

6. INTERFETE GRAFICE – DESIGN

6.1. Notiuni introductive.....	<u>2</u>
6.1.1. Concepte.....	<u>2</u>
6.1.2. Seturi de componente grafice disponibile in Java.....	<u>2</u>
6.1.3. Arhitectura unei aplicatii cu interfata grafica.....	<u>3</u>
6.2. Construirea si afisarea unei interfete grafice Swing.....	<u>4</u>
6.2.1. Tipuri de componente grafice.....	<u>4</u>
6.2.2. Clase de tip componenta.....	<u>4</u>
6.2.3. Etape de lucru.....	<u>5</u>
6.2.4. Aplicare: un scurt exemplu.....	<u>5</u>
6.3. Plasarea si dimensionarea automatizata a componentelor.....	<u>6</u>
6.3.1. Principii.....	<u>6</u>
6.3.2. BorderLayout.....	<u>7</u>
6.3.3. FlowLayout.....	<u>8</u>
6.3.4. GridLayout.....	<u>8</u>
6.3.5. CardLayout.....	<u>8</u>
6.3.6. GridBagLayout.....	<u>9</u>
6.4. Facilitati si detalii de utilizare ale componentelor grafice Swing.....	<u>9</u>
6.4.1. Atribute si metode generale.....	<u>9</u>
6.4.1.1. Conventii de denumire a campurilor si metodelor.....	<u>9</u>
6.4.1.2. Dimensionare si plasare.....	<u>10</u>
6.4.1.3. Culori.....	<u>11</u>
6.4.1.4. Mnemonice.....	<u>11</u>
6.4.1.5. Icon-uri.....	<u>12</u>
6.4.1.6. Alte metode utile.....	<u>12</u>
6.4.2. Lucrul cu label-uri.....	<u>12</u>
6.4.3. Lucrul cu componente de tip buton.....	<u>13</u>
6.4.3.1. Clasa AbstractButton si capacitatatile generale ale butoanelor.....	<u>13</u>
6.4.3.2. Particularitati ale butoanelor de tip toggle.....	<u>13</u>
6.4.4. Lucrul cu componente pentru editare de text.....	<u>14</u>
6.4.4.1. Clasa JTextField.....	<u>14</u>
6.4.4.2. Particularitati JTextArea.....	<u>15</u>
6.4.5. JSlider.....	<u>15</u>
6.4.6. JProgressBar.....	<u>16</u>
6.4.7. Lucrul cu componente de tip container.....	<u>16</u>
6.4.7.1. Clasa Container.....	<u>16</u>
6.4.7.2. Clasa JPanel si subclasele sale.....	<u>17</u>
6.4.7.3. Clasa JFrame.....	<u>17</u>
6.4.7.4. Clasa JDialog.....	<u>17</u>
6.4.8. Meniuri.....	<u>18</u>
6.4.8.1. Structura de meniuri a unei ferestre si clasele corespunzatoare.....	<u>18</u>
6.4.8.2. Meniuri contextuale.....	<u>18</u>
6.5. BIBLIOGRAFIE.....	<u>19</u>
6.6. ANEXA 1 - Ghidul vizual al componentelor Swing.....	<u>19</u>
6.7. ANEXA 2 - Elaborarea unei interfete grafice: un exemplu.....	<u>21</u>

6.1. Notiuni introductive

6.1.1. Concepte

Programele scrise pana acum rulau in consola – input-ul utilizatorului era obtinut sub forma de text scris de la tastatura, iar afisarea output-ului se facea in fereastra de terminal (fie el DOS/Command prompt sau shell Linux/Unix). In realitate insa multe aplicatii Java interactioneaza cu utilizatorul prin intermediul unei interfete grafice, consola avand doar un eventual rol de diagnostic si monitorizare, in masura in care este disponibila.

O interfata grafica este un ansamblu de obiecte numite componente. O componenta este un obiect cu reprezentare vizuala grafica pe ecran, instanta a unei clase derive din *Component* sau *JComponent*. Componentele pot fi butoane, scroll bar-uri, campuri in care utilizatorul poate introduce text, meniuri etc, incluse intr-o componenta de nivel superior (o fereastra sau un applet). Functionarea aplicatiei consta din interactiunea utilizatorului cu aceste componente, prin intermediul operatiilor elementare: apasarea, eliberarea sau click-ul unui buton de mouse cu eventuali modificatori (ALT/SHIFT/CTRL), simpla miscare a cursorului deasupra unei componente, apasari de taste etc.

Nota: interfata grafica a unei aplicatii este deseori referita cu acronimul GUI (Graphical User Interface).

6.1.2. Seturi de componente grafice disponibile in Java

Java pune la dispozitia dezvoltatorilor o infrastructura de clase si pachete destinate construirii de interfete grafice denumita JFC (Java Foundation Classes), si care cuprinde printre altele:

- **AWT** (Abstract Windowing Toolkit) – un set de clase ce defineste un model robust de generare/tratarea evenimentelor generate de interactiunea cu utilizatorul si o ierarhie bogata de componente grafice. Fiecare componenta AWT are de fapt in spate o componenta din GUI-ul sistemului de operare gazda (asa-numitul „native peer” al componentei AWT); spre exemplu, atunci cand masina virtuala Java ruleaza pe Windows, la crearea unui buton AWT el ii va fi “solicitata” Windows-ului, care se va ocupa de desenarea lui. Componentele AWT mai sunt numite **componente heavyweight** si se regasesc in pachetul **java.awt**. Aceasta abordare are urmatoarele caracteristici:
 - avantaje:
 - viteza buna, deoarece componente sunt “asigurate” de sistemul de operare gazda, cu interventie minimala din partea Java
 - in momentul modificarii aspectului componentelor in sistemul de operare gazda (ex: Windows Vista sau 7 comparativ cu Windows XP), componentele AWT migreaza automat la noul aspect
 - dezavantaje:
 - este un set de componente mai simpliste si mai putin flexibile
 - inconsistentele care pot aparea la migrarea aplicatiei de pe un sistem de operare pe altul (componentele grafice au aspect si functionare diferita in functie de sistemul de operare gazda)
- **Swing** – un set suplimentar de componente grafice, care se bazeaza pe API-ul definit de AWT insa realizeaza implementarea integrala in Java. Spre deosebire de AWT, Swing isi deseneaza singur componentele (nu se mai bazeaza pe sistemul de operare si pe „native peers”), ceea ce deschide posibilitati sporite: folosirea de imagini ca etichete pentru componente, transparenta, formatarea textului folosind HTML, posibilitatea de personalizare a aspectului aplicatiilor (facilitate numita „Pluggable look&feel”). Componentele Swing sunt denumite si **componente lightweight** si se gasesc in pachetul **javax.swing**. Aceasta abordare are urmatoarele caracteristici:
 - avantaje:
 - flexibilitate maxima, deoarece desenarea componentelor se efectueaza integral din Java, nemaidepinzand de facilitatile (prezente sau nu) ale componentei in sistemul de operare gazda
 - dezavantaje:
 - viteza mai redusa comparativ cu AWT, deoarece desenarea componentelor se efectueaza integral din cod Java
 - aspectul componentelor nu se modifica automat odata cu imbunatatirile aduse de noile sisteme de operare. Aceasta poate fi in mod egal un avantaj sau un dezavantaj. Aspectul componentelor Swing este dat de asa-numitul “look and feel” - de fapt o tema prestabilita, care nu depinde de

sistemul de operare, ci doar emuleaza aspectul unuia sau altuia dintre sistemele de operare. Este necesara crearea unui nou look and feel pentru a alinia aspectul componentelor la cel al unui nou sistem de operare; pe de alta parte, avem garantia ca, schimbând sistemul de operare, aspectul aplicației nu este afectat.

In afara de seturile de componente incluse in JFC există felurite soluții alternative pentru dezvoltarea de aplicații grafice. Mentionăm aici **SWT** (Standard Widget Toolkit), care este un set de componente grafice dezvoltat la origini de IBM și preluat apoi de proiectul Eclipse. SWT se situează undeva între AWT și Swing, încercând să preia avantajele amanduroră: viteza primului și flexibilitatea celui de-al doilea. SWT „imbracă” într-un API unitar componente native ale sistemului de operare gazdă, folosindu-le direct ori de câte ori este posibil (asa cum face AWT) însă acolo unde este nevoie emulează cu cod Java facilitatile lipsă (abordarea Swing).

Materialul de față va prezenta interfețele grafice Swing, care însă se bazează pe infrastructura oferită de AWT: să ne gândim numai la faptul că clasa parinte a tuturor componentelor Swing - *JComponent* - deriva din *Container* (clasa AWT) care la rândul său deriva din *Component* (clasa parinte a tuturor componentelor grafice AWT).

6.1.3. Arhitectura unei aplicatii cu interfata grafica

O practică recomandată în cazul aplicațiilor cu interfață grafică este separarea logicii aplicației de aspectul acesteia. Să considerăm exemplul unui program ce memorează cartile dintr-o bibliotecă; am putea începe implementarea acestei cerințe creând o clasa *Carte* și una *Biblioteca*. Cea de-a două este cea responsabilă cu memorarea listei de carti (și deci cu operațiile de adăugare, stergere și căutare a unei carti). Aceste două clase formează nucleul aplicației și pot implementa funcționalitatea dorită independent de modul în care utilizatorul va interacționa cu ele:

- dacă interacțiunea cu utilizatorul se desfășoară din consola, acestuia i-ar putea fi prezentat un meniu cu operațiile posibile (adăugare carte, căutare etc) și prin selectii în acest meniu el va avea acces la funcțiile programului
- dacă aplicația beneficiază de interfață grafică, *aceeași* operații vor fi efectuate printr-o interacțiune mult mai comodă pentru utilizator: introducerea de date într-un formular, click pe butoane, parcurgerea rezultatelor căutării folosind scrollbar-uri etc

Sesiuzăm însă că, oricare ar fi soluția aleasă pentru interacțiunea cu utilizatorul, clasele *Carte* și *Biblioteca* vor fi implementate în același mod, deoarece în ambele cazuri managementul listei de carti se realizează prin apelarea unor metode ale acestor clase (*adaugaCarte()*, *cautaCarte()* etc). Din acest punct de vedere putem privi interfața grafică ca pe o simplă adăugire – un mod mai elegant de a interacționa cu aceeași logică a aplicației. Este total contraindicat că clasele *Carte* și *Biblioteca* să fie “conștiente” (sa depindă în vreun fel) de compozitia interfeței grafice – sau fie să nu existe ei! Dacă scriem nucleul aplicației dependent de GUI, interdependența portiunilor de cod poate duce la situații precum următoarele:

- dacă vom dori la un moment dat să folosim codul din nucleu în alta aplicație (care are un alt fel de interfață grafică sau nu are deloc!), vom fi obligați la o rescriere consistentă a nucleului pentru a se potrivi noilor necesități
- dacă pentru aceeași aplicație dorim să schimbăm interfața grafică, vom fi obligați și să efectuăm modificări în nucleu

A separa logica aplicației de interfață grafică oferă posibilitatea dezvoltării și depanării modulare, independente, a celor două. Necesitatea acestui gen de separare există de mult timp în programare și a fost elaborată în timp o soluție ce-i răspunde: un design pattern¹ numit **MVC (Model-View-Controller)**.

Conform design pattern-ului MVC, o aplicație cu interfață grafică este divizată în următoarele elemente:

- **model** – reprezintă ansamblul claselor ce dă funcționalitatea programului, logica sa (ex: în cazul exemplului de mai sus, o clasa *Carte*, și una *Biblioteca* ce permite operațiile de adăugare/stergere/căutare/ordonare etc)
- **view** – interfața dintre utilizator și model. View-ul reprezintă fatada aplicației; acțiunile utilizatorului aplicate view-ului (click de mouse, apăsari de taste etc) au ca ecou operații efectuate de model. Spre exemplu, dand click pe butonul de adăugare introducem un nou obiect *Carte* în lista menținută de obiectul *Biblioteca*
- **controller** – reprezintă „veriga lipsă” între view și model, și anume codul care tratează interacțiunea cu utilizatorul (un click de mouse, închiderea unei ferestre, apăsarea unei taste etc). Acest cod specifică ce metode

¹ Un design pattern reprezintă o soluție tehnică standard (sablon) dezvoltată de-a lungul timpului și aplicabilă unei categorii de probleme din programare

ale modelului trebuie apelate – si cu ce parametri – in cazul aparitiei unui anumit eveniment din partea view-ului. Spre exemplu, un click pe butonul “Cautare carte” poate avea ca efect apelarea metodei *cautaCarte()* din clasa *Biblioteca*, pasandu-i ca parametri datele specificate de utilizator in celelalte campuri ale GUI-ului (titlu, autor etc)

Nota: deseori, GUI-ul nu face altceva decat sa astepte input de la utilizator; programul nostru „munceste” cu adevarat numai atunci cand apare un eveniment, o actiune din partea utilizatorului. De aici si conceptul de programare “event-driven”: programatorul scrie cod ce va fi rulat ca reactie la interactiunea interfetei grafice cu utilizatorul.

Materialul de fata se va concentra pe view – realizarea design-ului interfetei grafice astfel incat sa corespunda cerintelor aplicatiei – urmand ca, odata rezolvata problema aspectului, intr-un capitol viitor sa atasam functionalitate diverselor componente prin “conectarea” lor cu metode ale modelului.

6.2. Construirea si afisarea unei interfete grafice Swing

6.2.1. Tipuri de componente grafice

Componentele grafice se impart in doua categorii:

- **componente elementare** (simple) – sunt componentele uzuale cu care interacționează utilizatorul (butoane, checkbox-uri etc). Acestea nu se pot afisa pe ecran de sine statator, ci doar prin inglobarea lor într-un container
- **componente de tip container** – sunt gădite special pentru a conține în interiorul lor alte componente (ex: fereastra sau panel-ul, care pot conține butoane, scrollbar-uri, checkbox-uri etc). Acestea sunt la randul lor de două feluri:
 - containere radacina (“root containers”) – sunt singurele componente care pot fi afisate direct pe ecran. Principalele astfel de containere sunt ferestrele și applet-urile². Atunci când un root container se afisează, el transmite comanda de desenare (afisare) tuturor componentelor pe care le conține. Acesta este singurul mod în care o componentă non-root container ajunge să se afiseze pe ecran
 - containere intermediare – sunt componente care îndeplinesc rolul de container însă pot fi afisate pe ecran doar prin plasarea într-un root container (ex: panel-urile – vezi mai jos)

Dupa cum se va vedea, a crea o interfata grafica inseamna a crea componente simple si containerele, a adauga toate componente in containere in functie de necesitati, a adauga containerele intermediare superioare in root containere si a afisa root containerele.

6.2.2. Clase de tip componentă

In Swing, baza intregii ierarhii de componente o reprezinta clasa abstracta *JComponent* (derivata din *java.awt.Container*, care la randul sau deriva din *java.awt.Component*). Subclasele concrete ale acesteia formeaza intreaga baza de componente disponibile programatorului; ele se gasesc in pachetul **javax.swing** si subpachetele sale.

Crearea unei componente se realizeaza prin simpla instantiere a clasei corespunzatoare (subclasa directa sau indirecta de *JComponent*); componenta nu va fi insa afisata pe ecran in momentul instantierii, ci dupa cum urmeaza:

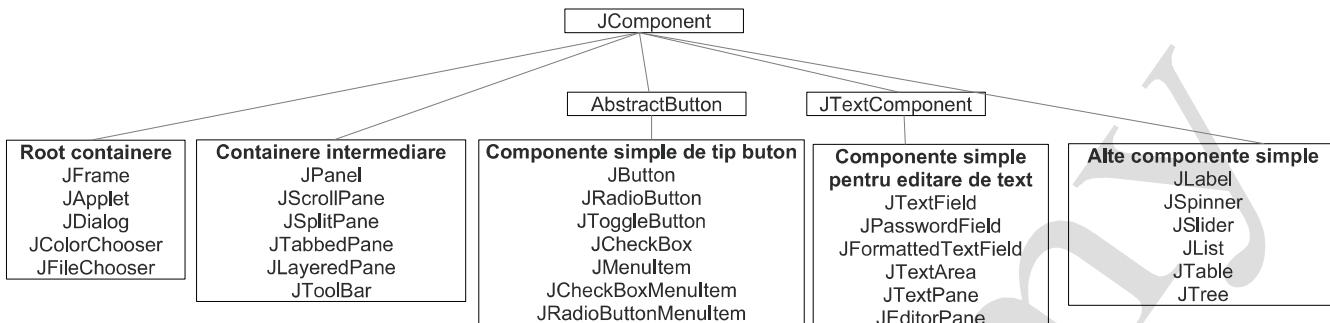
- daca componenta e un root container:
 - pentru ferestre este necesar ca programatorul sa apeleze explicit metoda de afisare (*setVisible()*)
 - applet-urile sunt afisate odata cu aparitia in browser a paginii web din care fac parte
- daca componenta e una simpla sau un container intermediar, ea va fi afisata numai ca urmare a afisarii containerului din care face parte.

Fiecare componentă conține cod care ii deseneaza continutul. La afisarea unui container acesta isi deseneaza partea vizibila (neacoperita de alte componente) si apoi transmite comanda de afisare catre componentele direct continute, care la randul lor isi vor folosi propriul cod de afisare pentru a isi desena suprafața. Daca subcomponentele sunt la randul lor containere, ele vor propaga comanda de afisare catre subcomponentele lor s.a.m.d. pana cand intreaga interfata grafica a fost desenata.

² Un applet este o aplicatie Java care ruleaza din cadrul browserului, ca parte a unei pagini web

Studentul poate utiliza prezentul material si informatiile continute in el exclusiv in scopul asimilarii cunoștințelor pe care le include, fara a afecta dreptul de proprietate intelectuala detinut de InfoAcademy.

Ierarhia de clase componente se prezinta astfel (nota: nu a fost reprodusa integral si fidel ierarhia de clase, ci se doreste mai degraba o identificare a relatiilor si similitudinilor intre diversele componente):



Atentie! Studiati anexa materialului care contine ghidul vizual explicat al componentelor! Este necesara (atat in examen cat si in viata reala) cunoasterea rolului si posibilitatilor fiecarei componente!

Constatam ca exista serii intregi de clase care au un parinte comun, desi vizual se prezinta diferit. Acest lucru se intampla deoarece, independent de aspect, componentele in cauza au acelasi mod de manifestare sau de interactiune cu utilizatorul, si deci campuri si metode comune. Spre exemplu, toate componentele derive din *AbstractButton* pot afisa un text/icon si au posibilitatea de a fi "apasate", chiar daca unele dintre ele isi mentin aceasta stare (ex: *JCheckBox*, *JToggleButton*) si altele nu (*JButton*, *JMenuItem*).

Nota: exista si alte componente grafice – atat elementare cat si de tip container – insa materialul de fata le prezinta pe cele mai importante si folosite.

6.2.3. Etape de lucru

Elaborarea si afisarea unei interfete grafice presupune o insuruire logica de pasi, dupa cum urmeaza:

- design
 - schitarea interfetei grafice – este o etapa pre-programare, realizata deseori pe hartie (sau echivalentul...), care presupune alegerea aspectului si a modului in care utilizatorul va interactiona cu interfata grafica
 - identificarea tipurilor de componente ce compun interfata grafica elaborata
 - divizarea interfetei grafice in containere alese in mod convenabil astfel incat aspectul resultant sa fie cel dorit chiar si in conditiile redimensionarii ferestrei
- implementare
 - instantierea tuturor componentelor (elementare+containere)
 - stabilirea proprietatilor componentelor (dimensiuni, culori, text, icon-uri etc)
 - stabilirea regulilor interne de gestionare a dimensiunii si pozitiei componentelor in cadrul containerelor (vezi sectiunea despre layout manager)
 - adaugarea componentelor in containerele lor, in mod recursiv, pana se ajunge la root containere
 - dimensionare si plasare root containere
- afisare root containere, ceea ce va determina afisarea tuturor componentelor continute in acestea

6.2.4. Aplicare: un scurt exemplu

```

public class ExempluGUI {
    public static void main(String[] a){
        // instantierea componentelor
        String s=("Eu sunt ");
        JFrame f=new JFrame();
        JPanel p=new JPanel();
        JButton b=new JButton(s+"JButton");
        JTextArea ta=new JTextArea(s+"JTextArea",3,15); // 3 linii, 15 coloane
        JLabel l=new JLabel(s+"JLabel");
        JTextField tf=new JTextField(s+"JTextField");
    }
}
  
```

```

JCheckBox cb=new JCheckBox(s+"JCheckBox");
JRadioButton rb=new JRadioButton(s+"JRadioButton");
JMenuBar mb=new JMenuBar();
JMenu m=new JMenu(s+"JMenu");
JMenuItem mi=new JMenuItem(s+"JMenuItem");
// crearea unui combo box populat pe baza unui tablou de String
String[] elemente = {s+"JComboBox"};
JComboBox c=new JComboBox(elemente);

// adaugam componente elementare in containerul intermediar de tip panel
p.add(l);p.add(tf); p.add(cb);p.add(rb);p.add(b); p.add(c);p.add(ta);
// construim bara de meniu a ferestrei si o adaugam la frame
m.add(mi); mb.add(m); f.setJMenuBar(mb);
// adaugam panelul la frame, dimensionam automat frame-ul si il afisam
f.add(p); f.pack(); f.setVisible(true);
}
}

```

Fereastra rezultata se va prezenta astfel (nota: folosind *look and feel*-ul Nimbus):



Observam plasarea componentelor in container in ordinea adaugarii, de la stanga la dreapta, mentionand dimensiunea implicita a componentei chiar si acolo unde aceasta creeaza un efect vizual neplacut. Vor fi prezentate in continuare modalitatile de plasare si dimensionare a componentelor in cadrul unui container.

6.3. Plasarea si dimensionarea automatizata a componentelor

6.3.1. Principii

Scopul unui container este acela de a ingloba alte componente, aranjandu-le in modul dorit de catre programator. Avem doua posibilitati pentru plasarea si dimensionarea componentelor din container:

- stabilirea manuala a dimensiunii si pozitiei dorite pentru fiecare componenta in parte, si de asemenea precizarea regulilor de redimensionare/repositionare a componentelor in cazul redimensionarii containerului din care fac parte (ex: resize de fereastra). Este varianta cea mai flexibila dar si cea mai complexa si mai putin eficienta
- folosirea unui layout manager. Un *LayoutManager* reprezinta un obiect ajutator de care se foloseste containerul pentru a determina modul in care vor fi distribuite componente in interiorul sau. Layout manager-ul stabileste un sablon – un mod prestabilite de plasare si dimensionare a componentelor in cadrul containerului – degrevand programatorul de grija plasarii manuale a componentelor in container si de stabilirea regulilor dupa care diferitele componente isi schimba pozitia si dimensiunile atunci cand containerul parinte este redimensionat.

Stabilirea layout manager-ului pentru un container se realizeaza apeland metoda *void setLayout(LayoutManager)* a container-ului, pasandu-i ca argument:

- null - pentru a nu folosi un layout manager. In acest caz, plasarea componentelor se face manual, folosind metodele *setSize()* si *setLocation()* sau *setBounds()* pentru fiecare componenta in parte
- un obiect de tip *LayoutManager*. *LayoutManager* este o interfata, de aceea argumentele primite de metoda *setLayout()* sunt referinte catre obiecte ale unor clase care implementeaza aceasta interfata.

Atunci cand este adaugata o noua componenta intr-un container care dispune de un layout manager, in general este nevoie de specificarea unor restrictii suplimentare. De aceea metoda *add()* a containerelor este supraincarcata:

- *add(Component c, Object restrictii)* - adauga componenta in cauza tinand cont de restrictiile impuse sub forma celui de-al doilea argument. Natura restrictiilor difera de la un layout manager la altul
- *add(Component c)* - adauga componenta cu restrictiile default (acolo unde layout managerul ofera aceasta posibilitate)

Un layout manager divizeaza suprafata containerului intr-un numar de zone, fiecare zona putand contine cate o componenta. Daca se adauga o noua componenta intr-o zona deja populata, cea veche este eliminata automat. Felul in care componenta dintr-o zona este dimensionata depinde de layout manager-ul ales: unele layout manager vor tine cont de indiciile pe care programatorul le furnizeaza (dimensiune minima, maxima, preferata - vezi sectiunea despre *JComponent*) iar altele nu.

In cadrul unei interfete grafice intervin de obicei mai multe layout manager, deoarece unul singur este rareori suficient (cu exceptia GUI-urilor foarte simple). In etapa de design, programatorul imparte interfata grafica in containere si stabileste layout managerul cel mai potrivit pentru fiecare container. Prin colaborarea acestor preset-uri se obtine in final aspectul si functionalitatea dorita. De aceea programatorul Java trebuie sa cunoasca modul de operare al celor cateva layout manager mai frecvent folosite pentru a le putea specula eficient caracteristicile.

JRE contine cateva clase predefinite care implementeaza *LayoutManager*; o parte dintre ele sunt prezentate in continuare.

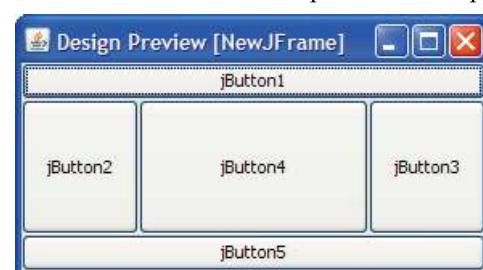
6.3.2. BorderLayout

Acest layout manager imparte suprafata containerului sau in 5 regiuni (zone) in care pot fi plasate componente: cele 4 zone corespunzatoare marginilor containerului, si a 5-a care este cea ramasa in centru. Fiecare zona poate contine exact o componenta (dar care poate fi la randul sau de tip container!). Zonele sunt desemnate prin intermediul unor constante din clasa *BorderLayout* dupa cum urmeaza:

- zona centrala: *BorderLayout.CENTER*
- cele 4 zone periferice pot fi referite in doua moduri:
 - cu denumiri absolute date dupa punctele cardinale: *BorderLayout.EAST*, *BorderLayout.WEST*, *BorderLayout.NORTH*, *BorderLayout.SOUTH*
 - cu denumiri relative, care tin cont de faptul ca unei componente i se poate stabili un sens de populare ce depinde de tara/zona culturala in care este folosita (ex: exista limbi in care se scrie de la dreapta la stanga, altfel viceversa). Constantele folosite sunt *BorderLayout.PAGE_START*, *BorderLayout.PAGE_END*, *BorderLayout.LINE_START*, *BorderLayout.LINE_END*.

Fiecare zona contine o singura componenta. Componenta va fi automat dimensionata astfel incat sa umple toata zona pe care o ocupa, si va fi redimensionata la fiecare modificare a dimensiunii zonei.

La redimensionarea ferestrei, zona de centru preia toata diferenta de spatiu: spre exemplu, la marirea ferestrei pe orizontala, zona vestica si cea estica isi vor pastra latimea, in schimb se vor lati zonele de centru, nord si sud. Analog, la marirea pe verticala, se va modifica inaltimea numai pentru zonele estica, vestica si centrala.



Adaugarea componentelor intr-un container cu *BorderLayout* se face specificand ca restrictie zona dorita - fie sub forma unei constante din clasa *BorderLayout*, fie sub forma unui simplu *String* ce contine numele punctului cardinal dorit sau al centrului: "North", "South", "East", "West" sau "Center":

```
...
panel.setLayout(new BorderLayout());
panel.add(textfield1,"North");
// acelasi efect putem obtine cu:
panel.add(textfield1, BorderLayout.NORTH);
...
```

Nota: *BorderLayout* reprezinta aranjarea implicita pentru containerul Frame.

6.3.3. FlowLayout

Acest layout manager plaseaza componentele in ordinea adaugarii in container urmand aceeasi logica dupa care scriem pe o pagina: de la stanga la dreapta (in limita latimii containerului) si de sus in jos, trecand pe randul urmator cand am ajuns la capat de rand. Atunci cand un rand din container “se termina” noile componente sunt trecute automat dedesupt si repeta procesul pana cand sunt plasate toate componentele.

FlowLayout respecta dimensiunile originale ale componentelor (nu le redimensioneaza la plasarea in container).

Fiecare linie e centrata pe orizontala din oficiu, insa acest lucru poate fi modificat (cu metoda *setAlignment()*, pasandu-i ca argument constante din clasa *FlowLayout*: *CENTER*, *LEADING*, *TRAILING*, *LEFT*, *RIGHT*)

In caz de redimensionare a containerului pe orizontala, componentele se redistribuie pe randuri in functie de noul spatiu disponibil pe fiecare rand. Spre exemplu, atunci cand utilizatorul reduce dimensiunile ferestrei pe orizontala, latimea “randului” devine insuficienta iar componentele care nu mai incap pe randul curent sunt automat trecute dedesupt (in acelasi fel in care se reaseaza textul din notepad cand modificam latimea ferestrei).



Nota: *FlowLayout* este layout manager-ul implicit pentru containere de tip *Panel* si *JPanel*.

6.3.4. GridLayout

GridLayout imparte suprafata containerului intr-o grila de celule, cu numar de linii si coloane specificat de programator la crearea obiectului:

```
p.setLayout(new GridLayout(3,2)); // 3 linii, 2 coloane
```

Fiecare celula a grilei poate contine exact o componenta. Componenta va fi dimensionata automat astfel incat sa umple toata celula (nu sunt onorate dimensiunile preferate ale componentei).

Componentele sunt adaugate in grila pe rand, de la stanga la dreapta si de sus in jos; nu putem specifica linia si coloana dorita (nu avem voie sa „sarim” celule). **Atentie! Nu exista posibilitatea adaugarii unei componente “la punct fix”, prin specificarea coordonatelor celulei!**



La redimensionarea containerului toate celulele isi modifica dimensiunea si implicit vor face acelasi lucru si toate componentele din cadrul lor.

Acest layout este recomandabil cand se doreste o buna aliniere a componentelor dar este tolerabila modificarea dimensiunii acestora (vezi figura).

6.3.5. CardLayout

CardLayout este un layout manager care trateaza toate componentele din cadrul unui container ca fiind un pachet de carti de joc: toate exista simultan, insa numai una este vizibila la un moment dat (cea aflata “deasupra”). Programatorul are la dispozitie metode pentru adaugare de componente, pentru parcurgerea componentelor din container si pentru setarea componentei vizibile.

Acest layout manager este util atunci cand anumite portiuni ale unei interfete grafice se schimba in functie de interacțiunea cu utilizatorul; spre exemplu, intr-o interfata de afisare a caracteristicilor unui fisier, userul poate opta pentru afisarea detaliilor fisierului sau a unui preview al acestuia, in aceeasi suprafata de container.

Fiecare componenta plasat intr-un CardLayout este in general un panel (dar nu exista vreodata obligativitate in acest sens). La adaugarea componentei in panel se specifica numele acesteia, ce poate fi folosit ulterior pentru selectia acelei "carti din pachet". Metodele clasei CardLayout permit apoi stabilirea componentei afisate astfel:

- afisarea primei sau a ultimei componente: metodele *first()* si *last()*
- urmatoarea si precedenta componenta: metodele *next()* si *previous()*
- o anumita componenta - metoda *show()*, unul dintre argumentele sale fiind NUMELE componentei asa cum a fost stabilit la adaugarea acesteia in container

Nota: un efect echivalent lui CardLayout se poate obtine in Swing folosind o componenta de tip JTabbedPane.

```
CardLayout cl = new CardLayout();
panel.setLayout(cl);
panel.add(textfield1, "text"); panel.add(label1, "label"); panel.add(checkbox1,"check");
cl.last(panel);           // se afiseaza checkbox-ul
cl.first(panel);          // se afiseaza textfield-ul
cl.show(panel,"label");   // se afiseaza label-ul
cl.next(panel);           // se afiseaza checkbox-ul
```

6.3.6. GridBagLayout

GridBagLayout este un layout manager complex care incercă să pastreze avantajele lui GridLayout dar să aducă flexibilitatea care îl lipsea acestuia din urmă. Suprafața containerului este divizată tot într-o grilă de celule, însă cu următoarele diferente față de GridLayout:

- fiecare componentă poate fi plasată în celula dorită, prin specificarea coordonatelor acesteia
- fiecare componentă poate ocupa una sau mai multe celule ale grilei, pe orizontală sau pe verticală
- componenta nu mai este automat redimensionată astfel încât să ocupe toată celula; acest lucru se întâmplă numai la dorința programatorului, și separat pentru orizontală și verticală

Pentru fiecare componentă adăugată, programatorul poate specifica o serie întreagă de caracteristici, folosind ca argument secund al metodei *add()* un obiect de tip

GridBagConstraints ce stabilizează următoarele proprietăți:

- pozitia componentei în grila (*gridX,gridY*)
- câte celule ocupă componentă (*gridWidth, gridHeight*)
- dacă componentă umple celula în care se află, pe orizontală sau pe verticală (*fill*), sau dacă ramane cu dimensiunea preferată
- (în cazul în care componentă nu umple celula) felul în care este ancorată componentă față de marginile, centrul sau colturile celulei (*anchor*)
- spatiul pastrat între componente și marginile celulei (*insets*)
- spatiul minim în jurul componentei (*ipadx, ipady* - "internal padding")
- proportia în care componentă preia redimensionările containerului parinte (*weightx, weighty*). Wheight-urile sunt numere subunitare; pe fiecare linie și coloana a matricei suma weight-urilor trebuie să fie 1.



Dată fiind complexitatea acestui layout manager, în general este preferabilă personalizarea sa din cadrul unui designer de interfețe grafice (ex: cel din NetBeans, care conține o modalitate facilă de editare a unui GridBagLayout).

Nota: pentru un exemplu de cod complet, comentat, care utilizează și GridBagLayout, consultați anexa 2 a materialului.

6.4. Facilități și detalii de utilizare ale componentelor grafice Swing

6.4.1. Atribute și metode generale

6.4.1.1. Convenții de denumire a campurilor și metodelor

Modul de denumire a campurilor și metodelor Java respectă și în cazul componentelor grafice un set de convenții. Dacă un atribut al clasei se numește *atr*, atunci:

- setterul se va numi **setAtr()** si va primi un argument ce are acelasi tip de date ca si *atr*
- getterul se va numi **getAtr()**, nu va primi argumente si va avea tipul de date de intoarcere identic cu *atr*. Exceptie fac campurile cu tip de date *boolean*, in cazul carora getterul este denumit **isAttr()**.

Campurile mentionate sunt prezente si editabile in designer-ele de interfete grafice ale mediilor de dezvoltare (ex: in NetBeans, in fereastra Properties).

Prezentam in continuare cele mai importante campuri si metode ale diverselor componente Swing, cu urmatoarele mentiuni:

- doar o parte dintre campurile si metodele prezentate sunt utile din punctul de vedere al acestui capitol, care se concentreaza pe partea de design; restul vor deveni insa utile odata cu capitolul urmator, unde este prezentata tratarea evenimentelor si interacțiunea cu componentele
- componente mai complexe (*JList*, *JTable*, *JComboBox*, *JSpinner*) vor fi prezentate in materialul urmator, dupa ce sunt bine fixate cunostintele de design

Toate componentele grafice AWT si Swing deriva direct sau indirect din clasa *Component* si deci mostenesc metodele acestei clase. Prezentam asadar in continuare cateva metode de interes general cuprinse in *Component* si *JComponent*.

6.4.1.2. Dimensionare si plasare

6.4.1.2.1. Scenarii posibile

Dimensionarea si plasarea unei componente se realizeaza astfel:

- daca componenta este un root container, programatorul stabileste direct dimensiunea si locatia sa prin apelarea unor metode (vezi mai jos)
- daca componenta este un container intermediar sau o componenta simpla, ea trebuie plasata intr-un alt container pentru a se putea afisa:
 - atunci cand containerul parinte nu foloseste un layout manager (apelând *setLayout(null)*), componenta este dimensionata si plasata manual prin apelarea metodelor dedicate
 - atunci cand containerul dispune de un layout manager, acesta este cel care decide modalitatea de distribuire a componentelor pe suprafata containerului: unele layout manager onoreaza dimensiunile dorite ale componentelor, pe cand altele le redimensioneaza in functie de necesitati

6.4.1.2.2. Dimensionarea si plasarea manuala

Atunci cand containerul ce contine componenta nu are layout manager, programatorul poate utiliza urmatoarele metode pentru plasarea si dimensionarea acesteia:

- metode de dimensionare
 - **setSize(Dimension d)** si **setSize(int latime, int inaltime)** – metode care stabilesc dimensiunea componentei. Aceste metode sunt utile si pentru root container, care nu au container parinte
- metode de plasare+dimensionare
 - **setBounds(int x, int y, int latime, int inaltime)** si **setBounds(Rectangle r)** – metode folosite pentru a specifica atat locatia, cat si dimensiunea componentei

Clasa *Dimension* dispune de urmatorii membri de interes:

- campurile **publice width** si **height**
- constructorul *Dimension(int width, int height)*
- setterii *double setWidth(int w)*, *double setHeight(int h)* si *setSize(int width,int height)*

6.4.1.2.3. Dimensionarea si plasarea in prezenta unui LayoutManager

Fiecare componenta dispune, printre altele, de urmatoarele 3 proprietati:

- *minimumSize* – reprezinta dimensiunea minima acceptabila a componentei. Metode atasate:
 - **Dimension getMinimumSize()** - returneaza dimensiunea minima a componentei sub forma unui obiect *Dimension*

- **setMinimumSize(Dimension d)** – seteaza dimensiunea minima dorita
- **maximumSize** – dimensiunea maxima acceptabila a componentei. Metode atasate: **Dimension getMaximumSize() si setMaximumSize(Dimension d)**
- **preferredSize** – reprezinta dimensiunea minima la care componenta se afiseaza corect. Metode atasate: **Dimension getPreferredSize() si setPreferredSize(Dimension d)**

Toate aceste proprietati sunt simple indicatii, de care layout manager-ul containerului parinte poate tine cont sau nu. Spre exemplu, *GridLayout* nu onoreaza niciuna dintre aceste preferinte, *FlowLayout* le onoreaza pe toate, iar *BorderLayout* le onoreaza partial: spre exemplu, componentele plasate in nord si sud isi vor pastra inaltimea preferata, insa latimea dorita va fi ignorata.

6.4.1.3. Culori

Fiecare componenta utilizeaza de doua culori:

- *background* – reprezinta culoarea de fundal. Atunci cand componenta este opaca, suprafata sa este umpluta implicit cu aceasta culoare. Metode atasate: **Color getBackground() si void setBackground(Color c)**
- *foreground* – reprezinta culoarea de prim-plan, cu care sunt desenate elementele de pe suprafata componentei. Spre exemplu, pentru butoane, label-uri, textfield-uri etc. aceasta este culoarea cu care va fi afisat textul componentei

Un obiect de tip Color poate fi obtinut in doua moduri:

- folosind una dintre constantele publice ale clasei *Color* pentru culorile uzuale (*Color.RED*, *Color.GREEN* etc)
- folosind unul dintre constructorii clasei *Color*. Putem obtine o culoare specificandu-i componenta RGB cu constructorul **Color(int r, int g, int b)**; spre exemplu, rosu s-ar obtine cu *new Color(255,0,0)*.

6.4.1.4. Mnemonice

Mnemonicele sunt caractere asociate butoanelor sau label-urilor care permit urmatoarele operatii:

- in cazul unui buton, activarea (apasarea) butonului folosind tastatura. Spre exemplu, daca eticheta unui buton este *Adaugare*, utilizatorul va putea apasa butonul folosind in Windows combinatia *ALT-a*.
- in cazul unui label, primirea focusului de catre componenta "pereche" a label-ului. Spre exemplu, daca intr-un formular avem un label ce afiseaza *Nume* alaturi de textfield-ul corespunzator, apasarea combinatiei *ALT-u* va plasa cursorul in cadrul textfield-ului

Metodele pentru managementul mnemonicelor sunt urmatoarele:

- pentru butoane de orice natura (toate subclasele lui *AbstractButton*):
 - **void setMnemonic(int) si int getMnemonic()** - setarea sau extragerea mnemonicului. Tasta ce constituie mnemonicul se defineste sub forma unei constante din clasa *KeyEvent* (*KeyEvent.VK_A*, *KeyEvent.VK_P* etc). In cadrul etichetei butonului va fi evidentiata automat prima aparitie a caracterului corespunzator. Spre exemplu, daca butonul *Modificare* foloseste ca mnemonic tasta *a* (*KeyEvent.VK_A*) atunci eticheta sa va deveni *Modificare*.
 - **void setDisplayedMnemonicIndex(int) si int getDisplayedMnemonicIndex()** - stabilirea sau extragerea pozitiei literei evidente din cadrul etichetei butonului. Pentru un buton care afiseaza *Clear*, *getDisplayedMnemonicIndex()* va returna 3
- pentru label-uri:
 - **void setDisplayedMnemonic(int) si getDisplayedMnemonic()** - desemnarea sau determinarea literei folosite ca mnemonic in cadrul label-ului (si a carei prima aparitie va fi evidenitata automat). Litera se specifica tot sub forma unei constante din clasa *KeyEvent*
 - **void setDisplayedMnemonicIndex(int) si int getDisplayedMnemonicIndex()** - stabilirea sau extragerea pozitiei literei evidente ca mnemonic in cadrul textului etichetei

6.4.1.5. Icon-uri

Multe componente Swing au capabilitatea de a include un icon alaturi de textul afisat (ex: *JLabel*, *JButton*, *JToggleButton* etc). Ele dispun de urmatoarele metode in acest sens:

- **void setIcon(Icon)** - seteaza icon-ul inclus in componenta (vezi mai jos detalii)
- **Icon getIcon()** - returneaza o referinta catre icon-ul componentei

Icon este o interfata, iar principala clasa care o implementeaza este *ImageIcon*. Cea mai simpla modalitate de a crea un *ImageIcon* este sa-l pasa in constructor calea catre un fisier imagine. Sunt acceptate fisierele de tip JPG, GIF sau PNG:

```
ImageIcon poza = new ImageIcon("c:\\poze\\icon1.jpg");
label.setIcon(poza);
```

Nota: spre deosebire de exemplul de mai sus, in viata reala se folosesc modalitati independente de platforma de a accesa resursele externe de tip imagine (vezi capitolul despre localizarea si incarcarea resurselor).

6.4.1.6. Alte metode utile

In plus fata de metodele prezentate mai sus, clasa *Component* dispune si de:

- **setCursor(Cursor c)** – stabileste cursorul afisat atunci cand mouse-ul se afla deasupra componentei
- **setEnabled(boolean b)** – activeaza sau dezactiveaza componenta. O componenta dezactivata nu raspunde la interacțiunea cu userul, nu primește focus și nu generează evenimente (vezi capitolul dedicat evenimentelor)
- **boolean requestFocusInWindow()** - solicita ca componenta sa primeasca focus. Cand o componenta are focus, input utilizatorului este directionat catre ea (ex: dand click intr-un text field, putem apoi sa scriem in el deoarece apasurile de taste sunt directionate catre el). Returneaza false daca componenta nu poate primi focus
- **setFocusable(boolean b)** – stabileste daca componenta poate primi focus
- **setVisible(boolean b)** – arata sau ascunde componenta. Reprezinta modalitatea de a controla afisarea unui root container (ex: o fereastra) dar poate fi aplicata si pentru componente elementare
- **setComponentPopupMenu(JPopupMenu)** - ataseaza un meniu contextual componentei (vezi sectiunea despre meniuri)

6.4.2. Lucrul cu label-uri

Un *JLabel* Swing reprezinta o eticheta rectangulara in cadrul careia se poate afisa text si un icon. Programatorul poate stabili textul, icon-ul, fontul si dimensiunea textului, pozitia acestuia fata de icon si alinierarea intregului continut, atat pe orizontala cat si pe verticala.

Label-urile au proprietatea de a putea fi atasate unei alte componente. Atunci cand un label dispune de mnemonic, actionarea acestuia va face componenta atasata label-ului sa capete focus (ex: intr-un formular, folosirea mnemonic-ului pentru label-ul Nume (ALT-u) va plasa automat cursorul in textfield-ul pentru introducerea numelui).

Metode	Descriere
<i>JLabel()</i> <i>JLabel(String text)</i> <i>JLabel(String text, int alignment)</i>	constructor care creeaza label gol constructor care creeaza un label cu textul specificat constructor care creeaza un label cu textul si aliniamentul specificat
<i>Icon getIcon()</i> <i>setIcon(Icon)</i>	setarea si extragerea icon-ului continut in label
<i>void setText(String)</i> <i>String getText()</i>	setarea si extragerea textului afisat de label
<i>void setLabelFor(Component c)</i> <i>Component getLabelFor()</i>	setarea si extragerea componentei la care este atasat label-ul (cel care va primi focus la actionarea mnemonicului din label)
<i>void setHorizontalAlignment(int)</i> <i>int getHorizontalAlignment(int)</i> <i>void setVerticalAlignment(int)</i> <i>int getVerticalAlignment()</i>	stabilirea si determinarea aliniamentului orizontal sau vertical al continutului label-ului. Aliniamentul se specifica sub forma unei constante a clasei <i>JLabel</i> : LEFT , RIGHT , CENTER , LEADING , TRAILING pentru orizontala, respectiv TOP , BOTTOM , CENTER pentru verticala

```
void setHorizontalTextPosition(int)
int getHorizontalTextPosition()
void setVerticalTextPosition(int)
int getVerticalTextPosition()
```

stabilirea sau determinarea pozitiei textului in raport cu icon-ul pe orizontala sau verticala. Se folosesc aceleasi constante ca mai sus

Nota: constantele pentru diversele tipuri de alinieri (LEFT, RIGHT etc) sunt definite in interfata SwingConstants. Ea este implementata de catre toate clasele componenta care folosesc alinierea a continutului, astfel constantele putand fi accesate si cu numele clasei in cauza (ex: JLabel.LEFT)

6.4.3. Lucrul cu componente de tip buton

6.4.3.1. Clasa AbstractButton si capabilitatile generale ale butoanelor

AbstractButton reprezinta clasa parinte pentru toate componentele cu comportament de buton. Folosim aceasta sintagma deoarece, desi unele dintre ele au intr-adevar aspect de buton (*JButton*, *JRadioButton*) altele nu prezinta similitudini evidente (*JCheckBox*, *JMenuItem*).

Toate componentele de tip buton au in comun faptul ca pot fi "apasate". Operatia se poate realiza in mai multe feluri:

- click cu mouse-ul pe buton
- folosirea mnemonic-ului corespunzator butonului
- navigarea prin interfata grafica folosind TAB pana cand butonul dorit capata focus, si apoi SPACE

Fiecare buton poate afisa un text si un icon, la fel ca in cazul label-urilor; exista insa mai multe icon-uri posibile, corespunzand diferitelor stari ale butonului. Fiecare dintre atributele urmatoare beneficiaza de getter si setter in clasele buton:

- *icon* - corespunde icon-ului afisat din oficiu cand butonul nu este apasat. Este folosit automat si pentru celelalte stari daca nu sunt definite icon-uri separate
- *rolloverIcon* - icon-ul afisat atunci cand cursorul mouse-ului se afla deasupra butonului
- *pressedIcon* - icon-ul afisat atunci cand butonul este apasat
- *disabledIcon* - icon-ul afisat cand butonul este dezactivat

In functie de manifestarea butonului odata ce este apasat, impartim componentele de tip buton in:

- butoane care isi mentin starea ("raman apasate"). Este cazul componentelor de tip *JToggleButton* (cu subclasele *JCheckBox* si *JRadioButton*), *JRadioButtonMenuItem*, *JCheckBoxMenuItem*
- butoane care nu isi mentin starea - clasele *JButton* si *JMenuItem*

Metode	Descriere
<code>String getText(), void setText(String) Icon getIcon(), void setIcon(Icon) int getHorizontalAlignment(), void setHorizontalAlignment(int) ...etc...</code>	metodele pentru stabilirea si extragerea icon-ului, textului si aliniamentelor sunt aceleasi (si actioneaza in acelasi fel) ca in cazul lui <i>JLabel</i>
<code>void setDisabledIcon(Icon), Icon getDisabledIcon() void setRolloverIcon(Icon), Icon getRolloverIcon() void setPressedIcon(Icon), Icon getPressedIcon()</code>	extragerea sau modificarea icon-ului afisat in diferitele stari aditionale ale butonului
<code>void setMnemonic(int), int getMnemonic()</code>	setarea si extragerea tastei folosita ca mnemonic pentru apasarea butonului folosind tastatura (vezi sectiunea despre mnemonice)
<code>void setSelected(boolean), boolean isSelected()</code>	seteaza sau determina starea de "apasat" a butonului. Util pentru componentele de tip buton care, odata apasate, isi mentin aceasta stare (<i>JToggleButton</i> , <i>JCheckBox</i> , <i>JCheckBoxMenuItem</i>)

6.4.3.2. Particularitati ale butoanelor de tip toggle

Clasa *JToggleButton* implementeaza un buton care, apasat prima data, "ramane apasat", iar apasat a doua oara revine la starea initiala. Există două subclase, corespunzătoare componentelor cu manifestare identică: *JCheckBox* și *JRadioButton*.

O particularitate importantă a componentelor de acest fel este aceea că pot forma grupuri în care un singur buton poate fi selectat la un moment dat; dacă utilizatorul apasă un alt buton, acesta se va selecta ("apasa") iar cel precedent se va

deselecta. Desi componentele de tip *JRadioButton* sunt cel mai des folosite in astfel de situatii, capabilitatea exista si in cazul celorlalte doua clase mentionate.

Nota: sintagma “radio button” provine de la aparatele de radio mai vechi la care, cand se selecta o alta gama de frecvențe (lungi, ultrascurte etc) decat cea curenta, butonul precedent “sarea” concomitent cu apasarea celui nou.

Pentru a crea un grup de butoane ce se exclaud reciproc este necesara definirea unui obiect de tip *ButtonGroup* si adaugarea tuturor butoanelor la acelasi grup, folosind metoda *add(AbstractButton)* a acestuia. *ButtonGroup*-ul nu este o componenta grafica; el reprezinta doar “lantul” intre butoanele grupului, oferind metode pentru adaugare/eliminare butoane din grup, enumerarea butoanelor componente si pentru modificarea/determinarea selectiei curente.

Atentie! Un set de toggle button-uri neincluse in acelasi grup vor actiona independent! (inclusiv radio button-urile!)

```
JRadioButton rb1 = new JRadioButton("unu");
JRadioButton rb2 = new JRadioButton("doi");
JRadioButton rb3 = new JRadioButton("trei");
ButtonGroup bg = new ButtonGroup();
bg.add(rb1);bg.add(rb2);bg.add(rb3);
```



Metode ButtonGroup	Descriere
void add(AbstractButton), remove(AbstractButton)	adaugare si eliminare a unui buton in/din grup
void clearSelection()	deselectarea tuturor butoanelor din grup
int getButtonCount()	obtinerea numarului de butoane componente ale grupului
Enumeration getElements()	obtinerea listei de butoane continute in grup
ButtonModel getSelection()	returneaza modelul butonului selectat (vezi in capitolul urmator arhitectura compon. Swing)

6.4.4. Lucrul cu componente pentru editare de text

6.4.4.1. Clasa JTextComponent

JTextComponent este clasa parinte a componentelor care permit editare de text: *JTextField*, *JTextArea* si *JEditorPane*. Capabilitatile comune ale acestor componente sunt urmatoarele:

- permit stabilirea pozitiei cursorului in cadrul textului, de catre user (prin mouse/taste) sau de catre programator (folosind metode - vezi tabelul)
- permit selectarea unei portiuni de text si modificarea selectiei de catre user sau de catre programator
- permit editarea de text neformatat, inclusiv operatii de copy, cut si paste
- permit comutarea componentei in modul needitabil: textul va fi afisat si selectat in continuare si va functiona operatia de copy, insa nu va putea fi modificat

Metode JTextComponent	Descriere
String getText(), void setText(String)	setarea/extragerea textului continut in componenta
boolean isEditable() void setEditable(boolean)	controlarea modului de lucru ne-editabil al componentei
int getCaretPosition() void setCaretPosition(int)	stabilirea/determinarea pozitiei cursorului in cadrul textului. 0 inseamna inaintea primului caracter, 1 dupa primul caracter etc. Daca componenta contine N caractere, atunci pozitia N-1 inseamna inaintea ultimului caracter, iar pozitia N dupa ultimul caracter (finalul sirului)
getSelectedText() selectAll() select(int pozitieStart, int pozitieStop) void setSelectionStart(int), int getSelectionStart() void setSelectionEnd(int), int getSelectionEnd()	modificarea sau determinarea textului selectat in cadrul componentei, in diferite feluri: selectare integrala, selectarea unei portiuni, modificarea pozitiei stangi sau drepte a zonei selectate

Atentie! In cazul componentei JPasswordField, metoda *getText()* este inlocuita cu *getPassword()*!

6.4.4.2. Particularitati JTextField si JTextArea

Clasa *JTextField* are la randul sau doua subclase:

- *JPasswordField* - componenta gandita pentru introducerea de parole. In locul fiecarui caracter scris de utilizator afiseaza un caracter ales de programator (ex: *)
- *JFormattedTextField* - un textfield care impune introducerea datelor intr-un format specificat de programator. Spre exemplu, in cazul unei date calendaristice in format ZZ.LL.AAAA, campul va contine caracterele '!' (punct), iar utilizatorului i se va permite sa editeze numai celelalte pozitii, dar nu va fi lasat sa introduca altceva decat cifre

Componentele de tip *JTextField* (plus derivatele) ofera posibilitatea dimensionarii pornind de la numarul de coloane dorit; exista in acest sens metodele **void setColumns(int)** si **int getColumns()**. Dupa setarea numarului de coloane i se transmite containerului parinte comanda de redesenare.

Clasa *JTextArea* ofera urmatoarele posibilitati suplimentare fata de *JTextComponent*:

- stabilirea dimensiunii componentei pornind de la numarul de randuri si coloane dorite
- operatii de modificare a textului afisat prin adaugare si introducere (operatii relative, comparativ cu singura posibila in *JTextComponent* - inlocuirea)
- controlul wrapping-ului (trecerii automate pe linia urmatoare in cazul liniilor lungi)
- operatii de translatie dintr-un sistem de coordinate in altul. Spre exemplu, dat fiind un numar de linie putem afla pozitia caracterului de inceput si de final al liniei in cadrul intregului text; sau, avand pozitia (numarul) unui caracter, putem determina linia pe care se afla acesta.

Metode JTextArea	Descriere
void setColumns(int), int getColumns()	setarea/extragerea numarului de coloane (latimea in litere a componentei)
void setLineWrap(boolean), boolean getLineWrap()	impune sau determina strategia de divizare automata a liniilor lungi
int getLineCount()	determinarea numarului de linii afisate in componenta
int getLineStartOffset(int nrLinie) int getLineEndOffset(int nrLinie) int getLineOfOffset(int)	determinarea pozitiei caracterului de inceput pentru o anumita linie determinarea pozitiei caracterului de final pentru o anumita linie determinarea liniei pe care se afla un caracter in functie de pozitia (numarul) acestuia
void append(String) void insert(String, pozitie) void replaceRange(String, pozitieStart, pozitieFinal)	adauga sirul primit ca argument la cel existent in cadrul componentei introduce sirul specificat pe pozitia dorita, "mutand" la dreapta continutul existent inlocuieste caracterele aflate intre pozitiile [pozitieStart, pozitieFinal-1] cu sirul cerut

Nota: metoda *insert()* poate primi ca argument chiar lungimea sirului existent in *JTextArea*: daca aceasta are N caractere, *insert("sirNou",N)* este echivalent cu *append("sirNou")*.

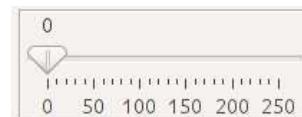
Atentie! Metoda *replaceRange()* nu inlocuieste si caracterul de pe pozitia de final! Astfel, pentru un *JTextArea* care contine textul "mare", este legal apelul *replaceRange("sa",2,4)*, iar rezultatul va fi "masa". In acelasi timp, apeland pentru acelasi sir *replaceRange("es",1,3)*, sirul rezultat va fi "mese".

6.4.5. JSlider

JSlider este o componenta ce permite reglarea unei valori intregi prin mutarea unui cursor, ca in figura alaturata. Intern, componenta dispune de 3 atribute de tip intreg: minimum, maximum si value. *Value* reprezinta valoarea curenta, ea putand fi reglata intre limitele date de celelalte doua atribute.

Vizual, componente i se pot regla urmatoarele aspecte:

- afisare gradatii - pot fi afisate *major ticks* (liniile mari, mai rare, din figura alaturata) si *minor ticks* - liniile mici, mai dese)
- intervalul dintre gradatatile majore, respectiv dintre cele minore
- afisare etichete valorice in dreptul major tick-urilor
- orientare - orizontala sau verticala
- ordonarea valorilor (crescatoare sau descrescatoare). In ordonarea descrescatoare, valorile sunt dispuse in ordine inversa in cadrul componentei, iar cursorul este plasat initial pe valoarea maxima



Metode JSlider	Descriere
void setMinimum(int), int getMinimum() void setMaximum(int), int getMaximum() void setValue(int), int getValue()	setarea/extragerea valorii minime setarea/extragerea valorii maxime setarea/extragerea valorii curente
void setOrientation(int), int getOrientation()	setarea/determinarea orientarii. Valori: JSlider.VERTICAL, JSlider.HORIZONTAL
void setPaintTicks(boolean), boolean getPaintTicks() void setMajorTickSpacing(int), int getMajorTickSpacing() void setMinorTickSpacing(int), int getMinorTickSpacing()	setarea/determinarea afisarii gradatilor setarea/extragerea intervalului dintre doua gradatii majore consecutive setarea/extragerea intervalului dintre doua gradatii minore consecutive
void setPaintLabels(boolean), boolean getPaintLabels()	setarea/determinarea afisarii etichetelor valorice
void setInverted(boolean), boolean getInverted()	true produce ordonare descrescatoare a valorilor

6.4.6. JProgressBar

JProgressBar este o componenta utilizata pentru a evidenta evolutia unei operatii. Ca implementare interna foloseste aceleasi principii ca si JSlider: are o valoare interna (*value*) ce evolueaza intre *minimum* si *maximum*, toate cele trei valori fiind de tip intreg si configurabile. In plus, componenta poate afisa un text (dat de atributul *string*) si poate functiona in modul *indeterminate* - propriu operatiilor a caror evolutie nu poate fi urmarita, dar pentru care vrem sa evidentiem faptul ca sunt inca in desfasurare.

JProgressBar nu are (si nici nu ar putea avea) o modalitate automatizata de a urmari progresul unei operatii, data fiind diversitatea operatiilor posibile. Programatorul este cel in masura sa sesizeze evolutia operatiei in cauza si sa modifice *value* pe masura ce operatia avanseaza.

Metode JProgressBar	Descriere
void setMinimum(int), int getMinimum() void setMaximum(int), int getMaximum() void setValue(int), int getValue()	setarea/extragerea valorii minime setarea/extragerea valorii maxime setarea/extragerea valorii curente
void setOrientation(int), int getOrientation()	setarea/determinarea orientarii, cu constantele VERTICAL si HORIZONTAL ale clasei
void setStringPainted(boolean), boolean isStringPainted() void setString(String), String getString()	controleaza afisarea textului seteaza/extrage textul componentei
void setIndeterminate(boolean), boolean isIndeterminate()	controleaza modul de lucru "indeterminate"

6.4.7. Lucrul cu componente de tip container

6.4.7.1. Clasa Container

Dupa cum se cunoaste deja, o componenta de tip *Container* este una care poate contine la randul sau alte componente (fie ele containere sau componente simple). In acest scop, clasa *Container* inglobeaza metodele necesare pentru adaugarea, eliminarea si enumerarea componentelor continue:

- **add(Component c)** – adauga componenta specificata la sfarsitul listei deja existente in container
- **add(Component c, Object Constraints)** – folosita pentru a adauga componente in container in conditiile in care acesta foloseste un LayoutManager. Exemplu: in cazul lui BorderLayout, *add(c, "North")*.
- **getComponents()** - returneaza un tablou de Component ce contine referintele catre toate componentele continue
- **getComponentCount()** - returneaza numarul de componente aflate in container
- **remove(Component c)** – elimina componenta specificata din container
- **removeAll()** - elimina toate componentele din container
- **setLayout(LayoutManager m)** – stabileste layout manager-ul folosit pentru containerul in cauza

Nota: nu este nicio greseala - este vorba despre clasa *java.awt.Container*! In Swing nu exista clasa *JContainer*, caci *JComponent* deriva oricum din *java.awt.Container*!

6.4.7.2. Clasa JPanel si subclasele sale

Panelurile sunt probabil componentele cele mai putin vizibile dar care “duc greul” interfetelor grafice. Ele sunt simple containere rectangulare, gandite sa contina si sa organizeze in interiorul lor alte componente. Ceea ce face panel-ul pretios in economia unei interfete grafice este posibilitatea de a i se atasca un layout manager si deci de a automatiza dimensionarea si plasarea componentelor continue. Elaborarea unei interfete grafice presupune divizarea suprafetei root containerelor in subpanel-uri de asa natura incat interfata sa aiba aspectul dorit atat la prima afisare, cat si dupa eventualele redimensionari.

Clasa JPanel are cateva subclase specializate:

- **JScrollPane** - este un panel care afiseaza automat scrollbar-uri in cazul in care continutul sau depaseste suprafata disponibila (ex: o poza prea mare, o lista de elemente lunga etc)
- **JSplitPane** - un panel ce contine exact doua componente, despartite de un separator vertical mobil. Cand utilizatorul deplaseaza separatorul, dimensiunile celor doua componente sunt ajustate automat (ce “pierde” una “castiga” cealalta)
- **JTabbedPane** - un panel ce permite afisarea in aceeasi suprafata a mai multor pagini, prin accesarea unor tab-uri (vezi ghidul vizual al elementelor)
- **JLayeredPane** - panel ce permite afisarea suprapusa, in mod controlat, a mai multor componente (“ordonare in profunzime”)

6.4.7.3. Clasa JFrame

JFrame este principalul root container pentru aplicatiile desktop. El corespunde unei ferestre decorate (cu border si bara de titlu). Optional, fereastra poate avea bara de meniuri, titlu, icon, toate aflate sub controlul programatorului.

JFrame-ul are o compositie stratificata: singura sa subcomponenta directa este un *root pane* (instanta de *JRootPane*), care la randul sau poate contine doua subcomponente: un *JMenuBar* (bara de meniu - optionala) si asa-numitul *content pane*. Acesta din urma este cel care contine toate componentele non-meniu in cazul unui *JFrame*; metoda *add()* a *JFrame*-ului este delegata catre el.

In calitatea sa de root container, *JFrame*-ul nu mai poate avea dimensiunile si pozitia stabilite automat de catre un layout manager, ci este responsabilitatea programatorului sa se ingrijeasca de aceste aspecte. De asemenea, afisarea *JFrame*-ului trebuie realizata tot manual, prin apelarea metodei sale *setVisible()*.

Metode JFrame	Descriere
<code>void setTitle(String), String getTitle()</code>	setarea/extragerea titlului ferestrei (acesta este afisat automat in bara de titlu)
<code>void setIconImage(Image), Image getIconImage()</code>	setarea/extragerea icon-ului atasat ferestrei, afisat de obicei in bara de titlu
<code>void getContentPane(Container), Container getContentPane()</code>	setarea/extragerea content pane-ului
<code>void setVisible(boolean) boolean isShowing() void dispose()</code>	afiseaza fereastra pe ecran sau o ascunde, in functie de valoarea argumentului verifica daca fereastra este afisata inchide fereastra si elibereaza resursele alocate ei de catre de sistemul de operare
<code>void setResizable(boolean), boolean isResizable()</code>	controleaza daca fereastra este redimensionabila de catre utilizator
<code>void setUndecorated(boolean), boolean isUndecorated()</code>	controleaza daca fereastra are decoratiuni (border si bara de titlu)
<code>void setExtendedState(int) int getExtendedState()</code>	controleaza starea ferestrei (minimizata, maximizata, normala). Se folosesc constante din clasa JFrame: NORMAL, ICONIFIED (minimizata), MAXIMIZED_HORIZ, MAXIMIZED_VERT, MAXIMIZED_BOTH
<code>setSize(), setBounds() etc</code>	stabilirea pozitiei si dimensiunii se realizeaza deosebi cu metodele din JComponent
<code>pack()</code>	dimensioneaza automat fereastra, alegand cea mai mica dimensiune care “imbraca” componentele continue

6.4.7.4. Clasa JDialog

Ferestrele de dialog sunt ferestre utilizate pentru obtinerea de input de la utilizator sau pentru simpla afisare de mesaje informative sau de eroare. Ele difera de cele obisnuite (*JFrame*) prin faptul ca pot fi afisate **modal** – interzic accesul la o

alta fereastra, care este denumita *fereastra parinte*. Spre exemplu, atunci cand folosim meniul *File* al unei aplicatii pentru a deschide fereastra de dialog pentru *Open* sau *Save*, accesul la fereastra de baza a aplicatiei este oprit atata timp cat dialogul de selectie de fisier este afisat.

Afisarea modală presupune fereastra parinte, si de aceea toti constructorii lui *JDialog* au ca prim argument tocmai aceasta fereastra. Iata cativa dintre ei: **JDialog(Dialog parinte)**, **JDialog(Frame parinte)**, **JDialog(Frame parinte, String titlu, boolean modal)**.

Nota: o modalitate simpla de a crea rapid ferestre de dialog standard este oferita de clasa *JOptionPane* prin metodele sale statice.

6.4.8. Meniuri

6.4.8.1. Structura de meniuri a unei ferestre si clasele corespunzatoare

Dintre toate containerele prezentate, numai *JFrame* si *JDialog* pot avea bara de meniu. Bara ia forma unui obiect de tip *JMenuBar*, clasa *JFrame* disponand de urmatoarele metode:

- **setJMenuBar(JMenuBar)** – seteaza bara de meniu a ferestrei
- **JMenuBar getJMenuBar()** - returneaza o referinta catre obiectul de tip *JMenuBar* asociat ferestrei, sau null in cazul absentei acestuia

Un obiect de tip *JMenuBar* contine obiecte de tip *JMenu*, care la randul lor contin elemente de tip *JMenuItem* si/sau descendente (*JMenu*, *JCheckBoxMenuItem*, *JRadioButtonMenuItem*). Un meniu poate contine la randul sau alte submeniuri.

Clasa *JMenuItem* este clasa parinte a tuturor elementelor ce pot fi incluse intr-un meniu, chiar si a meniului insusi! Ea este la randul sau derivata din *AbstractButton* si deci raman valabile toate caracteristicile si metodele discutate in sectiunea dedicata acestiei. In plus, fiecare *JMenuItem* poate avea un asa-numit *accelerator* - o combinatie de taste care realizeaza actiunea corespunzatoare elementului de meniu fara a mai fi necesara navigarea prin meniuri si apasarea cu ajutorul mouse-ului sau mnemonic-ului.

Comparativ cu *JMenuItem*, clasa *JMenu* adauga metode pentru operatiile specifiche:

- adaugare de elemente in meniu (fie ele de tip *JMenuItem* sau *JMenu*). Noul element poate fi adaugat la sfarsit sau introdus pe o anumita pozitie. Exista variante ale metodelor de adaugare care primesc ca argument un *String* si care creeaza automat un *JMenuItem* cu acel text
- eliminare de elemente din meniu. Elementul ce se doreste eliminat poate fi specificat fie prin intermediul referintei catre el, fie prin intermediul pozitiei sale in cadrul meniului, numerotata de la 0

metode JMenuItem	void setAccelerator(KeyStroke) KeyStroke getAccelerator()	gestionare shortcut atasat
metode suplimentare JMenu	add(Component), add(Component, int pozitie) add(String), insert(String, int pozitie) insert(Component, int pozitie), insert(String, int pozitie)	adaugare componenta in meniu, in diverse moduri. Daca pozitia lipseste sau este -1, componenta se adauga la sfarsitul meniului
	addSeparator(), insertSeparator(int pozitie)	adaugarea unui separator (linie orizontala)
	remove(Component), remove(pozitie), removeAll()	eliminarea unei componente sau a tuturor din meniu

6.4.8.2. Meniuri contextuale

Un meniu contextual (pop-up menu) este de fapt o fereastra separata ce contine elemente de meniu, si care este afisata in urmatoarele situatii:

- la click dreapta pe o componenta (sau la apasarea unei combinatii de taste specifice sistemului de operare)
- la deschiderea unui meniu (*JMenu*) din bara de meniuri
- la deschiderea unui submeniu din cadrul unui meniu

Clasa ce implementeaza un meniu contextual este *JPopupMenu*, ce deriva direct din *JComponent*. Ea dispune de metode pentru managementul listei de elemente continute in meniu. In unele look and feel-uri, la aparitia pe ecran, meniul afiseaza un titlu deasupra listei de elemente - configurabil prin intermediul atributului *label*.

Pentru a atasa un meniu contextual unei componente se foloseste metoda clasei *JComponent* cu semnatura
void setComponentPopupMenu(JPopupMenu).

6.5. BIBLIOGRAFIE

- Ghidul vizual al componentelor Swing: <http://docs.oracle.com/javase/tutorial/ui/features/components.html>
- Arhitectura componentelor Swing: <http://java.sun.com/products/jfc/tsc/articles/architecture/>
- Ghidul vizual al layout managerelor: <http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html>
- Despre diversele dimensiuni ale unei componente: <http://www.developer.com/java/ent/article.php/630651/Swing-from-A-to-Z-Minimum-Maximum-and-Preferred-Sizes.htm>
- Lucrul cu icon-uri: <http://docs.oracle.com/javase/tutorial/uiswing/components/icon.html>

6.6. ANEXA 1 - Ghidul vizual al componentelor Swing

Aspect	Clasa corespunzatoare	Descriere
Label cu icon si text, aliniat la stanga	JLabel	Eticheta needitabila ce permite afisarea unui text si a unui icon. Programatorul poate customiza toate aspectele - fontul si dimensiunea textului, aliniamentul componentei, pozitia si distanta textului fata de imagine etc.
Buton cu icon	JButton	Un buton Swing poate avea atat text cat si un icon, toate aspectele conexe fiind customizable
Optiuni afisare fisiere	JCheckBox	O casuta cu doua stari posibile
<input type="checkbox"/> afiseaza fisiere ascunse <input type="checkbox"/> afiseaza extensia		
Pig Bird Cat Dog Rabbit Pig	JComboBox	O lista desfasurabila ce permite selectia unui singur element. Optional, componenta poate permite editarea manuala a valorii (de unde si numele de <i>combo box</i> - o combinatie intre drop-down list si textfield). Lista poate contine atat text cat si imagini.
Rosu Galben Albastru Maro	JList	O lista ce permite selectie multipla. Afisarea elementelor poate fi efectuata pe una sau mai multe coloane. Lista poate contine atat text cat si imagini.

	JMenu, JMenuBar, JMenuItem, JRadioButtonMenuItem, JCheckBoxMenuItem, JSeparator	<p>O fereastra poate avea bara de menu (JMenuBar) ce contine meniu (JMenu), care la randul lor pot contine alte meniu, elemente de meniu (JMenuItem si derivatele) sau linii separate (JSeparator)</p>
	JRadioButton	<p>Grupuri de butoane in care unul singur poate fi selectat la un moment dat. Atunci cand utilizatorul selecteaza un alt buton, cel precedent se deselecteaza automat.</p>
	JSlider	<p>Echivalentul unui potentiometru cu reglare liniara: utilizatorul modifica valoarea interna a componentei prin mutarea cursorului</p>
	JSpinner	<p>Componenta ce permite parcurgerea unei succesiuni prestabilite de valori sau imagini folosind sagetile sus-jos (ex: liste de elemente, date calendaristice etc)</p>
	JTextField	<p>Camp ce permite introducerea si editarea de text pe o singura linie (inclusiv operatii de selectie de text si cut/copy/paste)</p>
	JPasswordField	<p>Un JTextField care ascunde caracterele scrise de utilizator</p>
	JFormattedTextField	<p>un JTextField care impune un anumit format al informatiei introduse (validare in momentul scrierii). Se observa caracterele obligatorii - , deja prezente in camp</p>
	JTextArea	<p>Componenta ce permite introducerea si editarea de text neformatat pe mai multe linii, inclusiv operatii de cut/copy/paste (un mini-Notepad)</p>
	JTable	<p>Componenta pentru afisarea tabelara de informatie, ce permite si editarea in-place a informatiei (stil Microsoft Excel).</p>
	JTree	<p>Componenta ce permite afisarea de informatii cu structura ierarhica (ex: resurse din sistemul de fisiere)</p>
	JProgressBar	<p>Componenta utilizata pentru evidențierea vizuala a evoluției unei operații</p>
	JSplitPane	<p>Panel ce contine doua zone despartite de un separator mobil. Fiecare zona poate contine o componenta. Cand utilizatorul muta separatorul dimensiunile componentelor din cele doua parti vor fi ajustate automat</p>
	JTabbedPane	<p>Panel ce poate afisa alternativ mai multe interfete (pagini) pe aceeasi suprafață. Alegerea paginii se realizeaza prin click pe tab-ul corespunzator sau actiunea echivalenta (ex: combinatii de taste)</p>



6.7. ANEXA 2 - Elaborarea unei interfete grafice: un exemplu

Construim interfata din figura alaturata, care reprezinta fereastra de introducere a unei persoane intr-o lista. Cerintele sunt urmatoarele:

- etichetele de nume si prenume sa fie aliniate la marginea stanga
- cele doua text field-uri sa aiba aceeasi dimensiune si sa fie aliniate
- atunci cand utilizatorul redimensioneaza fereastra:
 - lista sa preia tot spatiul liber aparut pe orizontala verticala
 - butoanele sa ramana centrate pe orizontala fata de marginile ferestrei si sa-si pastreze distanta dintre ele
 - formularul (etichetele+textfield-urile) sa ramana centrate pe verticala in spatiul delimitat intre zona de butoane si marginea de sus a ferestrei



si

Etape parcurse:

- divizam interfata in subcontainere. Avand in vedere ca atat zona de formular, cat si cea de butoane au propriile reguli interne, dedicam cate un panel fiecareia dintre ele. Fie *pForm* panel-ul pentru formular si *pButoane* cel care contine butoanele
- alegem layout managerul pentru toate containerele implicate:
 - pentru *pButoane* alegem *FlowLayout*. Aceasta are atat optiunea de centrare, cat si de spatiere a componentelor
 - pentru *pForm* alegem *i*, impartit in 4 celule. Componentele nu vor umple celula si vor fi ancorate in marginea ei stanga, rezultand astfel alinierea dorita
 - pentru root container (sau mai exact pentru content pane-ul sau) alegem *GridBagLayout*. Aceasta va fi divizat in 4 celule care vor forma 3 zone - cele doua celule de sus, plus zona de jos care va ocupa ambele celule ale celei de-a doua linii
- scriem clasa, avand grija ca la fiecare adaugare de componenta in container sa specificam restrictiile necesare. Instantiem obiectele necesare, setam layout-ul pentru containere, adaugam componente in containere si in final facem containerul de baza vizibil

Iata implementarea rezultata:

```

import java.awt.*;
import javax.swing.*;

public class ImplementareGUI {

    public static void main(String[] args) {
        GridBagConstraints gridBagConstraints;

        // instantiere componente
        JFrame frame = new JFrame("Exemplu GUI");
        JPanel pForm = new JPanel();
        JPanel pButoane = new JPanel();
        JLabel lNume = new JLabel("Nume:");
        JLabel lPrenume = new JLabel("Prenume:");
        JTextField tfNume = new JTextField();
        JTextField tfPrenume = new JTextField();
        JButton bAdauga = new JButton("Adaugare");
        JButton bRenunta = new JButton("Renuntare");
        JList lista = new JList(new String[]{"Ionel Popescu"});

        // stabilire layout managere
    }
}

```

```

frame.getContentPane().setLayout(new GridBagLayout());
pForm.setLayout(new GridBagLayout());
pButoane.setLayout(new FlowLayout(FlowLayout.CENTER, 50, 5));

// ***** adaugare butoane in pButoane *****
pButoane.add(bAdauga);
pButoane.add(bRenunta);

// ***** adaugare componente in pForm *****
// - adaugare lNume
gridBagConstraints = new GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 0;
gridBagConstraints.anchor = GridBagConstraints.WEST;
gridBagConstraints.weightx = 1.0;
pForm.add(lNume, gridBagConstraints);

// - adaugare lPrenume
gridBagConstraints = new GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 1;
gridBagConstraints.anchor = GridBagConstraints.WEST;
gridBagConstraints.weightx = 1.0;
pForm.add(lPrenume, gridBagConstraints);

// - adaugare tfNume
tfNume.setColumns(40);
tfNume.setMinimumSize(new Dimension(100, 28));
tfNume.setPreferredSize(new Dimension(100, 28));
gridBagConstraints = new GridBagConstraints();
gridBagConstraints.gridx = 1;
gridBagConstraints.gridy = 0;
gridBagConstraints.anchor = GridBagConstraints.WEST;
gridBagConstraints.weightx = 1.0;
pForm.add(tfNume, gridBagConstraints);

// - adaugare tfPrenume
tfPrenume.setColumns(40);
tfPrenume.setMinimumSize(new Dimension(100, 28));
tfPrenume.setPreferredSize(new Dimension(100, 28));
gridBagConstraints = new GridBagConstraints();
gridBagConstraints.gridx = 1;
gridBagConstraints.gridy = 1;
gridBagConstraints.anchor = GridBagConstraints.WEST;
gridBagConstraints.weightx = 1.0;
pForm.add(tfPrenume, gridBagConstraints);

// ***** adaugare containere intermediare in frame *****
// - adaugare pForm
pForm.setMinimumSize(new Dimension(100, 28));
pForm.setPreferredSize(new Dimension(100, 28));
gridBagConstraints = new GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 0;
gridBagConstraints.fill = GridBagConstraints.BOTH;
gridBagConstraints.ipadx = 100;
gridBagConstraints.ipady = 100;
gridBagConstraints.anchor = GridBagConstraints.NORTHWEST;
frame.getContentPane().add(pForm, gridBagConstraints);

// - adaugare pButoane
pButoane.setMinimumSize(new Dimension(100, 28));
gridBagConstraints = new GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 1;
gridBagConstraints.gridwidth = 2;
gridBagConstraints.fill = GridBagConstraints.HORIZONTAL;
gridBagConstraints.anchor = GridBagConstraints.NORTHWEST;

```

```
gridBagConstraints.insets = new Insets(10, 0, 0, 10);
frame.getContentPane().add(pButoane, gridBagConstraints);

// - adaugare lista
lista.setMinimumSize(new Dimension(150, 28));
lista.setPreferredSize(new Dimension(150, 28));
gridBagConstraints = new GridBagConstraints();
gridBagConstraints.fill = GridBagConstraints.BOTH;
gridBagConstraints.gridx = 1;
gridBagConstraints.gridy = 0;
gridBagConstraints.weightx = 1.0;
gridBagConstraints.weighty = 1.0;
frame.getContentPane().add(lista, gridBagConstraints);

// ***** cusotmizare si afisare root container *****
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.pack();
frame.setVisible(true);
}

}
```