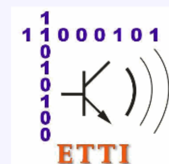
SwRTc ISw

POO – an II

Catedra de Telecomunicatii



22/11/2010

Limbaje de Programare pentru Aplicatii Internet (LPAI)

Laborator 4

Fire de executie. Socketuri Java. Interfete grafice Swing (2)

4.1. Descrierea laboratorului

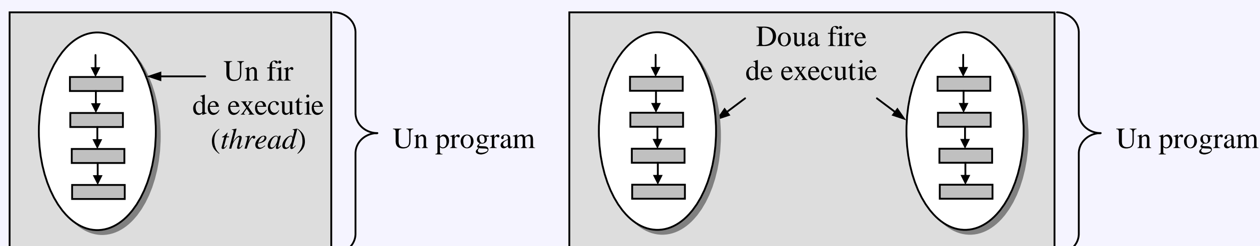
In aceasta lucrare de laborator vor fi acoperite urmatoarele probleme:

- Lucrul cu fire de executie – programe multifilare
- Lucrul cu socket-uri flux (TCP) – (vezi si Lab. an V)
- Programe de lucru cu socket-uri - clienti si servere TCP
  – vezi si Lab.4 An II, Lab.5 An II, Lab.6 An II)
- Teme de casa
- Anexe

4.2. Utilizarea firelor de executie (threads) in Java

4.2.1. Modalitati de creare a unui fir de executie

Programele realizate in laboratoarele anterioare sunt programe de calcul simple, secventiale, fiecare avand un inceput, o secventa de executii si un sfarsit, iar in orice moment pe durata rularii lor existand un singur punct de executie. Un **fir de executie** (***thread***) este similar unui program secvential, avand inceput, secventa de executii si sfarsit, iar in orice moment al rularii lui existand un singur punct de executie.



Totusi, un fir nu este el insusi un program, deoarece nu poate fi executat de sine statator. In schimb, firul este executat (rulat) intr-un program, in paralel cu alte fire de executie. Cu alte cuvinte, programele pot executa (sau par ca executa) mai multe operatii diferite in acelasi timp. In general, firele de executie, partajeaza de fapt aceleasi resurse ale calculatorului (o aceiasi zona de memorie unde sunt pastrate datele programului; utilizeaza acelasi procesor sau aceleasi dispozitive de intrare/iesire) si in consecinta, nu pot fi executate in paralel. Insa Java poate crea iluzia ca firele sunt executate simultan comutand rapid controlul de la un fir la altul (fiecare fir executandu-se pentru scurt timp inainte de a pasa controlul spre un alt fir). Posibilitatea utilizarii mai multor fire de executie intr-un singur program este numita *multithreading*.

Limbajul Java permite crearea firelor de executie prin doua metode:

1. Prin **declararea unei clase care extinde clasa *Thread***. Aceasta metoda presupune parcurgerea urmatorilor pasi:

- se creaza o clasa derivata din clasa *Thread:*

```
class FirT extends Thread{ ... }
```

- in interiorul noi clase se suprascrie metoda **public void run()** mostenita din clasa *Thread:*

```
public void run() {  ... /*codul firului de executie*/ }
```

- se instantiaza un obiect al noii clase folosind operatorul **new**:

```
FirT fir = new FirT();
```

- se porneste thread-ul instantiat, prin apelul metodei **start()** mostenita din clasa *Thread:*

```
fir.start();
```

2. Prin **declararea unei clase care implementeaza interfata Runnable** (java.lang.Runnable). (Nota1: definire interfete). Interfata *Runnable* contine doar declaratia unei metode **run()** care are acelasi rol cu cea din clasa *Thread*. Necesitatea implementarii interfetei *Runnable* apare atunci cand se doreste crearea unei clase de fire de executie care nu extinde clasa *Thread*. Motivul ar putea fi acela ca noua clasa trebuie sa extinda o alta clasa, iar in Java mostenirea multipla nu este posibila. Aceasta metoda presupune parcurgerea urmatorilor pasi:

- se creeaza o clasa care implementeaza interfata *Runnable:*

```
class FirR implements Runnable { ... }
```

- se implemeteaza metoda **run()** din interfata *Runnable:*

```
public void run() {  ... /*codul firului de executie*/ }
```

- se instantiaza un obiect al noii clase folosind operatorul **new**:

```
FirR r = new FirR();
```

- se creeaza un obiect din clasa *Thread* folosind un constructor care are ca parametru un obiect al clasei ce implementează interfata *Runnable*:

```
Thread fir = new Thread(r);
```

- se porneste thread-ul instantiat anterior, prin apelul metodei **start()** din clasa *Thread:*

```
fir.start();
```

4.2.2. Programe de lucru cu fire de executie

Urmatorul program este folosit pentru a ilustra lucrul cu fire de executie. Clasa **`FirSimplu`** extinde clasa **`Thread`**, iar metoda principala lanseaza metoda **`run`**`()` ca nou fir de executie.

In laborator:

1. Se deschide mediul de programare NetBeansIDE 6.1 si se sterg (click dreapta -> Delete) toate proiectele create anterior;

2. Se creaza un nou proiect (*New Project*) cu numele **FireExecutie** (de tipul Java / Java Class Library);

3. Proiectul se va salva intr-ul folder de tipul: “D:\LPAI\Laborator4\GrupaXYZ”;

4. Se adauga un nou fisier .java (click dreapta pe numele proiectului -> New -> Java Class...) cu numele **FirSimplu**;

5. Se implementeaza urmatoarea secventa de cod care permite lansarea unui singur fir de executie.

```
1  public class FirSimplu extends Thread {
2                               //obiectele din clasa curenta sunt thread-uri
3     public FirSimplu(String str) {
4        super(str);            // invocarea constructorului Thread(String)
5                               // al superclasei Thread
6     }
7     public void run() {       // "metoda principala" a thread-ului curent
8        for (int i = 0; i < 5; i++) {
9           System.out.println(i + " " + getName());
10                              // obtinerea numelui thread-ului
11          try {
12             sleep((long)(Math.random() * 1000));
13                              // thread-ul "doarme" 0...1 secunda
14          } catch (InterruptedException e) {}
15       }
16       System.out.println("Gata! " + getName());
17                              // obtinerea numelui thread-ului
18    }
19    public static void main (String[] args) {
20       new FirSimplu("Unu").start();
21                              // "lansarea" thread-ului (apeleaza run())
22    }
23 }
```

In urma executiei programului se obtine urmatorul rezultat:

```
Output - Lab4Socket (run-single) #2
init:
deps-jar:
Compiling 1 source file to D:\Java\Lab4\
compile-single:
run-single:
0 Unu
1 Unu
2 Unu
3 Unu
4 Unu
Gata! Unu
BUILD SUCCESSFUL (total time: 2 seconds)
```

Metoda **run()** este cea care contine programul propriu-zis care trebuie executat de un fir de executie. Totusi, aceasta metoda nu este invocata explicit. Ea este invocata de masina virtuala Java, atunci cand firul respectiv este pus in mod efectiv in executie. Prin apelarea metodei start(), firul va fi pregatit de executie, dar nu neaparat va si rula imediat. El devine *concurent* cu alte fire deja existente (in primul rand cu cel care l-a creat), si va fi pus in executie in mod efectiv (i se va acorda acces la procesor) atunci cand ii vine randul, conform cu strategia de alocare a resurselor adoptata.

Un fir in curs de executie poate fi oprit pentru o perioada de timp invocand metoda `sleep(long millis)`. Argumentul **millis** reprezinta timpul cat firul de executie va fi "adormit" (se va opri din executie), exprimat in milisecunde.

Clasa `DemoDouaFire` lanseaza doua fire de executie de tip `FirSimplu` executate **concurent**.

```
1 public class DemoDouaFire {
2    public static void main (String[] args) {
3       new FirSimplu("Unu").start();   // "lansarea" primului thread
4       new FirSimplu("Doi").start();   // "lansarea" celui de-al doilea thread
5    }
6 }
```

Firele au evolutii diferite, in functie de durata intarzierii introdusa in linia de cod:

```
sleep((long)(Math.random() * 1000));
```

a programului `FirSimplu`.

Rezultatele a doua executii succesive:

```
0 Unu
0 Doi
1 Doi
1 Unu
2 Doi
2 Unu
3 Doi
4 Doi
3 Unu
4 Unu
Gata! Unu
Gata! Doi
```

```
0 Unu
0 Doi
1 Unu
1 Doi
2 Unu
3 Unu
2 Doi
4 Unu
3 Doi
Gata! Unu
4 Doi
Gata! Doi
```

In laborator:

1. Se vor modifica cele doua clase definite anterior (/ crea altele noi) astfel incat sa se foloseasca cea de-a doua metoda de creare a firelor de executie (prin implementarea interfetelor Runnable).

Hint! Deoarece clasa **FirSimplu** nu mai mosteneste clasa `Thread`, metodele `getName()` si `sleep()` nu mai pot fi accesate direct. Apelarea acestora se va face astfel: `Thread.currentThread().getName()` si `Thread.currentThread().sleep(...);`

4.3. Introducere in socket-uri Java

Pentru realizarea unor programe care sa comunice in retelele bazate pe IP (Internet Protocol), programul Java pune la dispozitie o serie de clase utile, disponibile in pachetul `java.net`. Comunicare in Internet intre doua terminale, respectind modelul client-server presupune stabilirea unei conexiuni ce se bazeaza pe adresa IP a serverului si numarul portului pe care este deschisa aplicatia (valori intre 0 si 65535). Combinaţia, adresa IP si numarul portului este folosită pentru crearea unui termen abstract, numit socket (canal de comunicatie). Un socket permite crearea de fluxuri de intrare/iesire utile pentru schimbul de date intre client si server. La stabilirea unei conexiuni, atat clientul cat si serverul vor avea asociate cate un socket, comunicarea efectiva realizandu-se intre socketuri.

Principalele functii indeplinite de socketuri sunt:
- conectarea la un socket corespondent;
- acceptarea conexiunilor;
- trimiterea de date
- recepţionarea datelor
- inchiderea conexiunii cu socket-ul corespondent;

4.3.1. Socket-uri flux (TCP) Java

Java ofera, in pachetul `java.net`, mai multe clase pentru lucrul cu *socket-uri* flux (TCP). Urmatoarele clase Java sunt implicate in realizarea conexiunilor TCP obisnuite: `ServerSocket` si `Socket`.

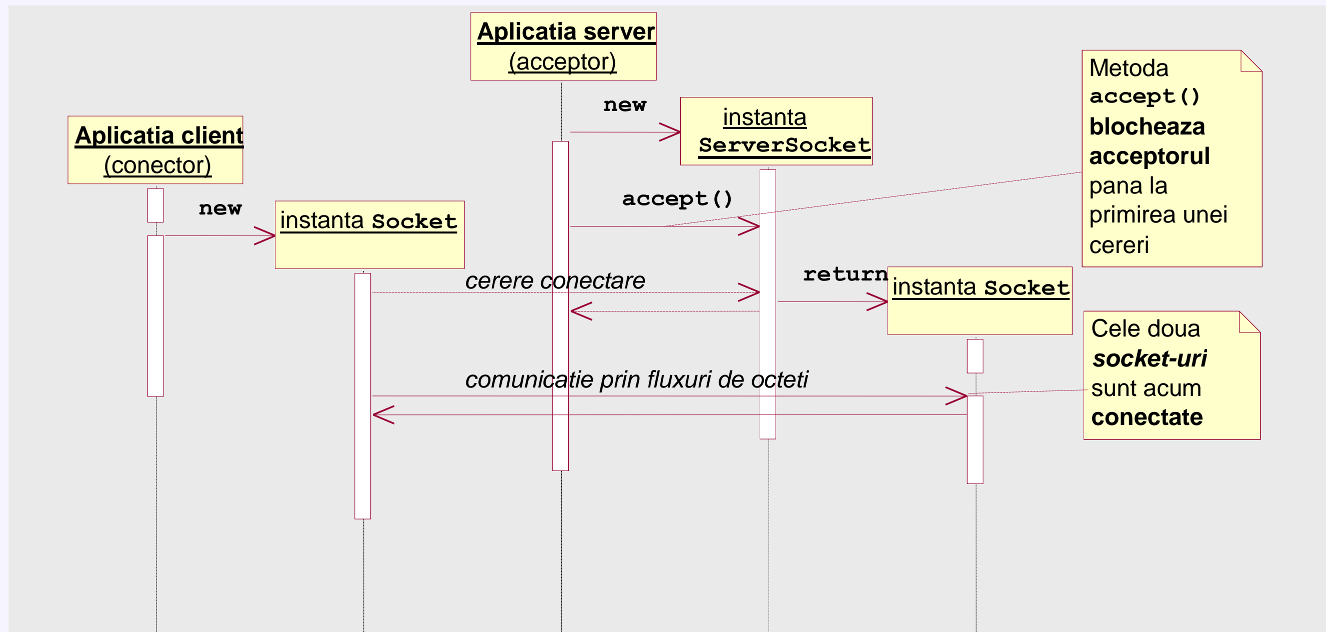
Clasa ServerSocket reprezinta socket-ul aflat pe un server (bazat pe TCP) care asteapta si accepta cereri de conexiune provenite din partea unui client (bazat eventual tot pe TCP).
Clasa Socket reprezinta punctul terminal al unei conexiuni TCP intre doua masini (eventual un client si un server).

Clientul (sau, mai general, **masina conector**) **creeaza un punct terminal** `Socket` in momentul in care cererea sa de conexiune este lansata si acceptata.
Serverul (sau, mai general, **masina acceptor**) **creeaza un** `Socket` in momentul in care primeste si accepta o cerere de conexiune. Acesta continua apoi sa asculte si sa astepte alte cereri pe `ServerSocket`.

Dintre metodele clasei ServerSocket, cele mai importante sint **accept()** si **close()**. Metoda **accept()** este folosita pentru crearea unui socket si atasarea acestuia la un port al serverului, iar metoda **close()**

este utilizata pentru a inchide o conexiune care s-a terminat. ServerSocket va returna clientului prin metoda accept(), socket-ul deschis pe server pentru realizarea comunicatiei. Este stabilita astfel o conexiune Socket la Socket.

Secventa tipica a mesajelor schimbate intre client si server este urmatoarea:



Odata conexiunea stabilita, metodele **`getInputStream()`** si **`getOutputStream()`** ale clasei `Socket` pot fi utilizate (pentru fiecare socket in parte) pentru a obtine fluxuri de octeti, de intrare respectiv iesire, pentru comunicatia intre aplicatii.

4.3.2. Utilizarea clasei java.net.InetAddress

Java ofera, in pachetul **`java.net`**, clasa **`InetAddress`** care poate furniza informatii despre adresa (simbolică şi numerică) unui calculator gazda. Clasa **`InetAddress`** nu are constructori publici. De aceea, pentru a crea obiecte ale acestei clase trebuie invocata una dintre metodele de clasa **`getByAddress()`** sau **`getByName()`**.

O adresa IP speciala este **adresa IP *loopback*** (tot ce este trimis catre aceasta adresa IP se intoarce si devine intrare IP pentru gazda locala), cu ajutorul careia pot fi testate local programe care utilizeaza *socket-uri*. Pentru a identifica adresa IP *loopback* sunt folosite numele **`"localhost"`** si valoarea numerica **`"127.0.0.1"`**.

Pentru a obtine **`InetAddress`** care incapsuleaza adresa IP *loopback* pot fi folosite urmatoarele apelurile:

```
InetAddress.getByName(null)
InetAddress.getByName("localhost")
InetAddress.getByName("127.0.0.1")
```

4.3.3. Utilizarea clasei Socket

Secventa tipica pentru **crearea *socket-ului* unei aplicatii conector (client)**:

```
1  // Stabilirea adresei serverului
2  String adresaServer = "localhost";
3  
4  // Stabilirea portului serverului
5  int portServer = 2000;
6  
7  // Crearea socketului (implicit este realizata conexiunea cu serverul)
8  Socket socketTCPClient = new Socket(adresaServer, portServer);
```

Dupa utilizare, *socket-ul* este inchis. Secventa tipica pentru **inchiderea** ***socket-ului***:

```
1  // Inchiderea socketului (implicit a fluxurilor TCP)
2  socketTCPClient.close();
```

4.3.4. Utilizarea clasei ServerSocket

Secventa tipica pentru **crearea** ***ServerSocket-ului*** **pentru o aplicatie acceptor (server)**:

```
1  // Stabilirea portului serverului
2  int portServer = 2000;
3
4  // Crearea socketului server (care accepta conexiunile)
5  ServerSocket serverTCP = new ServerSocket(portServer);
```

Secventa tipica pentru **crearea** ***socket-ului*** **pentru tratarea conexiunii TCP cu un client**:

```
1  // Blocare in asteptarea cererii de conexiune - in momentul acceptarii
2  // cererii se creaza socketul care serveste conexiunea
3
4  Socket conexiuneTCP = serverTCP.accept();
```

Secventa tipica pentru **crearea fluxurilor de octeti asociate** ***socket-ului*** (*detalii fluxuri IO*):

```
1  // Obtinerea fluxului de intrare octeti TCP
2  InputStream inTCP = socketTCPClient.getInputStream();
3
4  // Obtinerea fluxului scanner de caractere dinspre retea
5  Scanner scanTCP = new Scanner(inTCP);
6
7  // Obtinerea fluxului de iesire octeti TCP
8  OutputStream outTCP = socketTCPClient.getOutputStream();
9
10 // Obtinerea fluxului de iesire spre retea (similar consolei de iesire)
11 PrintStream printTCP  = new PrintStream(outTCP);
```

Secventa tipica pentru **trimiterea de date**:

```
1  // Crearea unui mesaj
2  String mesajDeTrimis = "Continut mesaj";
3
4  // Scrierea catre retea (trimiterea mesajului) si fortarea trimiterii
5  printTCP.println(mesajDeTrimis);
6  printTCP.flush();
```

Secventa tipica pentru **primirea de date**:

```
1  // Citirea dinspre retea (receptia unui mesaj)
2  String mesajPrimit = scanTCP.nextLine();
3
4  // Afisarea mesajului primit
5  System.out.println(mesajPrimit);
```

4.4. Programe de lucru cu socket-uri – clienti si servere

4.4.1. Server unifilar (care poate trata un singur client)

Clasa urmatoare ofera un "serviciu" accesibil la distanta prin intermediul unui server TCP unifilar (care poate trata un singur client o data):

In laborator:

1. Se creaza un nou proiect (*New Project*) cu numele `ClientServerTCP` (de tipul Java / Java Class Library);

2. Proiectul se va salva intr-ul folder de tipul: “D:\LPAI\Laborator4\GrupaXYZ”;

3. Se adauga un nou fisier .java (click dreapta pe numele proiectului -> New -> Java Class...) cu numele `Orar`;

4. Se implementeaza urmatoarea secventa de cod.

```
 1 public class Orar {
 2
 3     private String[] orar; // camp ascuns (starea obiectului)
 4
 5     public Orar() {
 6         orar = new String[7]; // alocarea dinamica a spatiului pentru tablou
 7         // popularea tabloului cu valori
 8         orar[0] = "Luni nu sunt ore de LPAI.";
 9         orar[1] = "Marti sunt laboratoare de LPAI.";
10         orar[2] = "Miercuri nu sunt ore de LPAI.";
11         orar[3] = "Joi sunt proiecte de LPAI.";
12         orar[4] = "Vineri este curs de LPAI.";
13         orar[5] = "Sambata nu sunt ore de LPAI.";
14         orar[6] = "Duminica nu sunt ore de LPAI.";
15     }
16     public String getOrar(int zi) { // metoda accesor - getter
17         return orar[zi];            // returneaza referinta la tablou
18     }
19     public void setOrar(int zi, String text) { // metoda accesor - setter
20         orar[zi] = text;                       // inlocuieste un element
21     }
22 }
```

Clasa server care urmeaza permite accesul la "serviciile" oferit de clasa de mai sus, prin crearea unui obiect din aceasta clasa si apelul metodelor accesor:

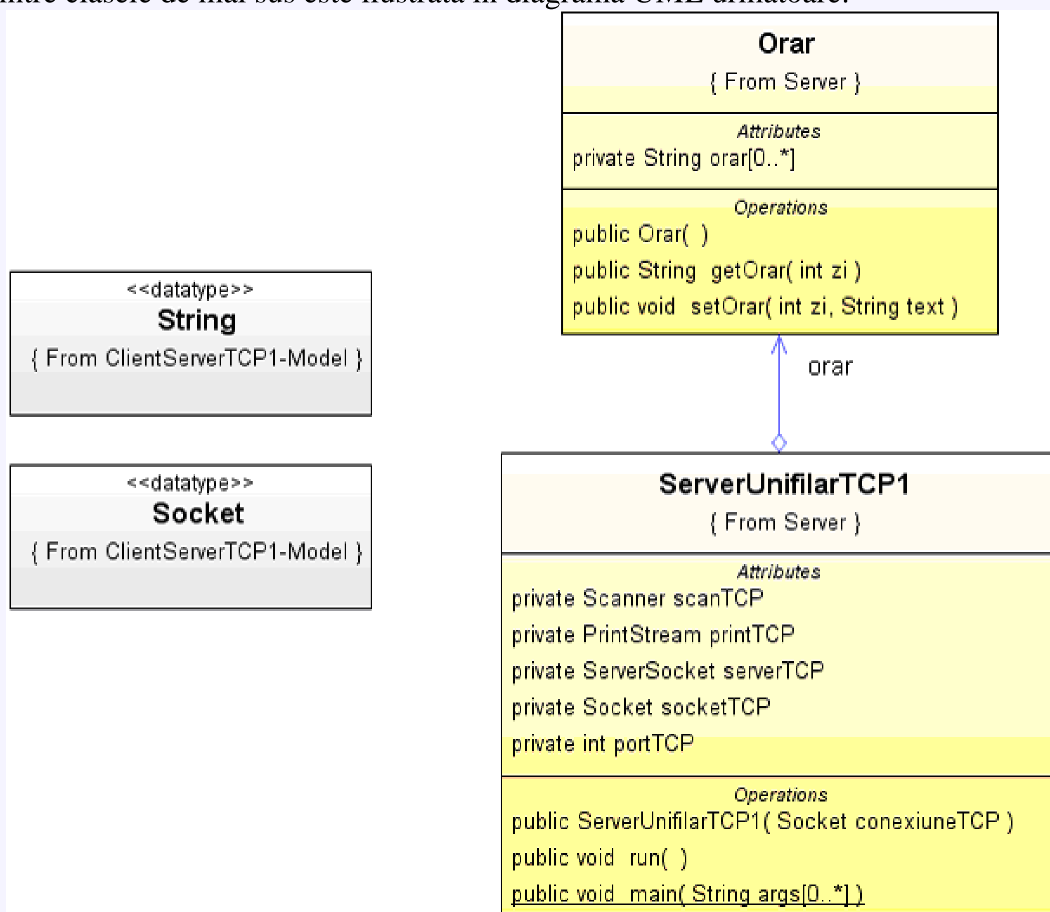
In laborator:

1. In proiectul `ClientServerTCP` se adauga un nou fisier .java cu numele `ServerUnifilarTCP1`.
2. Se implementeaza urmatoarea secventa de cod.

```
 1 import java.net.*;
 2 import java.io.*;
 3 import java.util.Scanner;
 4 import javax.swing.JOptionPane;
 5
 6 public class ServerUnifilarTCP1 {
 7     private Scanner scanTCP;
 8     private PrintStream printTCP;
 9     private ServerSocket serverTCP;
10     private Socket socketTCP;
11     private int portTCP;
12     private Orar orar = new Orar();
13
14     public ServerUnifilarTCP1(Socket conexiuneTCP) throws IOException {
15         this.socketTCP = conexiuneTCP;        // Obtinere socket
16         this.scanTCP = new Scanner(socketTCP.getInputStream());
17         this.printTCP = new PrintStream(socketTCP.getOutputStream());
18     }
19
20     public void run() {
21         String mesaj;
22         int zi;
23         try {
24
25             while(true) {
26                 if (scanTCP.hasNextLine()) {
27                     mesaj = scanTCP.nextLine();
28                     if (mesaj.equals("getOrar")) {
29                         try {
30                             zi = Integer.parseInt(scanTCP.nextLine());
31                             JOptionPane.showMessageDialog(null,
32                                   "Server: am primit " + mesaj + " si " + zi);
33                             String rezultat = orar.getOrar(zi);
34                             printTCP.println(rezultat);
35                             printTCP.flush();
36                         } catch (NumberFormatException ex) {
37                             printTCP.println("Stop");
38                             printTCP.flush();
39                             break;
40                         }
41                     }
```

```
42                      /*
43                      else if (mesaj.equals("setOrar")) {
44
45                      }
46                      */
47                      else {
48                          printTCP.println("Stop");
49                          printTCP.flush();
50                          break;
51                      }
52                  }
53              }
54              socketTCP.close();    // Inchiderea socketului si a fluxurilor
55              JOptionPane.showMessageDialog(null, "Server: Bye!");
56          } catch (IOException ex) {
57              ex.printStackTrace();
58          }
59      }
60
61      public static void main(String[] args) throws IOException {
62          int portTCP = Integer.parseInt(JOptionPane.showInputDialog(
63                        "Server: introduceti numarul de port al serverului"));
64
65          // Creare socket server
66          ServerSocket serverTCP = new ServerSocket(portTCP);
67          Socket conexiune = serverTCP.accept(); // Obtinere socket
68          ServerUnifilarTCP1 server = new ServerUnifilarTCP1(conexiune);
69          server.run();
70      }
71  }
```

Relatia intre clasele de mai sus este ilustrata in diagrama UML urmatoare:



4.4.2. Clienti pentru serverul unifilar

Clasa client care urmeaza (`ClientTCP1`) permite utilizatorilor accesul la distanta la "serviciul" aflat pe aceeasi masina cu serverul TCP anterior. Utilizatorul poate accesa (`getOrar`), prin intermediul acestui client, informatiile stocate in obiectul din clasa `Orar` de pe server.

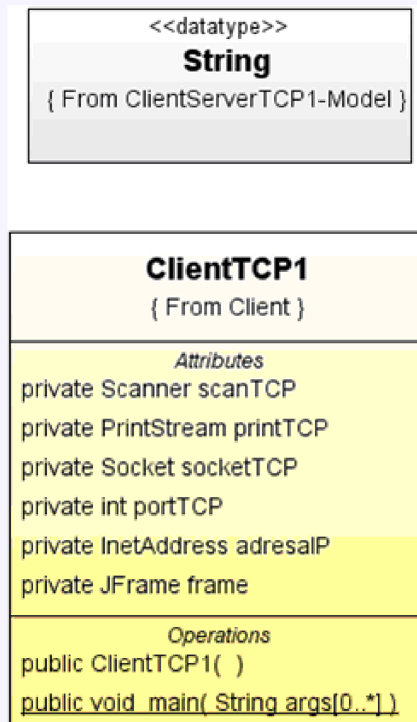
In laborator:

1. In proiectul **ClientServerTCP** se adauga un nou fisier .java cu numele **ClientTCP1**.
2. Se implementeaza urmatoarea secventa de cod.

```
 1  import java.awt.*;
 2  import java.awt.event.*;
 3  import javax.swing.*;
 4  import java.net.*;
 5  import java.io.*;
 6  import java.util.*;
 7
 8  public class ClientTCP1 {
 9      private Scanner scanTCP;
10      private PrintStream printTCP;
11      private Socket socketTCP;
12      private int portTCP;
13      private InetAddress adresaIP;
14      private JFrame frame;
15
16      public ClientTCP1() throws IOException {
17          this.portTCP = Integer.parseInt(JOptionPane.showInputDialog(
18                         "Client: introduceti numarul de port al serverului"));
19          this.adresaIP = InetAddress.getByName(JOptionPane.showInputDialog(
20                         "Client: introduceti adresa serverului"));
21          this.socketTCP = new Socket(adresaIP, portTCP);     // Creare socket
22          this.scanTCP = new Scanner(socketTCP.getInputStream());
23          this.printTCP = new PrintStream(socketTCP.getOutputStream());
24
25          frame = new JFrame("Client TCP");
26          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27          Container containerCurent = frame.getContentPane();
28          containerCurent.setLayout(new BorderLayout());
29
30          JPanel pane = new JPanel(new GridLayout(0, 1));
31          final JLabel eticheta = new JLabel("Orarul zilei de:");
32          pane.add(eticheta);
33
34          final int numButtons = 8;
35          final JRadioButton[] radioButtons = new JRadioButton[numButtons];
36          final ButtonGroup group = new ButtonGroup();
37
38          radioButtons[0] = new JRadioButton("Luni");
39          radioButtons[0].setActionCommand("0");
40          radioButtons[1] = new JRadioButton("Marti");
41          radioButtons[1].setActionCommand("1");
42          radioButtons[2] = new JRadioButton("Miercuri");
43          radioButtons[2].setActionCommand("2");
44          radioButtons[3] = new JRadioButton("Joi");
45          radioButtons[3].setActionCommand("3");
46          radioButtons[4] = new JRadioButton("Vineri");
47          radioButtons[4].setActionCommand("4");
48          radioButtons[5] = new JRadioButton("Sambata");
49          radioButtons[5].setActionCommand("5");
50          radioButtons[6] = new JRadioButton("Duminica");
51          radioButtons[6].setActionCommand("6");
52          radioButtons[7] = new JRadioButton("Stop");
53          radioButtons[7].setActionCommand("Stop");
54          radioButtons[0].setSelected(true);
55
56          for (int i = 0; i < numButtons; i++) {
57              group.add(radioButtons[i]);
58              pane.add(radioButtons[i]);
59          }
60          containerCurent.add(pane, BorderLayout.WEST);
61
62          JButton sendButton = new JButton("Trimite");
63
64          sendButton.addActionListener(new ActionListener() {
65
66              public void actionPerformed(ActionEvent e) {
67                  String service = "getOrar";
```

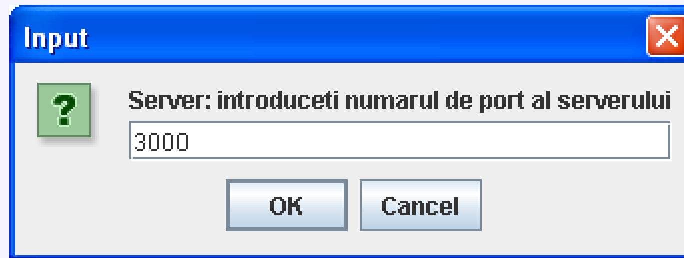
```
68                  printTCP.println(service);
69                  printTCP.flush();
70                  String command = group.getSelection().getActionCommand();
71                  printTCP.println(command);
72                  printTCP.flush();
73              }
74          });
75          containerCurent.add(sendButton, BorderLayout.NORTH);
76
77          final JTextArea outGrafic = new JTextArea(8,40); //Zona non-editabila
78          JScrollPane scrollPane = new JScrollPane(outGrafic,
79                      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
80                      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
81          outGrafic.setEditable(false);
82          containerCurent.add(outGrafic, BorderLayout.CENTER);
83
84          frame.pack();
85          frame.setVisible(true);
86
87          String mesaj;
88          while(true) {
89              if (scanTCP.hasNextLine()) {
90                  mesaj = scanTCP.nextLine();
91                  outGrafic.setText(outGrafic.getText() + mesaj + "\n");
92                  if (mesaj.equals("Stop")) System.exit(0);  // Conditie oprire
93              }
94          }
95      }
96      public static void main(String[] args) throws IOException {
97          ClientTCP1 client = new ClientTCP1();
        }
}
```

Urmatoarea diagrama UML reflecta codul de mai sus:



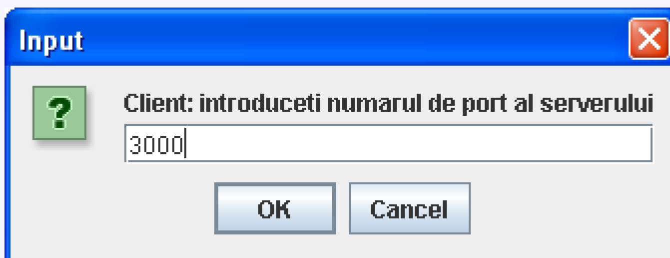
In laborator:

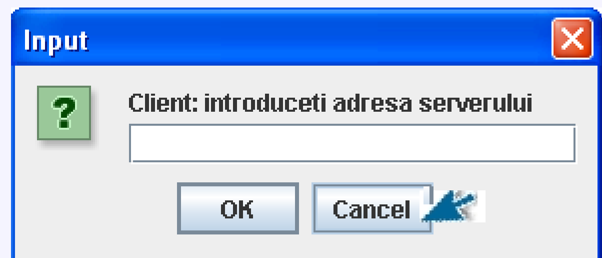
1. Se selecteaza **Run File** pe nodul clasei cu numele **`ServerUnifilarTCP1`**.
2. Se introduce valoarea **3000** pentru portul pe care asculta serverul TCP.



In laborator:

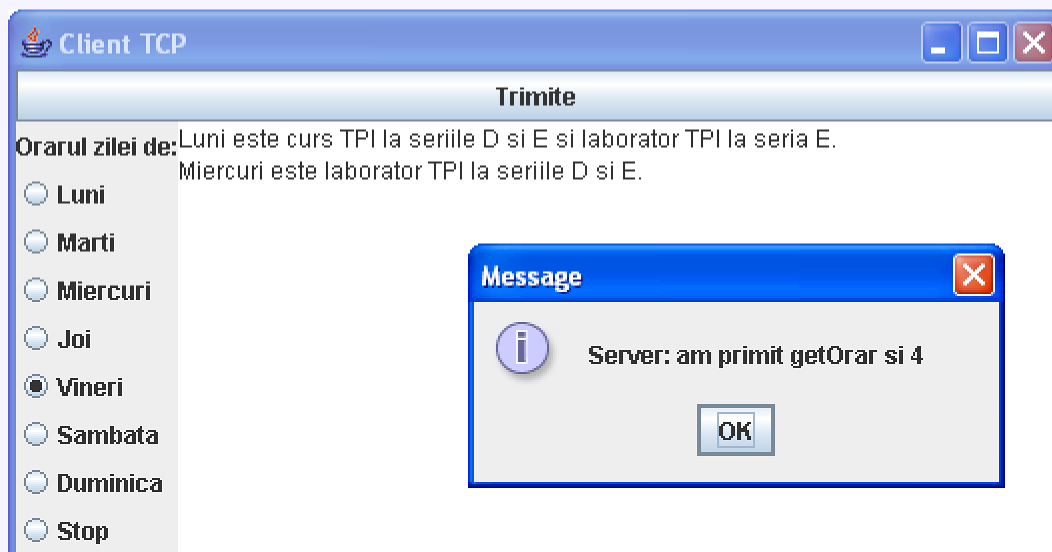
1. Se selecteaza **Run File** pe nodul clasei cu numele `ClientTCP1`.
2. Se introduce valoarea **3000** pentru portul serverului si adresa IP a calculatorului pe care a fost lansat serverul, sau se apasa **Cancel** (generandu-se astfel adresa **loopback "localhost" – 127.0.0.1**).





In laborator:

1. Se utilizeaza interfata clientului.
2. Pentru oprire se selecteaza **Stop** si se apasa **Trimite**.



Clasa client care urmeaza (`ClientTCP2`) permite de asemenea accesul la distanta la "serviciul" aflat pe aceeasi masina cu serverul TCP anterior. Utilizatorul poate modifica (`setOrar`), prin intermediul acestui client, informatiile stocate in obiectul din clasa `Orar` de pe server.

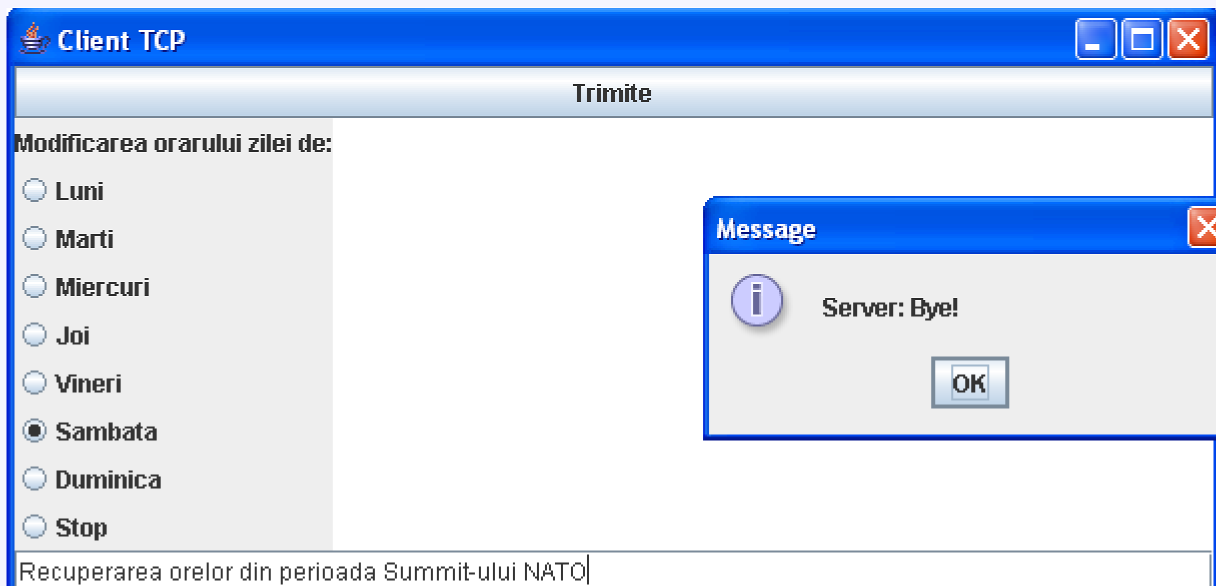
```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  import java.net.*;
5  import java.io.*;
6  import java.util.*;
7
8  public class ClientTCP2 {
9      private Scanner scanTCP;
10     private PrintStream printTCP;
11     private Socket socketTCP;
12     private int portTCP;
13     private InetAddress adresaIP;
14     private JFrame frame;
15
16     public ClientTCP2() throws IOException {
17         this.portTCP = Integer.parseInt(JOptionPane.showInputDialog(
18                        "Client: introduceti numarul de port al serverului"));
19         this.adresaIP = InetAddress.getByName(JOptionPane.showInputDialog(
20                        "Client: introduceti adresa serverului"));
21         this.socketTCP = new Socket(adresaIP, portTCP);     // Creare socket
22         this.scanTCP = new Scanner(socketTCP.getInputStream());
23         this.printTCP = new PrintStream(socketTCP.getOutputStream());
24
25         JFrame frame = new JFrame("Client TCP");
26         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27         Container containerCurent = frame.getContentPane();
28         containerCurent.setLayout(new BorderLayout());
29
30         JPanel pane = new JPanel(new GridLayout(0, 1));
31         final JLabel eticheta = new JLabel("Modificarea orarului zilei de:");
32         pane.add(eticheta);
33
34         final int numButtons = 8;
35         final JRadioButton[] radioButtons = new JRadioButton[numButtons];
36         final ButtonGroup group = new ButtonGroup();
37
38         radioButtons[0] = new JRadioButton("Luni");
39         radioButtons[0].setActionCommand("0");
40         radioButtons[1] = new JRadioButton("Marti");
41         radioButtons[1].setActionCommand("1");
42         radioButtons[2] = new JRadioButton("Miercuri");
43         radioButtons[2].setActionCommand("2");
44         radioButtons[3] = new JRadioButton("Joi");
45         radioButtons[3].setActionCommand("3");
46         radioButtons[4] = new JRadioButton("Vineri");
47         radioButtons[4].setActionCommand("4");
48         radioButtons[5] = new JRadioButton("Sambata");
49         radioButtons[5].setActionCommand("5");
50         radioButtons[6] = new JRadioButton("Duminica");
51         radioButtons[6].setActionCommand("6");
52         radioButtons[7] = new JRadioButton("Stop");
53         radioButtons[7].setActionCommand("Stop");
54         radioButtons[0].setSelected(true);
55
56         for (int i = 0; i < numButtons; i++) {
57             group.add(radioButtons[i]);
58             pane.add(radioButtons[i]);
59         }
60         containerCurent.add(pane, BorderLayout.WEST);
61
62         final JTextField inTextGrafic = new JTextField(40); //Camp editabil intrare
63         containerCurent.add(inTextGrafic, BorderLayout.SOUTH);
64         JButton sendButton = new JButton("Trimite");
65
66         sendButton.addActionListener(new ActionListener() {
67
68             public void actionPerformed(ActionEvent e) {
69                 String service = "setOrar";
70                 printTCP.println(service);
71                 printTCP.flush();
72                 String command = group.getSelection().getActionCommand();
73                 printTCP.println(command);
74                 printTCP.flush();
```

```
75                      printTCP.println(inTextGrafic.getText());
76                      printTCP.flush();
77                      inTextGrafic.setText("");
78                  }
79              });
80              containerCurent.add(sendButton, BorderLayout.NORTH);
81
82              final JTextArea outGrafic = new JTextArea(8, 40); // Zona non-editabila
83              JScrollPane scrollPane = new JScrollPane(outGrafic,
84                          JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
85                          JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
86              outGrafic.setEditable(false);
87              containerCurent.add(outGrafic, BorderLayout.CENTER);
88
89              inTextGrafic.requestFocus(); // Cerere focus pe intrarea de text
90
91              frame.pack();
92              frame.setVisible(true);
93
94              String mesaj;
95              while(true) {
96                  if (scanTCP.hasNextLine()) {
97                      mesaj = scanTCP.nextLine();
98                      outGrafic.setText(outGrafic.getText() + mesaj + "\n");
99                      if (mesaj.equals("Stop")) System.exit(0);  // Conditie oprire
00                  }
01              }
02          }
03
04          public static void main(String[] args) throws IOException {
05              ClientTCP2 client = new ClientTCP2();
06          }
07      }
08
09
```

In laborator:

1. Se selecteaza **Run File** pe nodul clasei cu numele `ServerUnifilarTCP1`.
2. Se introduce valoarea **4000** pentru portul pe care asculta serverul TCP.
3. . Se selecteaza **Run File** pe nodul clasei cu numele `ClientTCP2`.
4. Se introduce valoarea **4000** pentru portul serverului si se introduce adresa IP a calculatorului pe care a fost lansat serverul, sau se apasa **Cancel** (generandu-se astfel adresa “localhost”).
5. Se utilizeaza interfata clientului pentru modificarea orarului.

Atentie! Se poate observa ca de fiecare data cand se incearca modificarea unei zile din orar si se apasa butonul **Trimite**, serverul ne afiseaza mesajul: “Server: Bye!” si se inchide.



In laborator:

1. Sa se modifice/completeze codul clasei **ServerUnifilarTCP1** pentru a putea oferi si serviciul de modificare a orarului (**setOrar**);

2. Dupa ce serverul face modificarea efectiva in orar, sa trimita clientului un mesaj de confirmare, pe baza caruia clientul sa afiseze in obiectul **JTextArea outGrafic** modificarea facuta;

2. Se reiau pasii necesari executiei **ServerUnifilarTCP1** si **ClientTCP2**;

3. Se utilizeaza interfata clientului. Pentru oprire se selecteaza **Stop** si se apasa **Trimite**.

4.4.3. Server multifilar (care poate trata mai multi clienti in paralel)

Clasa server care urmeaza permite accesul mai multor clienti in acelasi timp la "serviciile" oferite de clasa **Orar** (prin implementarea firelor de executie). Clientii pot fi din ambele categorii anterioare, permitand astfel modificarea continutului orarului in timp ce acesta este vizualizat.

```
 1  import java.net.*;
 2  import java.io.*;
 3  import java.util.Scanner;
 4  import javax.swing.JOptionPane;
 5
 6  public class ServerMultifilarTCP1 extends Thread {
 7     private Scanner scanTCP;
 8     private PrintStream printTCP;
 9     private Socket socketTCP;
10     private static Orar orar = new Orar();
11
12     public ServerMultifilarTCP1(Socket conexiuneTCP) throws IOException {
13        this.socketTCP = conexiuneTCP;         // Obtinere socket
14        this.scanTCP = new Scanner(socketTCP.getInputStream());
15        this.printTCP = new PrintStream(socketTCP.getOutputStream());
16     }
17
18     public void run() {
19        String mesaj;
20        int zi;
21        try {
22           while(true) {
23              if (scanTCP.hasNextLine()) {
24                 mesaj = scanTCP.nextLine();
25                 if (mesaj.equals("getOrar")) {
26                    try {
27                       zi = Integer.parseInt(scanTCP.nextLine());
28                       JOptionPane.showMessageDialog(null,"Server:am primit"
29                                                        + mesaj + " si " + zi);
30                       String rezultat = orar.getOrar(zi);
31                       printTCP.println(rezultat);
32                       printTCP.flush();
33                       }
34                       catch (NumberFormatException ex) {
35                          printTCP.println("Stop");
36                          printTCP.flush();
37                          break;
38                       }
39                 }
40                 /*
41                 else if (mesaj.equals("setOrar")) {
42
43                 }
44                 */
45                 else {
46                    printTCP.println("Stop");
47                    printTCP.flush();
48                    break;
49                 }
50              }
51           }
52           socketTCP.close();     // Inchiderea socketului si a fluxurilor
53           JOptionPane.showMessageDialog(null, "Server: Bye!");
54        }
```

```
55         catch (IOException ex) {
56            ex.printStackTrace();
57         }
58      }
59
60      public static void main(String[] args) throws IOException {
61         int portTCP = Integer.parseInt(JOptionPane.showInputDialog(
62                          "Server: introduceti numarul de port al serverului"));
63         ServerSocket serverTCP=new ServerSocket(portTCP);//Creare socket server
64
65         while (true) {
66            Socket conexiune = serverTCP.accept();
67            ServerMultifilarTCP1 server = new ServerMultifilarTCP1(conexiune);
68            server.start();
69         }
      }
   }
```

In laborator:

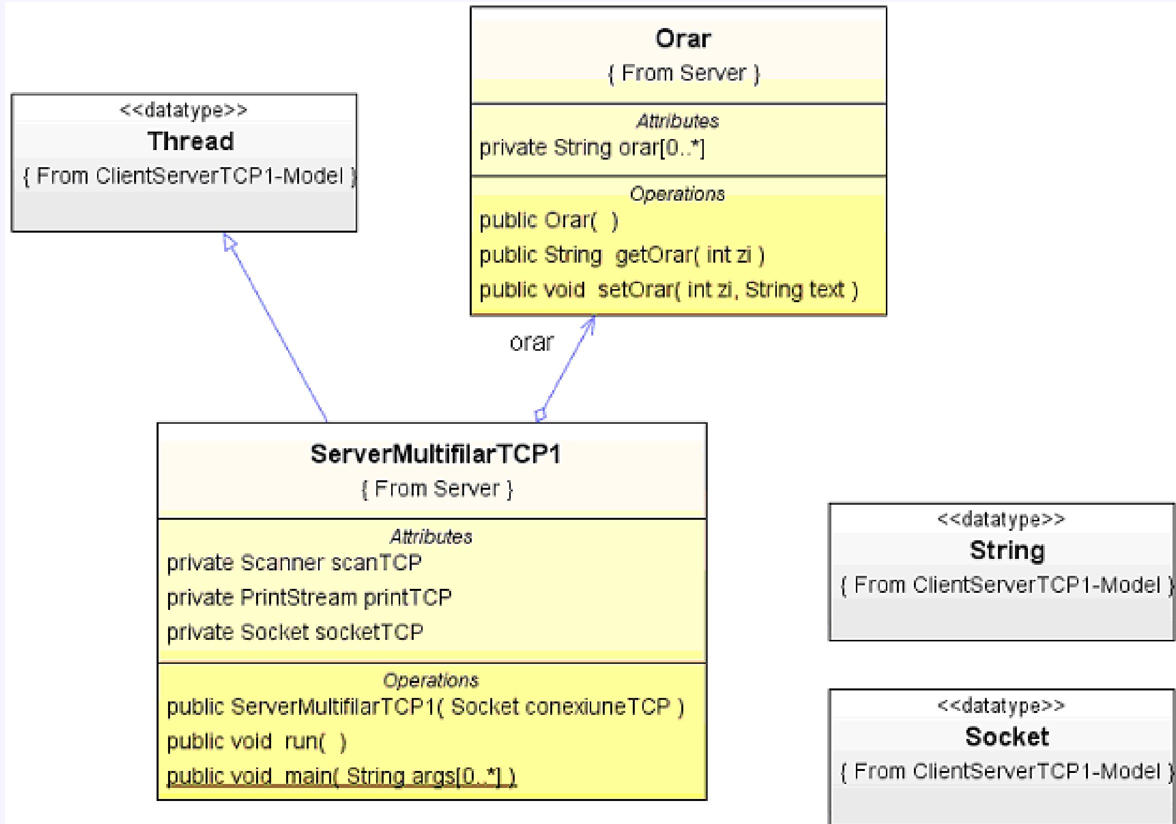
1. Se selecteaza **Run File** pe nodul clasei cu numele `ServerMultifilarTCP1`;
2. Se introduce valoarea **5000** pentru portul pe care asculta serverul TCP;
3. Se selecteaza **Run File** pe nodul clasei cu numele `ClientTCP1`;
4. Se introduce valoarea **5000** pentru portul serverului si se introduce adresa IP a calculatorului pe care a fost lansat serverul, sau se apasa Cancel (generandu-se astfel adresa “localhost”);
5. Se selecteaza **Run File** pe nodul clasei cu numele `ClientTCP2`;
6. Se introduce valoarea **5000** pentru portul serverului si se introduce adresa IP a calculatorului pe care a fost lansat serverul, sau se apasa Cancel (generandu-se astfel adresa “localhost”).

Atentie! Se poate observa ca de fiecare data cand se incearca modificarea unei zile din orar si se apasa butonul **Trimite**, serverul ne afiseaza mesajul: “Server: Bye!” si se inchide.

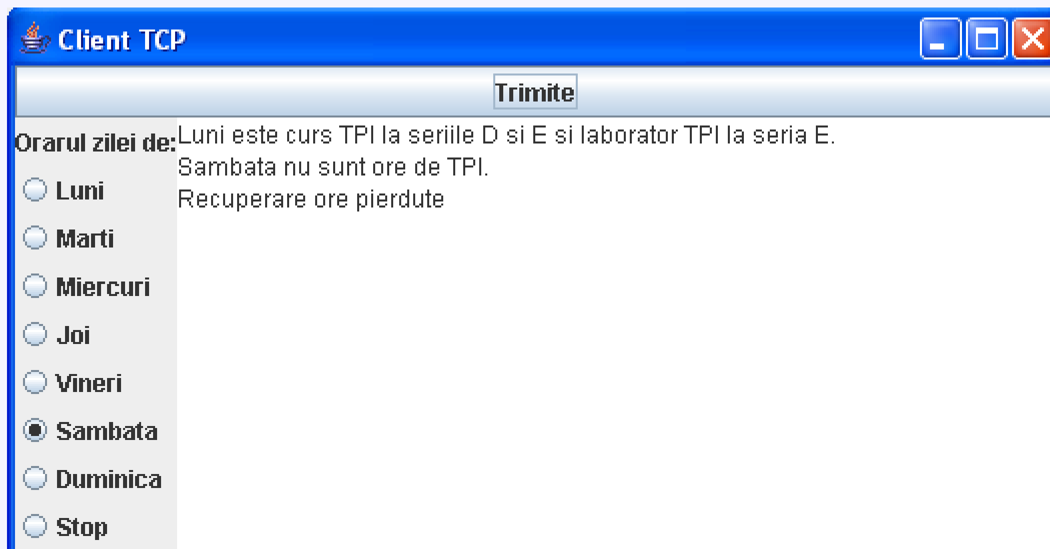
In laborator:

1. Sa se modifice/completeze codul clasei `ServerMultifilarTCP1` pentru a putea oferi si serviciul de modificare a orarului (`setOrar`);
2. Dupa ce serverul face modificarea efectiva in orar, sa trimita clientului un mesaj de confirmare, pe baza caruia clientul sa afiseze in obiectul **JTextArea outGrafic** modificarea facuta;
3. Se reiau pasii necesari executiei `ServerMultifilarTCP1`, `ClientTCP1` si `ClientTCP2`;

Urmatoarea diagrama UML reflecta codul de mai sus:



Asa trebuie sa arate interfata `ClientTCP1` dupa modificarile cerute si utilizarea unui client de tip `ClientTCP1` si a unuia de tip `ClientTCP2` (prin intermediul caruia orarul zilei de sambata a fost modificat in "Recuperare ore pierdute"):



4.5. Teme pentru acasa

Tema 1. Fiecare grup de 2 studenti (cei 2 studenti care lucreaza la acelasi calculator in cadul laboratorului) isi va alege o alta disciplina pentru care sa realizeze orarul (care va trebui sa corespunda celui real din acest semestru). Codul actual al clasei **`Orar`** va fi modificat pentru a permite aflarea orarului detaliat (incluzand orele de desfasurare) la respectiva disciplina, si personalizat pentru grupa si subgrupa din care fac parte studentii care il realizeaza (similar temei de la laboratorul anterior). **2pct**

Tema 2. Adaugarea unui buton care sa permita oprirea serverului multifilar de catre clientii care permit modificarea orarului (de tip 2). **2pct**

Tema 3. Adaptarea aplicatiilor de la clienti pentru a deveni appleturi. **3pct**

Tema 4. Se va inlocui butonul `Trimite` cu doua butoane (441F si 442F) care vor permite afisarea alternativa a orarului pentru cele doua subgrupe. **4pct**

Tema 5. Se va crea un al 3-lea tip de client care sa permita atat vizualizarea cat si modificarea datelor de pe server. Metoda de implementare este la alegere (Ex: 1. prin utilizarea unui obiect de tip `JCheckBox`; 2. prin utilizarea a doua butoane; 3. prin utilizarea unui alt grup de 2 obiecte `JRadioButton`; 4. alta metoda); **5pct**

Tema 6. Crearea altui serviciu, pe formatul celui oferit de clasa **`Orar`**, si adaptarea serverului si clientilor la noul serviciu (se vor modifica de asemenea si componentele grafice in interfata).**4pct**

Temele vor fi predate la lucrarea urmatoare, cate un exemplar pentru fiecare grup de 2 studenti, **pe hartie** (avand numele celor doi studenti scrise pe prima pagina sus) sub forma de **listing**. Se pot implementa la alegere un numar de teme din cele prezentate mai sus. Se pot realiza toate temele, dar punctajul maxim este de 12 puncte (10+bonus).

Anexa

1. Resurse suplimentare privind modul de lucru cu NetBeans IDE 5.5

1.1. Java Tutorials: Learning Swing with the NetBeans IDE
(http://java.sun.com/docs/books/tutorial/uiswing/learn/index.html)

1.2. Java Tutorials: How to Use Buttons, Check Boxes, and Radio Buttons
(http://java.sun.com/docs/books/tutorial/uiswing/components/button.html)

1.3. Java Tutorials: How to Use Combo Boxes
(http://java.sun.com/docs/books/tutorial/uiswing/components/combobox.html)

1.4. Java Tutorials: How to Use Lists
(http://java.sun.com/docs/books/tutorial/uiswing/components/list.html)

Note

Nota1: Definire interfete Java: O interfata Java defineste un set de metode pentru care nu se specifica insa nici o implementare (doar semnatura). O clasa care implementeaza o interfata trebuie obligatoriu sa specifice implementari pentru toate metodele interfetei, supunându-se asadar unui anumit comportament. O *interfata* poate fi vazuta ca o colectie de metode, fara implementare, si declaratii de constante. Interfeţele sunt elemente fundamentale în sistemele OO. Obiectele sunt cunoscute doar prin intermediul interfeţelor lor. O interfaţa nu dă nici un detaliu relativ la implementarea unui obiect, iar obiecte distincte pot implementa diferit o aceeaşi cerere.

O interfaţă Java este declarată prin cuvântul cheie *interface*:

```
[public] interface NumeInterfata
                      [extends SuperInterfata1 [,extends SuperInterfata2...]]
              {
                      //corpul interfetei:constane si metode abstracte
              }
```