 29;
    try {
        t.addElement(5);
        t.addElement(12);
        int el = t.getElementAt(pos);
        ...
    } catch(TablouException e) {
        if (e.toString().equals("Tablou Plin")) {
            System.err.println("Adaugarea in tablou nu s-a putut face.");
        } else if (e.toString().equals("Pozitie Invalida")) {
            System.err.println("Dimensiunea tabloului este mai mica.");
            pos = -1;
        } else {
            System.err.println("Nu exista atatea elemente in tablou.");
        }
    }
    ...
}
```

Atenţie

În multe situaţii, în loc de lanţuri *if-else-if* poate apare o instrucţiune *switch*. O altă modelare a excepţiei de mai sus ar putea conduce la utilizarea unui *switch* pentru a discerne între diferitele situaţii anormale.

Utilizarea instrucţiunii *switch* precum şi a lanţurilor *if-else-if* în scopul descris mai sus conduce însă la apariţia duplicărilor de cod de fiecare dată când e nevoie să se afle cauza concretă ce a generat excepţia. Mai mult, în viitor, metodele clasei *Tablou* ar putea emite excepţia *TablouException* şi datorită altor cauze. Prin urmare va fi necesar să căutăm prin tot programul locurile în care se testează motivul apariţiei excepţiei şi să mai adăugăm o ramură *if-else* în acele locuri.

Soluţia eliminării instrucţiunilor *if-else-if* este crearea de subclase ale clasei *TablouException* pentru fiecare motiv ce poate conduce la emiterea de excepţii în interiorul metodelor clasei *Tablou*.

Într-un program orientat pe obiecte existenţa unor lanţuri *if-else-if* sau a unor instrucţiuni *switch* cu rolul de a discerne între diferite "feluri de obiecte este, de cele mai multe ori, un semn al lipsei utilizării polimorfismului!

12.1.4 Data Class

O clasă de tip Data Class este o clasă ce conţine doar atribute precum şi metode pentru setarea, respectiv returnarea valorilor atributelor conţinute (aceste metode se numesc metode accesor). În multe situaţii, clienţii unei astfel de clase, în loc să ceară efectuarea de servicii de la instanţele clasei, preiau datele de la obiecte in scopul efectuării "serviciilor" ce ar fi trebuit furnizate de clasa de date.

Imaginaţi-vă o clasă *Telefon* definită ca în exemplul de mai jos. Oare ce am putea face cu o instanţă a acestei clase? Din păcate, o instanţă a acestei clase nu oferă serviciile specifice unui telefon (formează număr, răspunde la apel, porneşte/opreşte telefon). Un client al unui obiect de acest tip, pentru a porni spre exemplu telefonul, ar trebui să-i ceară acestuia să-i furnizeze ecranul iar apoi să-i ceară ecranului să se pornească. Din păcate, în cele mai multe cazuri, pornirea unui telefon nu implică numai pornirea unui ecran. Pentru realizarea cu succes a operaţiei de pornire a telefonului se impune probabil efectuarea mai multor operaţii similare. Astfel, un client al unui telefon ajunge să fie obligat să cunoască funcţionarea în detaliu a unui astfel de aparat. În acelaşi timp, un obiect telefon nu are un singur client, ci mai mulţi. Prin urmare, este uşor de înţeles că toţi clienţii telefonului trebuie să cunoască detalii de implementare ale telefonului pentru a-l utiliza. În acest mod complexitatea sistemului poate creşte foarte mult.

```
class Telefon {

    private Ecran ecran;
    private Buton onOff;
    .... //Alte componente

    public void setEcran(Ecran ecran) {
        this.ecran = ecran;
    }

    public Ecran getEcran() {
        return ecran;
    }
```

```
    public void setOnOff(Buton onOff) {
        this.onOff = onOff;
    }
    public Buton getOnOff() {
        return onOff;
    }
    ...
}
```

Imaginaţi-vă că o firmă constructoare de maşini ar furniza în loc de o maşină ce răspunde la anumite comenzi, componentele maşinii, cerându-le clienţilor să le folosească în mod direct. La fel se petrec lucrurile şi in programarea orientată pe obiecte, clienţii unui obiect fiind interesaţi de posibilele servicii furnizate de către un obiect şi nicidecum de atributele sale!!! Data class-urile apar din cauza deficienţelor de modelare a obiectelor din program, în exemplul de mai sus a unui telefon. Eliminarea acestei probleme se realizează principial prin identificarea şi introducerea în clasa de date a serviciilor necesare clienţilor clasei respective.

12.2 Despre javadoc şi jar

12.2.1 Instrumentul software javadoc

În cadrul mai multor lecţii au fost făcute trimiteri la documentaţia Java oferită de firma Sun la adresa *http://java.sun.com/j2se/1.5.0/docs/api*. Această documentaţie, în formatul HTML, a fost creată folosind instrumentul software *javadoc*, instrument obţinut odată cu instalarea platformei Java.

În figura 12.1 e prezentată documentaţia aferentă clasei *Object*. După cum se vede, este precizat scopul clasei precum şi al fiecărei metode existente în această clasă. Şi noi putem genera, folosind *javadoc*, documentaţie aferentă claselor scrise de noi. Pentru a introduce în documentaţia generată informaţii privind scopul clasei, al metodelor, al parametrilor unei metode, etc. e necesară inserarea în cod a *comentariilor de documentare*. Acest tip de comentarii se inserează în cod între */*** şi **/*.

Atenţie

Comentariile de documentare sunt singurele comentarii recunoscute de *javadoc*. Dacă comentariile sunt de tip */** şi **/* nu se vor produce afişările corespondente în documentaţia HTML generată de *javadoc*.

În interiorul comentariilor de documentare pot fi introduse tag-uri *javadoc*. Tag-urile *javadoc* se inserează după caracterul @ (caracter ce poate fi precedat doar de spaţii şi, opţional, de ***) şi permit generarea unei documentaţii detaliate şi uniforme.

Tag	Descriere
@author nume	Adaugă o intrare de tip Author în documentaţie.
@param nume descriere	Adaugă parametrul cu numele şi descrierea specificată în secţiunea de parametri a unei metode sau a unui constructor.
@return descriere	Adaugă o intrare de tip Returns în documentaţia aferentă unei metode. Descrierea trebuie să cuprindă tipul returnat şi, dacă este cazul, plaja de valori returnată.
@throws nume descriere	Adaugă o intrare de tip Throws în documentaţia specifică unei metode sau al unui constructor; nume este numele excepţiei ce poate fi emisă în interiorul metodei.
@version	Adaugă o intrare de tip Version în documentaţie.

Tabelul 12.1: CÂTEVA TAG-URI JAVADOC.

Fiecare comentariu de documentare trebuie să fie plasat înainte de entitatea comentată (clasă, interfaţă, constructor, metodă, atribut, etc.). Mai jos este documentată o parte din clasa *Tablou* definită în secţiunea 12.1.3.

```
/**
  * Aceasta clasa stocheaza elemente intregi intr-un tablou si permite
  * accesarea unui element prin intermediul unui index.
  * @author LooseResearchGroup
  */
class Tablou {

    /**
    * Returneaza elementul de pe pozitia indicata.
    * @param pos pozitia de pe care se returneaza elementul
    * @throws TablouException In cazul in care pos este parametru invalid
    */
    public int getElementAt(int pos) throws TablouException {
       if(pos>=tablou.length) throw new TablouException("Pozitie Invalida");
       if(nrElem<=pos) throw new TablouException("Prea Putine Elemente");
       return tablou[pos];
    }
}
```

Pentru obţinerea documentaţiei trebuie apelat

```
javadoc -sourcepath fisiere.java
```

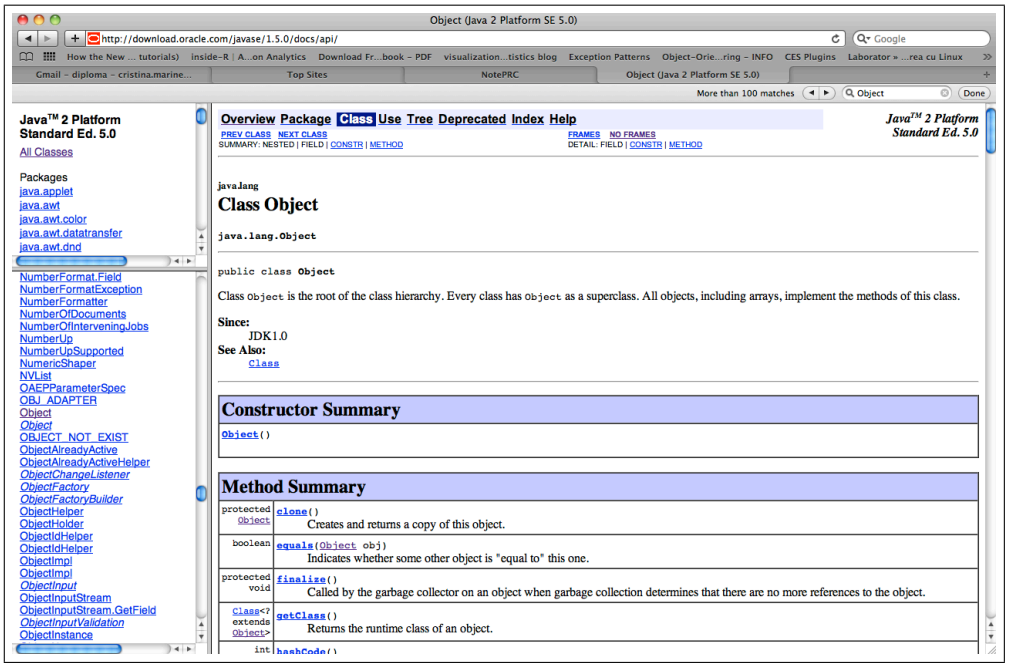

Figura 12.1: Documentaţia clasei Object oferită de Java API.

12.2.2 Instrumentul software jar

Până acum toate aplicaţiile care au fost create constau din fişierele sursă respectiv din fişierele cu extensia *class* rezultate în urma compilării surselor. Atunci când o aplicaţie este furnizată unui client nu e indicat ca respectivul client să primească şi fişierele sursă ale aplicaţiei! Ca urmare, un client primeşte o mulţime de fişiere cu extensia *class*. Totuşi, livrarea la un client a mai multor fişiere cu extensia *class* nu se recomandă şi nu e prea profesionistă.

Se recomandă ca aplicaţiile Java (fişierele class asociate ei) să se livreaze clienţilor într-o arhivă *jar*. Pentru realizarea unei arhive folosind instrumentul software *jar*, obţinut odată cu instalarea platformei Java, trebuie să executăm în linia de comandă:

```
jar cf app.jar fisier1 ... fisierN
```

Opţiunile cf indică faptul că se va crea o nouă arhivă a cărei denumire este *app.jar* iar fişierele incluse de ea sunt cele specificate de lista de fişiere.

Rularea unei aplicaţii împachetate într-o arhivă jar se face incluzând arhiva în argumentul *classpath* al maşinii virtuale Java şi specificând numele complet al clasei ce

include metoda *main*. Totuşi, acest lucru nu e prea convenabil pentru un client. O altă variantă de rulare a aplicaţiei este prezentată mai jos.

```
java -jar app.jar
```

Pentru ca o aplicaţie să poată fi rulată în acest mod, este necesar ca arhiva să conţină un fişier *manifest* care să conţină numele clasei ce conţine metoda *main*:

```
//Fisierul manifest.tmp
//Clasa este numele clasei ce contine metoda main
Main-Class: Clasa
Name: Clasa.class
```

Crearea arhivei ce conţine fişierul *manifest.tmp* se va face prin execuţia comenzii de mai jos:

```
jar cfm app.jar manifest.tmp fisier1 ... fisierN
```

12.3 Exerciţii

1. Eliminaţi duplicarea de cod din porţiunea de cod de mai jos.

```
class Matrice {

    private Integer[][] elemente;

    //Se face o citire a elementelor matricei din fisier
    public Matrice(int n) {
        elemente = new int[n][n];
        FileInputStream file = new FileInputStream("matrice.txt");
        InputStreamReader is = new InputStreamReader(file);
        BufferedReader stream = new BufferedReader(is);
        for (int i=0; i<n; i++)
            for (int j=0; j<n; j++)
                elemente[i][j] = Integer.parseInt(stream.readLine());
        stream.close();
    }

    //Se face o citire a elementelor matricei din fisier
    //daca neinitializat este false, in caz contrar elementele vor fi 0
```

```
    public Matrice(int n, boolean neinitializat) {
        if(neinitializat == true) elemente = new int[n][n];
        else {
            elemente = new int[n][n];
            FileInputStream file = new FileInputStream("matrice.txt");
            InputStreamReader is = new InputStreamReader(file);
            BufferedReader stream = new BufferedReader(is);
            for (int i=0; i<n; i++)
                for (int j=0; j<n; j++)
                    elemente[i][j] = Integer.parseInt(stream.readLine());
            stream.close();
        }
    }
}
```

2. Generaţi documentaţia specifică aplicaţiei dezvoltate în cadrul primului exerciţiu din Lecţia 9.
3. Creaţi o arhivă jar pentru aplicaţia dezvoltată în cadrul primului exerciţiu din Lecţia 9.

Bibliografie

1. Harvey Deitel & Paul Deitel. *Java - How to program.* Prentice Hall, 1999, Appendix G, Creating HTML Documentation with javadoc.
2. David Flanagan, *Java In A Nutshell. A Desktop Quick Reference*, Third Edition, O'Reilly, 1999.
3. Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.
4. Sun Microsystems Inc., *The Java Tutorial, http://java.sun.com/docs/books/tutorial/jar*, 2005.