

Graficzny Interfejs Użytkownika – GUI

Większość współcześnie pisanych aplikacji zawiera graficzny interfejs (Graphical User Interface). Biblioteki umożliwiające tworzenie interfejsów graficznych oraz obsługujące interakcję z użytkownikiem są wbudowane w systemy operacyjne lub są standardowym składnikiem systemu (Windows, XWindows, Motif.)

Interfejs graficzny konstruowany jest ze standardowych komponentów oferowanych przez środowisko (okna, pola edycyjne, przyciski, listy). Niektóre środowiska pozwalają także na tworzenie własnych wyspecjalizowanych komponentów i ich wielokrotne wykorzystanie, np.: formanty Active X.

Zadaniem twórcy interfejsu jest:

- ustalenie ich wzajemnego układu komponentów
- nadanie im parametrów początkowych (np.: tekstów pojawiających się na przyciskach, dodanie elementów do list, konfiguracja parametrów graficznych)
- napisanie kodu, który będzie uruchamiany w wyniku akcji użytkownika (np.: naciśnięcie przycisku, wybranie opcji z menu)

Biblioteki obsługujące GUI w języku Java

W momencie, kiedy powstawał język Java w każdym systemie operacyjnym istniały już zaawansowane biblioteki obsługujące GUI. Twórcy języka zdecydowali, że stworzą przenośną bibliotekę, która będzie abstrakcją dla istniejących środowisk graficznych.

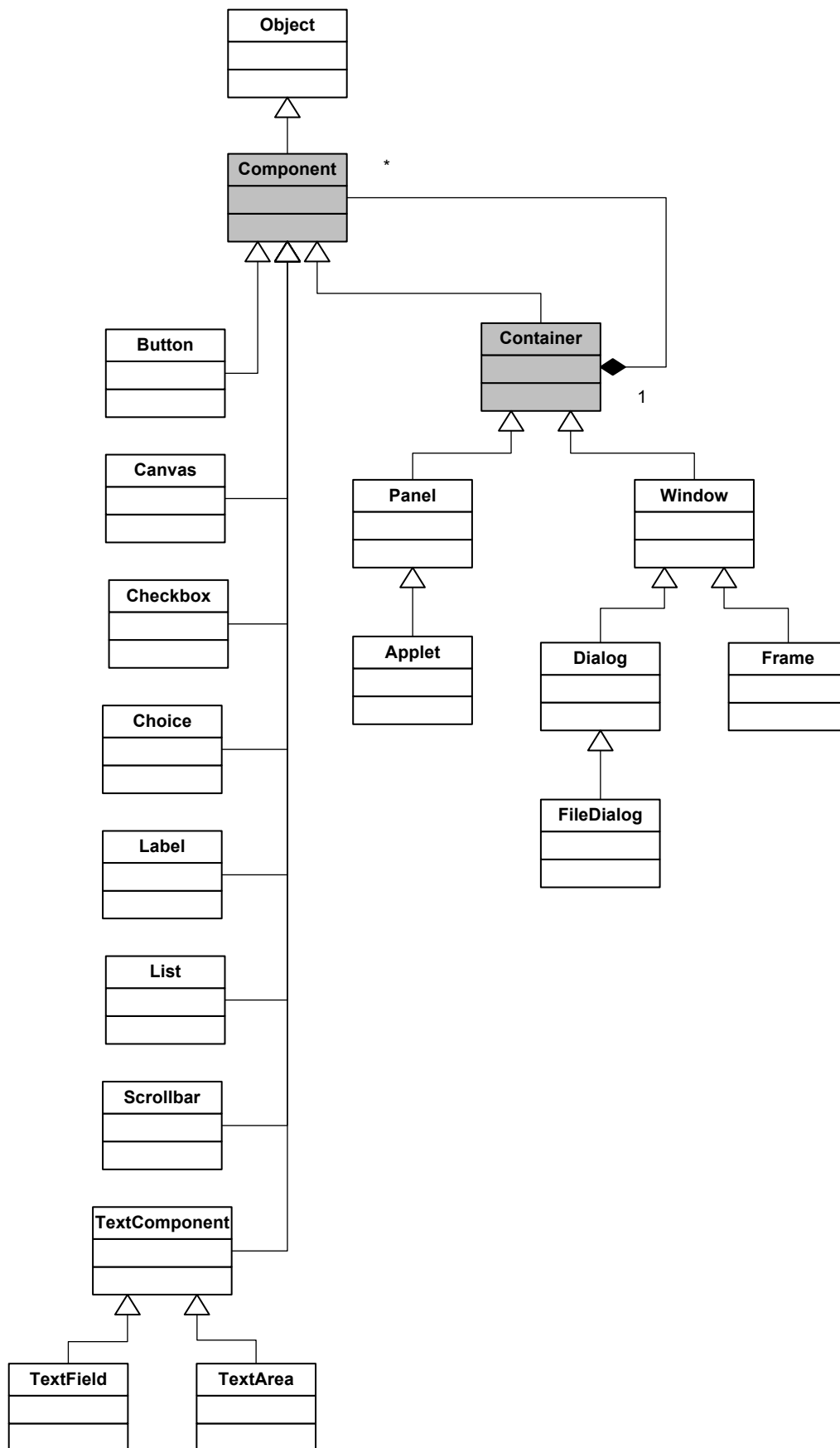
Biblioteka AWT (Abstract Windowing Toolkit)

- Definiuje zbiór standardowych komponentów pojawiających się w interfejsie użytkownika. Kod komponentów dla danej platformy jest implementowany za pomocą języka innego niż Java z użyciem bibliotek specyficznych dla danej platformy.
- Zarządza układem komponentów
- Definiuje podstawowy model obsługi zdarzeń
- Pozwala na oprogramowanie grafiki wektorowej, wyświetlanie obrazów i użycie kilku podstawowych czcionek

Kolejnym etapem było powstanie JFC (Java Foundation Class) obejmujących:

- **AWT**
- **Swing** (nowa równoległa hierarchia obiektów napisana całkowicie w języku Java, zapewniająca pełną przenośność)
- Bibliotekę **Java2D** (obsługa fontów, zaawansowana grafika wektorowa, transformacje afiniczne)
- Technologię **Drag and Drop**
- **Accessibility** API (ułatwienia dostępu – czytniki ekranu, wyświetlacze Braila)
- **Internacjonalizację** -- wsparcie dla aplikacji wielojęzycznych

Hierarchia komponentów AWT



Dodatkowo klasy Component i Frame implementują interfejs MenuContainer.

Komponenty

Każda komponent GUI pojawiający się w aplikacji dziedziczy po klasie `Component`.

W klasie umieszczono szereg standardowych metod:

- zarządzanie rozmiarami:
`pack()`, `doLayout()`, `invalidate()`, `getBounds()`,
`getHeight()`, `getLocation()`, `setBounds()`,
`setSize()`
- rysowanie:
`repaint()`, `paint(Graphics g)`,
`update(Graphics g)`, `paintAll(Graphics g)`
- zarządzanie czcionkami i kolorami:
`getFont()`, `setFont()`, `setBackground()`,
`getBackground()`
- ustalanie odbiorcy zdarzeń generowanych w komponencie:
`addKeyListener()`, `addMouseListener()`,
`addMouseMotionListener()`

Kontenery

Kontenery są komponentami, które mogą zawierać inne komponenty.

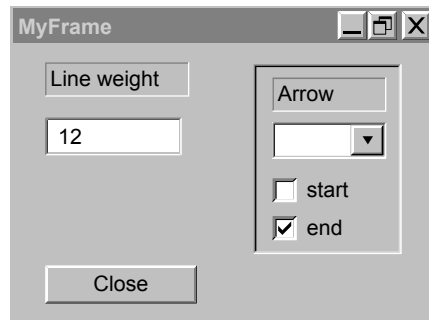
Referencje komponentów dodawanych do kontenera są przechowywane na liście. Kolejność elementów na liście określa kolejność wyświetlania i przetwarzania komponentów.

- Metoda `add(Component c)` dodaje komponent na końcu listy kontenera. Przeciążona wersja z dodatkowym parametrem określającym indeks pozwala na dodanie komponentu w określonym miejscu listy.
- Komponent może uzyskać dostęp do kontenera wywołując metodę `getParent()`.
- Kontener może uzyskać dostęp do swoich komponentów za pomocą metod `getComponentCount()` oraz `getComponent(int n)`.

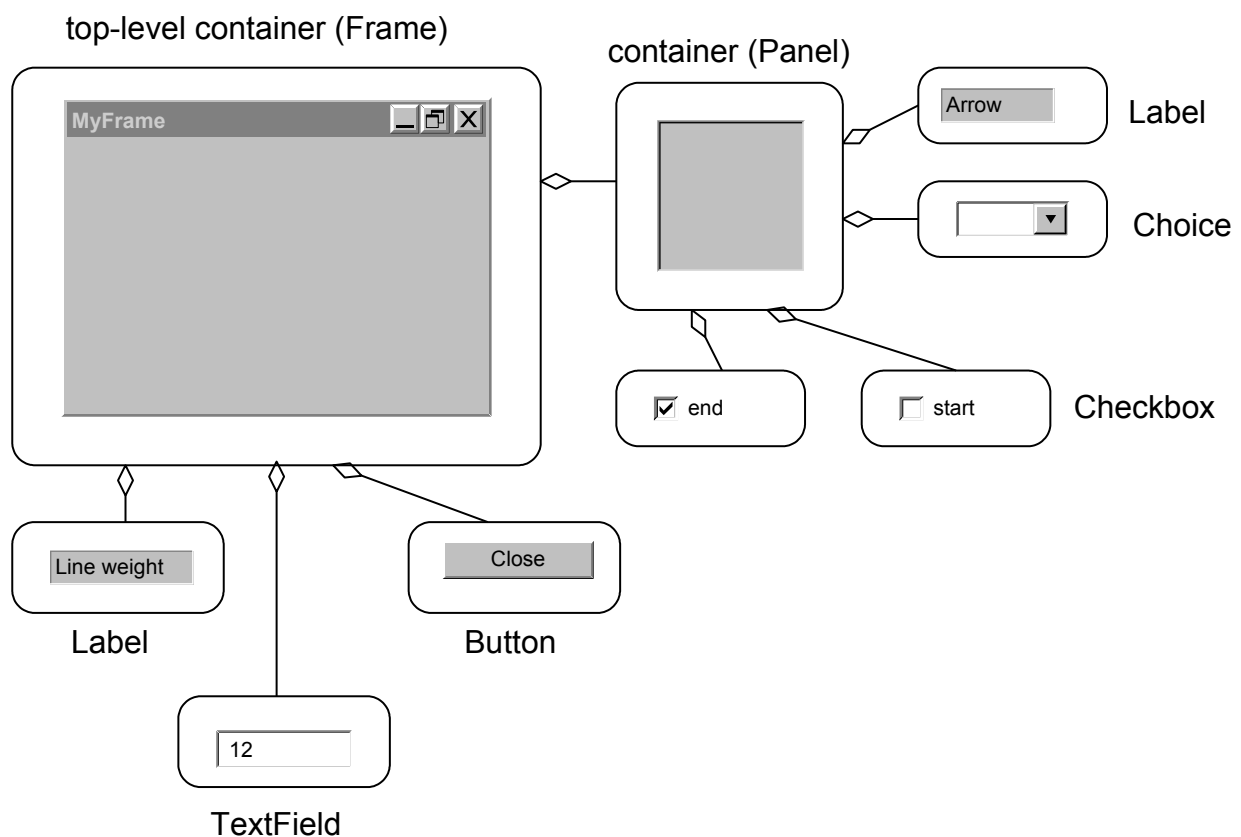
Budowa interfejsu graficznego

Graficzny interfejs użytkownika budowany jest hierarchicznie: do jednego z komponentów górnego poziomu: `Frame`, `Dialog` lub `Applet` dodawane są komponenty niższego poziomu. Kontener `Panel` umożliwia logiczne zgrupowanie elementów i lepsze zarządzanie układem komponentów na ekranie.

Przykład hierarchii komponentów



Okno z komponentami



Jego hierarchia komponentów

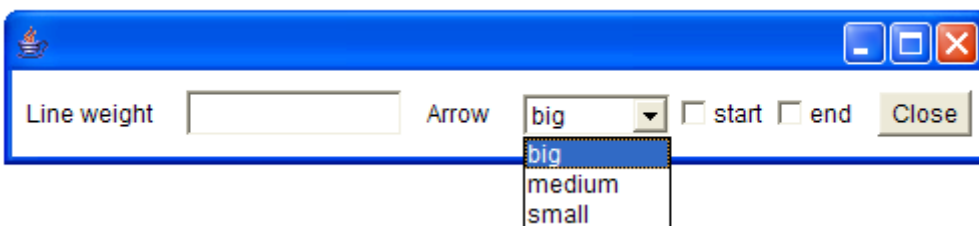
Kod konstruujący okno z komponentami

```
class MyFrame extends Frame
{
    MyFrame()
    {
        this.addWindowListener (new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });
        setLayout(new FlowLayout());

        add(new Label("Line weight"));
        add(new TextField(12));
        Panel p = new Panel();
        p.add(new Label("Arrow"));
        Choice c = new Choice();
        c.add("big");
        c.add("medium");
        c.add("small");
        p.add(c);
        p.add(new Checkbox("start"));
        p.add(new Checkbox("end"));
        add(p);
        add( new Button("Close"));
        pack();
    }
}
```

```
public static void main(String args[])
{
    MyFrame f = new MyFrame();
    f.setVisible(true); // show()
}
```

Rezultat (platforma Windows XP)



Rozmieszczenie komponentów

Projektując graficzny interfejs aplikacji zazwyczaj staramy się nadać komponentom określony układ (rozmiary i położenie wewnątrz kontenera).

- Dla wielu platform definicja komponentów i ich układ w kontenerze jest zdefiniowany *poza* kodem aplikacji – wewnątrz *zasobów*.

Zasoby zawierają pełne definicje zawartości okien dialogowych tekstów, a także innych danych binarnych, które mogą być wykorzystywane przez aplikację.

Zazwyczaj środowiska IDE pozwalają na graficzne projektowanie elementów interfejsu aplikacji i zapisują rezultaty w zasobach.

Położenie komponentów w zasobach określone jest jawnie, ale w jednostkach, które są określone względem rozmiarów czcionki przypisanej kontenerowi.

W momencie wyświetlania okna aplikacja odczytuje definicję komponentów z zasobów i automatycznie tworzy obiekty danej platformy (przyciski, pola edycyjne, itd.).

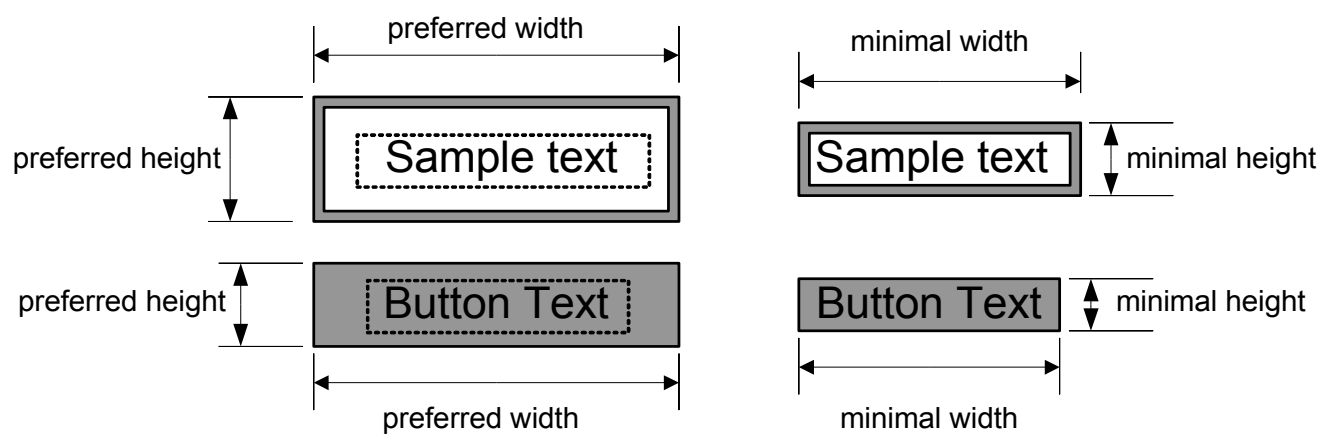
Rzeczywiste rozmiary i położenie okna kontenera i komponentów ustalane są na podstawie rozmiarów czcionki i rozdzielczości ekranu.

- W języku Java definiowanie komponentów wchodzących w skład kontenerów odbywa się jawnie. Nawet jeżeli stosowane są narzędzia typu *GUI Builder* pozwalające na wizualne projektowanie elementów interfejsu – wprowadzone zmiany odzwierciedlane są w kodzie.
- Środowisko IDE może tworzyć swoje pliki pomocnicze pełniące rolę zasobów. Są to automatycznie generowane pliki XML definiujące parametry komponentów kontenera (typu oczekiwane rozmiary) oraz atrybuty sterujące ich rozmieszczeniem.

Rozmiary komponentów (które w przypadku obiektów AWT są odwzorowane w rzeczywiste obiekty danej platformy) są również ustalane na podstawie fontu używanego przez komponent.

Biblioteka AWT posługuje się trzema zestawami danych definiującymi oczekiwane rozmiary komponentów:

- preferowane rozmiary (`getPreferredSize()`)
- minimalne rozmiary (`getMinimumSize()`)
- maksymalne rozmiary (`getMaximumSize()`)



Dane te należy traktować jako wskazówki dla biblioteki AWT określające w jaki sposób należy ustalić rozmiary komponentów na ekranie.

- Podczas rozmieszczania elementów interfejsu biblioteka AWT będzie starała się użyć preferowanych rozmiarów komponentów. Na tej podstawie określony zostanie rozmiar kontenera (jeżeli jego rozmiary nie są narzucone, jak w przypadku klasy `Applet`).
- Jeżeli rozmiary kontenera są wymuszone (np.: zmieniamy rozmiary okna), przyjęte zostaną rozmiary komponentów z zakresu `getMinimumSize()` do `getMaximumSize()` lub komponent zostanie ukryty.
- Programista implementujący własne komponenty może sterować ich rozmiarami przeddefiniowując metody określające oczekiwane rozmiary. Dla komponentów Swing istnieją także metody `setMinimumSize()`, `setPreferredSize()`, itd. Wartości mogą być zdefiniowane w kodzie lub w pliku zasobów.

Układ komponentów

Każdemu kontenerowi przypisany jest obiekt klasy implementującej interfejs `LayoutManager` (menadżer układu komponentów).

Steruje on rozmieszczeniem komponentów w kontenerze, kiedy zachodzi konieczność ustalenia ich układu:

- w momencie, kiedy kończymy dodawanie elementów do kontenera i wywoływana jest metoda `pack()`
- po zmianie rozmiarów okna
- w wyniku wywołania metody `validate()`

Menadżer układu kontenera może być ustawiony za pomocą metody `setLayout()`.

Klasy menadżerów układu komponentów:

- `FlowLayout` (standardowo przypisany kontenerom `Panel` i `Applet`)
- `BorderLayout` (standardowo przypisany kontenerom `Window`, `Frame` i `Dialog`)
- `GridLayout`
- `GridBagLayout`
- `CardLayout`
- `BoxLayout` (dodane w pakiecie `javax.swing`)
- `SpringLayout` (dodane w pakiecie `javax.swing`)
- `GroupLayout` (dodane w pakiecie `javax.swing`)

Równoczesne użycie kontenerów pośrednich (klasy `Panel`) i różnych klas `LayoutManager` pozwala na osiągnięcie złożonych układów komponentów – porównywalnych z tymi, które można uzyskać, przez jawne pozycjonowanie elementów.

Koszt manualnego kodowania jest większy niż np.: w przypadku edycji zasobów. Z drugiej strony obecne środowiska IDE w pełni automatycznie generują kod – użytkownik musi jednak zdawać sobie sprawę, jakie elementy dodawać i jakich menadżerów użyć, aby uzyskać pożądane efekty.

FlowLayout

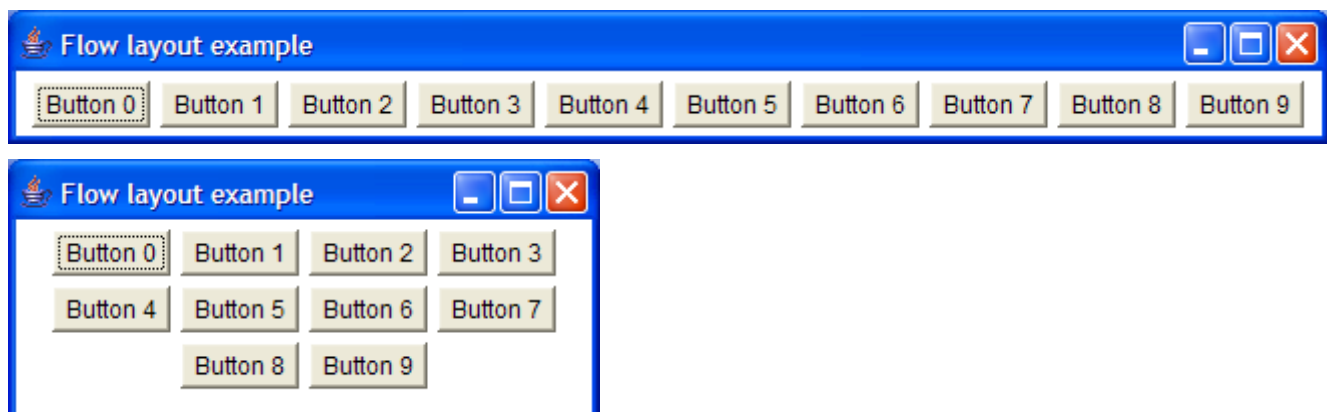
Jest to jeden z najprostszych układów – dodawane komponenty są rozmieszczane kolejno w wierszach. W momencie, kiedy wiersz przekraczałyby szerokość kontenera – rozpoczynany jest następny.

Układ `FlowLayout` respektuje preferowane rozmiary komponentów.

Przeciążone konstruktory klasy pozwalają na podanie sposobu justowania komponentów w wierszu (do środka, do lewej i do prawej) oraz odstępów pionowych i poziomych pomiędzy komponentami.

```
class FlowLayoutFrame extends Frame
{
    FlowLayoutFrame ()
    {
        //this.addWindowListener (...);

        setTitle("Flow layout example");
        setLayout(new FlowLayout());
        for(int i=0;i<10;i++)add(new Button("Button " + i));
        pack();
    }
}
```



- Menadżerem podobnym do `FlowLayout` jest `BoxLayout`. Pozwala on na rozmieszczanie komponentów wertykalnie lub horyzontalnie. Pomiedzy komponenty mogą być wstawione niewidoczne komponenty typu *glue* (klej) o zmiennych rozmiarach oraz *strut* (wzmocnienia) o ustalonych rozmiarach.

BorderLayout

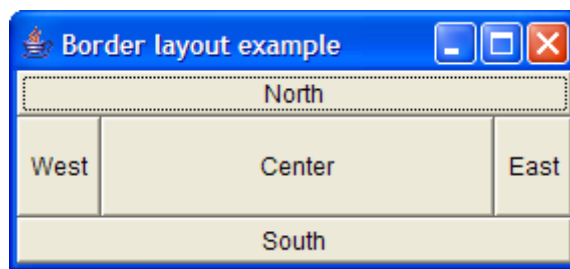
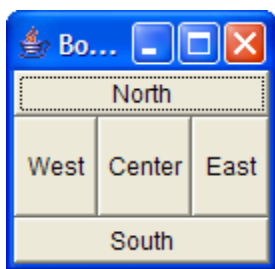
Menadżer BorderLayout rozmieszcza komponenty kontenera w pięciu obszarach rozmieszczonych wzdłuż brzegu kontenera (North, South, West, East) oraz w środku (Center).

Korzystając z menadżera układu należy dodając komponenty wyspecyfikować stałe tekstowe określające ich położenie – korzystamy z przeciążonej wersji metody:

`add(Component c, Object constraint).`

```
class BorderLayoutFrame extends Frame
{
    BorderLayoutFrame ()
    {
        //this.addWindowListener (...);

        setTitle("Border layout example");
        add(new Button("North"), BorderLayout.NORTH);
        add(new Button("South"), BorderLayout.SOUTH);
        add(new Button("Center"), BorderLayout.CENTER);
        add(new Button("West"), BorderLayout.WEST);
        add(new Button("East"), BorderLayout.EAST);
        pack();
    }
}
```



Menadżer zmienia rozmiary komponentów tak, aby zajmowały cały obszar kontenera. Podczas rozciągania w kierunku pionowym rozciągane są komponenty West, Center i East. Podczas rozciągania w kierunku poziomym North, Center oraz South.

GridLayout

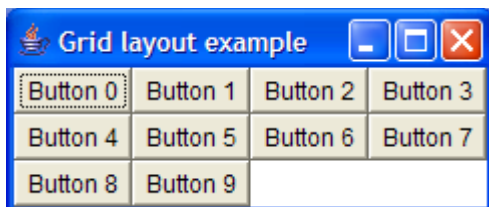
Menadżer układu `GridLayout` rozmieszcza komponenty w tablicy zawierającej komórki o tych samych rozmiarach.

Liczba wierszy i kolumn tablicy określana jest w konstruktorze klasy. Przeciążona wersja konstruktora pozwala także na podanie w pikselach odstępów pomiędzy wierszami i kolumnami.

Tablica w całości wypełnia kontener. Z kolei każdy z komponentów wypełnia całkowicie pojedynczą komórkę tablicy.

```
class GridLayoutFrame extends Frame
{
    GridLayoutFrame ()
    {
        //this.addWindowListener ();

        setTitle("Grid layout example");
        setLayout(new GridLayout(0,4));
        for(int i=0;i<10;i++)add(new Button("Button " + i));
        pack();
    }
}
```



Jeżeli rozmiar kontenera jest ustalany na podstawie rozmiarów komponentów – wówczas na podstawie największego komponentu ustalany jest rozmiar komórki tablicy i stąd rozmiar zawierającego ją kontenera.

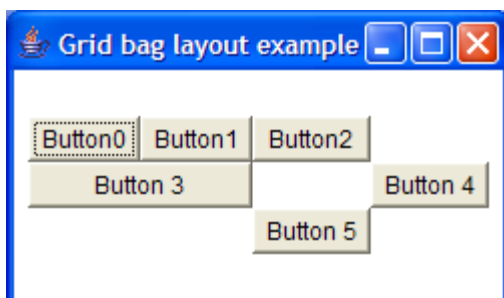
Jeżeli konieczna jest zmiana układu w wyniku zmiany rozmiarów kontenera, wówczas kontener narzuca wielkość komórki, co może prowadzić do nadania komponentom rozmiarów niezgodnych z parametrami *minimum size* i *maximum size*.

GridBagLayout

Układ GridBagLayout pozwala na rozmieszczenie komponentów w prostokątnych obszarach rozpiętych na siatce komórek o zmiennej szerokości wierszy i kolumn. Parametry sterujące rozmiarami i położeniem komponentu przekazywane są za pomocą obiektu klasy GridBagConstraints.

```
class GridBagLayoutFrame extends Frame {
    GridBagLayoutFrame() {
        //this.addWindowListener (...);
        setTitle("Grid bag layout example");
        setLayout(new GridBagLayout());
        GridBagConstraints c = new GridBagConstraints();
        c.fill = GridBagConstraints.HORIZONTAL; // wypelnienie komórek
        Button b;
        for(int i=0;i<3;i++){
            b = new Button("Button"+i);
            c.gridx=i;
            c.gridy=0; // pierwszy wiersz
            add(b,c);
        }

        b = new Button("Button 3");
        c.gridx=0;
        c.gridy=1;
        c.gridwidth=2; // podwójna szerokość
        add(b,c);
        b = new Button("Button 4");
        c.gridx=3; // czwarta kolumna
        c.gridy=1;
        c.gridwidth=1;
        add(b,c);
        b = new Button("Button 5");
        c.gridx=2;
        c.gridy=2; // trzeci wiersz, trzecia kolumna
        add(b,c);
        pack();
    }
}
```



CardLayout

Układ CardLayout umieszczenie w kontenerze zbioru *kart*. W danym momencie widoczna jest tylko jedna karta. Klasa zapewnia metody pozwalające na przełączanie kart – `first()`, `last()`, `next()`, `previous()`. Zazwyczaj konieczna jest implementacja przycisków sterujących wyborem kart.

```
class CardLayoutFrame extends Frame {
    Panel cp=new Panel();
    CardLayout cl = new CardLayout();
    CardLayoutFrame() {
        //this.addWindowListener (...);

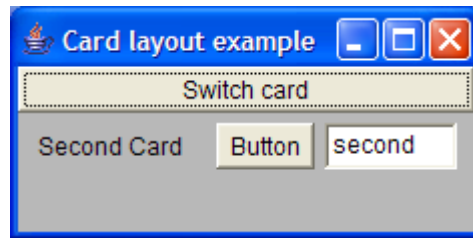
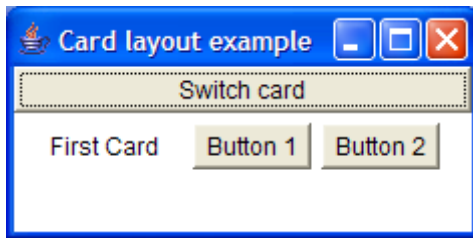
        setTitle("Card layout example");

        Button b= new Button("Switch card");
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                cl.next(cp);
            }
        });
        add(b, "North");

        cp.setLayout(cl);
        Panel p = new Panel(); // pierwsza karta
        p.add(new Label("First Card" ) );
        p.add(new Button("Button 1"));
        p.add(new Button("Button 2"));
        cp.add("Card 1",p);

        p = new Panel(); // druga karta
        p.setBackground(new Color(184,184,184));
        p.add(new Label("Second Card"));
        p.add(new Button("Button"));
        p.add(new TextField("second"));
        cp.add("Card 2",p);

        add(cp, "Center");
        pack();
    }
}
```



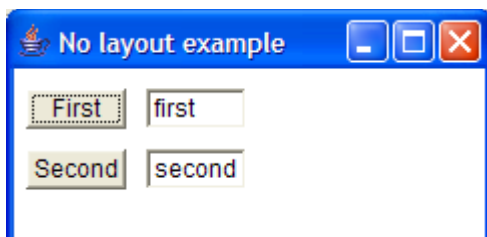
Brak menadżera układu

Jeżeli zrezygnujemy z zastosowania menadżera układu – wywołując `setLayout(null)` – wówczas położenie i rozmiary każdego elementu muszą zostać ustalone jawnie.

Przykład

```
class NoLayoutFrame extends Frame
{
    NoLayoutFrame()
    {
        // this.addWindowListener (...);
        setTitle("No layout example");
        setLayout(null);

        Button b = new Button("First");
        b.setBounds(10,40, 50,20);
        add(b);
        TextField t = new TextField("first");
        t.setBounds(70,40, 50,20);
        add(t);
        b = new Button("Second");
        b.setBounds(10,70, 50,20);
        add(b);
        t = new TextField("second");
        t.setBounds(70,70, 50,20);
        add(t);
        setBounds(0,0, 100,100);
    }
}
```



Obsługa zdarzeń w bibliotece AWT

Użytkownik aplikacji graficznej obsługuje ją za pośrednictwem urządzeń wejściowych – np.: myszy lub klawiatury. W wyniku akcji użytkownika takiej, jak kliknięcie na komponent, ruch myszy, naciśnięcie klawiszy generowane są zdarzenia.

- Zdarzeniem jest obiekt, który przekazuje informację, że „coś się stało”, np.: użytkownik wykonał pewną akcję.
- Źródłem zdarzenia jest komponent, który został uaktywniony w wyniku akcji użytkownika – przycisk, element menu, komponent, który zarejestrował ruch myszy, komponent, który oczekiwał na naciśnięcie klawiszy (miał „fokus”).
- Handlerem zdarzenia jest metoda (lub obiekt) , która otrzymuje informację o zdarzeniu i podejmuje odpowiednią akcję.

Hierarchiczny model obsługi zdarzeń był stosowany w pierwszej wersji biblioteki JDK 1.0.

Komponent, który był źródłem zdarzeń usiłował je obsłużyć (metody `handleEvent()` lub `action()`). Jeżeli zdarzenie mogło zostać obsłużone – handler zwracał wartość `true`.

W przeciwnym przypadku wołana była analogiczna metoda kontenera, który zawierał komponent. W ten sposób informacja o zdarzeniu propagowana była w górę hierarchii komponentów, aż do napotkania handlera, który ją obsłużył.

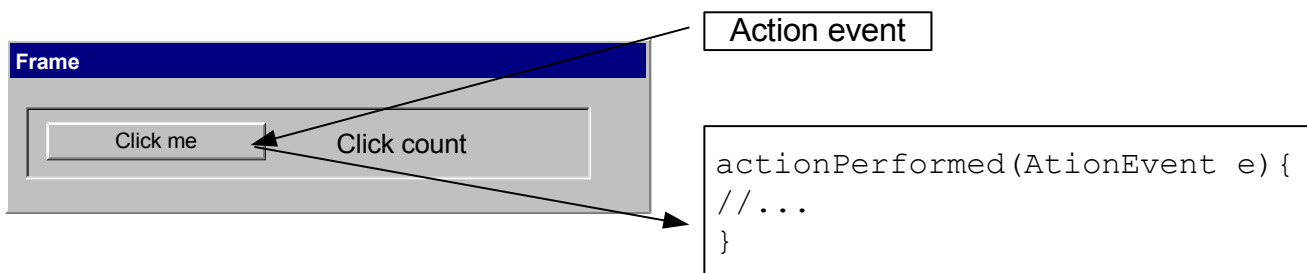
W praktyce w modelu hierarchicznym obsługa wielu zdarzeń odbywała się w jednej metodzie kontenera górnego poziomu. Dyskryminacja zdarzeń (który przycisk naciśnięto?) miała postać ciągu instrukcji warunkowych.

Dodatkowym problemem było rozmieszczeniem kodu standardowej obsługi zdarzeń na różnych poziomach hierarchii dziedziczenia. (Dla dwóch komponentów reagujących analogicznie na to samo zdarzenie konieczna była osobna implementacja kodu.)

Delegacyjny model obsługi zdarzeń pojawił się w JDK 1.1.

Nowy model całkowicie oddziela komponenty GUI od kodu obsługi zdarzeń.

Informacja o pojawieniu się zdarzenia nie trafia do funkcji, ale bezpośrednio do obiektu, który jest zarejestrowany, jako odbiorca (ang. *listener*) określonych zdarzeń generowanych przez dane źródło. Odbiorca „nasłuchuje”, czy zdarzenie pojawiło się. Po pojawieniu się zdarzenia wołana jest jego metoda, w której umieszczony jest kod obsługi.



- Zdarzenie – opisuje akcję użytkownika: naciśnięcie przycisku.
- Źródłem zdarzeń jest komponent `Button`.
- Handlerem zdarzenia jest dowolny obiekt implementujący metodę `actionPerformed()`.

Danemu źródłu zdarzeń może być przypisanych kilku odbiorców. Będą oni otrzymywali kolejno informację o zdarzeniu i wykonywali swoje procedury obsługi. Na przykład w programie graficznym odbiorcą zdarzenia informującego o ruchu myszy może być obiekt, który rysuje na ekranie oraz obiekt, który wyświetla współrzędne kursora graficznego.

W modelu delegacyjnym każdemu typowi zdarzeń `XXXEvent` przypisano interfejs `XXXListener`, definiujący metody uaktywniane w wyniku pojawienia się zdarzeń danego typu.

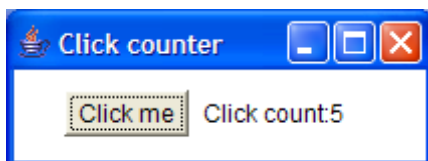
Klasa zainteresowana odbiorem zdarzeń danego typu implementuje odpowiedni interfejs i rejestruje się w komponencie będącym źródłem zdarzeń za pomocą metody `addXXXListener()`.

Przykład

```
class ClickCounter extends Frame implements ActionListener
{
    Label l = new Label("Click count: 0");
    int clickCount=0;
    ClickCounter()
    {
        //this.addWindowListener (...);

        setTitle("Click counter");
        setLayout(new FlowLayout());
        Panel p = new Panel();
        Button b= new Button("Click me");
        b.addActionListener(this);
        b.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Someone clicked me");
            }
        });

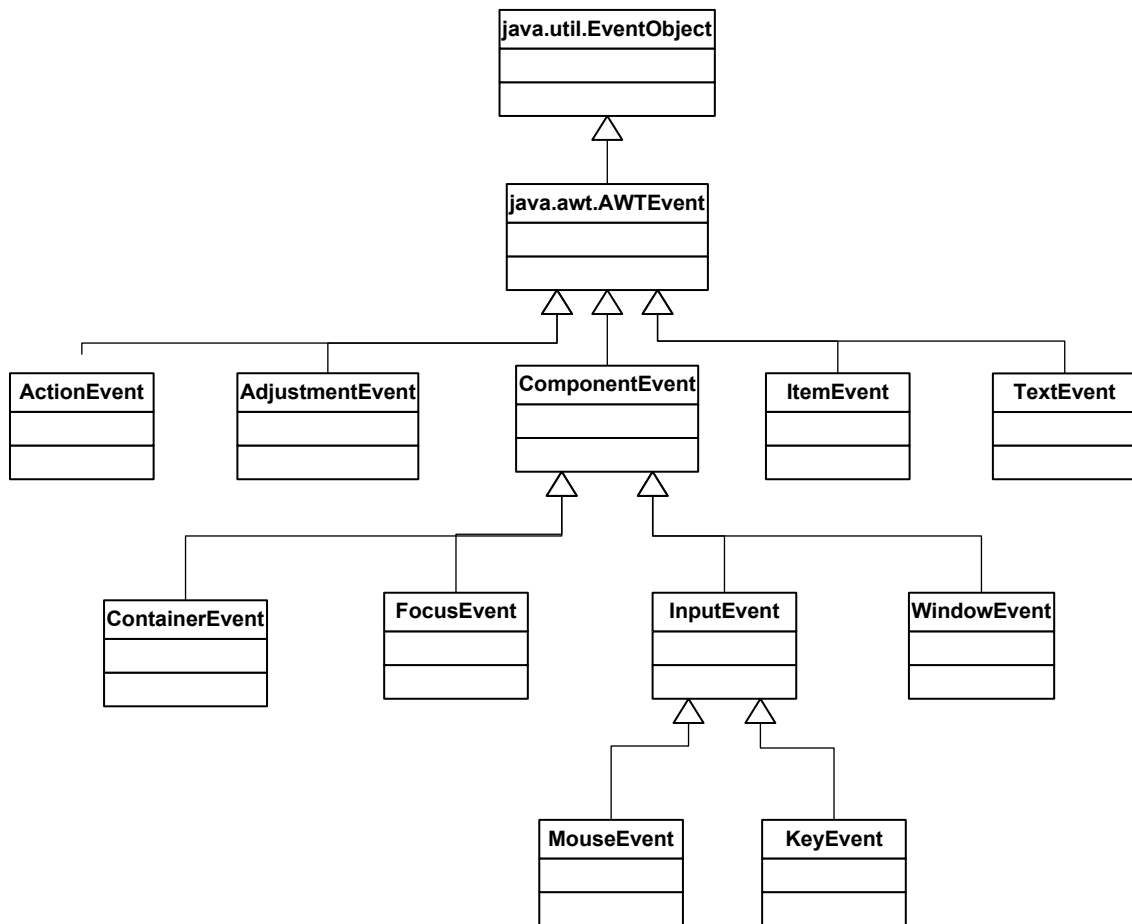
        p.add(b);
        p.add(l);
        add(p);
        pack();
    }
    public void actionPerformed(ActionEvent e)
    {
        clickCount++;
        l.setText("Click count:"+clickCount);
    }
}
```



- Przycisk jest źródłem zdarzeń typu `ActionEvent`.
- Interfejs `ActionListener` definiuje metodę `actionPerformed` przeznaczoną do obsługi tego typu zdarzeń.
- Klasa `ClickCounter` rejestruje się jako odbiorca zdarzeń generowanych przez przycisk – `addActionListener(this)`.

Hierarchia klas reprezentujących zdarzenia

Klasy reprezentujące zdarzenia zdefiniowane są w pakiecie `java.awt.event`.



`ActionEvent` – zdarzenie wysokiego poziomu oznaczające akcję użytkownika polegającą na uaktywnieniu komponentu (przycisku, elementu menu) za pomocą zdarzenia niższego poziomu (`InputEvent`).

`AdjustmentEvent` – zdarzenia generowane przez komponenty typu suwak.

`ItemEvent` – element listy, `Checkbox`, `Choice` został wybrany lub zmienił stan

`TextEvent` – zmienił się tekst przypisany komponentowi tekstowemu

`WindowEvent` – zdarzenie oznaczające zmianę stanu okna

`FocusEvent` – zdarzenie komunikujące utratę lub uzyskanie fokusu (zdolności przyjmowania znaków z klawiatury)

`ContainerEvent` – zdarzenie niskiego poziomu informujące o zmianie zbioru komponentów.

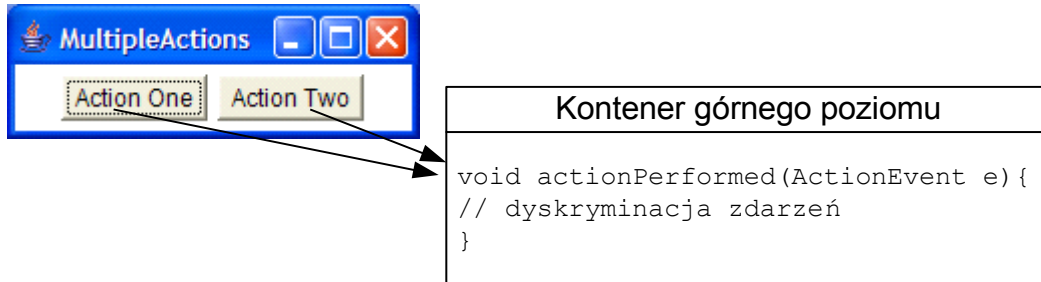
Kategoria zdarzeń	Nazwa interfejsu	Metody interfejsu
Akcja użytkownika	ActionListener	actionPerformed (ActionEvent)
Zmiana stanu elementu	ItemListener	itemStateChanged (ItemEvent)
Ruch myszy	MouseEventListener	mouseMoved (MouseEvent) mouseDragged (MouseEvent)
Inne zdarzenia generowane przez mysz	MouseListener	mousePressed (MouseEvent) mouseReleased (MouseEvent) mouseClicked (MouseEvent) mouseEntered (MouseEvent) mouseExited (MouseEvent)
Klawiatura	KeyListener	keyPressed (KeyEvent) keyReleased (KeyEvent) keyTyped (KeyEvent)
Fokus	FocusListener	focusGained (FocusEvent) focusLost (FocusEvent)
Okno	WindowListener	windowClosing (WindowEvent) windowOpened (WindowEvent) windowIconified (WindowEvent) windowDeiconified (WindowEvent) windowClosed (WindowEvent) windowActivated (WindowEvent) windowDeactivated (WindowEvent)
Komponent tekstowy	TextListener	textValueChanged (TextEvent)

Jak widać, interfejsy zapewniają wstępną dyskryminację zdarzeń. Na przykład zdarzenia `WindowEvent` oznaczające zmianę stanu okna są klasyfikowane i trafiają do odpowiednich funkcji.

Gdzie umieszczać klasy obsługujące zdarzenia

Kontener górnego poziomu Handler obsługi zdarzeń może być umieszczony w kontenerze górnego poziomu. Problemem jest sytuacja, gdy zachodzi konieczność obsługi zdarzeń tego samego typu generowanych przez różne komponenty interfejsu.

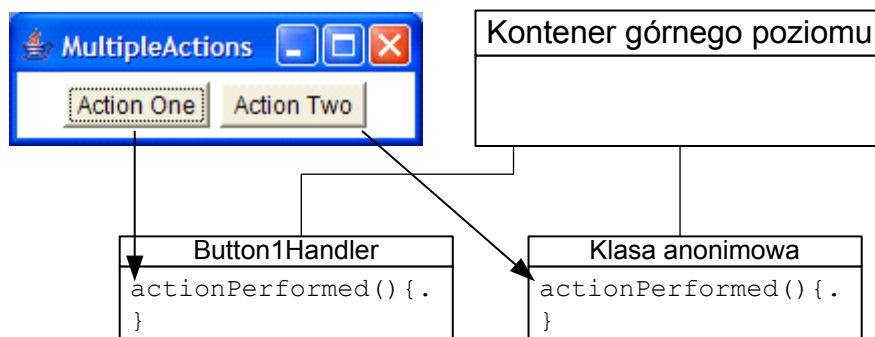
Konieczna jest wówczas dyskryminacja zdarzeń w postaci ciągu instrukcji warunkowych.



```
class MultipleActions extends Frame  
implements ActionListener  
{  
    MultipleActions()  
    {  
        //this.addWindowListener (...);  
  
        setTitle("MultipleActions");  
        setLayout(new FlowLayout());  
        Button b= new Button("Action One");  
        b.setActionCommand("ACTION1");  
        b.addActionListener(this);  
        add(b);  
        b= new Button("Action Two");  
        b.setActionCommand("ACTION2");  
        b.addActionListener(this);  
        add(b);  
        pack();  
    }  
    public void actionPerformed(ActionEvent e)  
    {  
        if(e.getActionCommand().equals("ACTION1"))  
            System.out.println("Action one");  
        if(e.getActionCommand().equals("ACTION2"))  
            System.out.println("Action two");  
    }  
}
```

Klasy wewnętrzne Dokumentacja JDK zaleca umieszczanie kodu obsługi zdarzeń w klasach wewnętrznych – nazwanych lub anonimowych.

Klasy wewnętrzne posiadają pełny dostęp do pól i metod otaczającej instancji klasy zewnętrznej.



Klasy wewnętrzne

Równocześnie – zgodnie z przyjętymi dla AWT regułami projektowania – odbiorcą zdarzenia jest obiekt, a nie metoda.

```
class MultipleActions2 extends Frame
{
    class Button1Handler implements ActionListener
    {
        public void actionPerformed(ActionEvent e) {
            System.out.println("Action one");
        }
    }

    MultipleActions2()
    {
        //this.addWindowListener (...);
        setTitle("MultipleActions2");
        setLayout(new FlowLayout());
        Button b= new Button("Action One");
        b.addActionListener(new Button1Handler());
        add(b);
        b= new Button("Action Two");
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Action two");
            }
        });
        add(b);
        pack();
    }
}
```

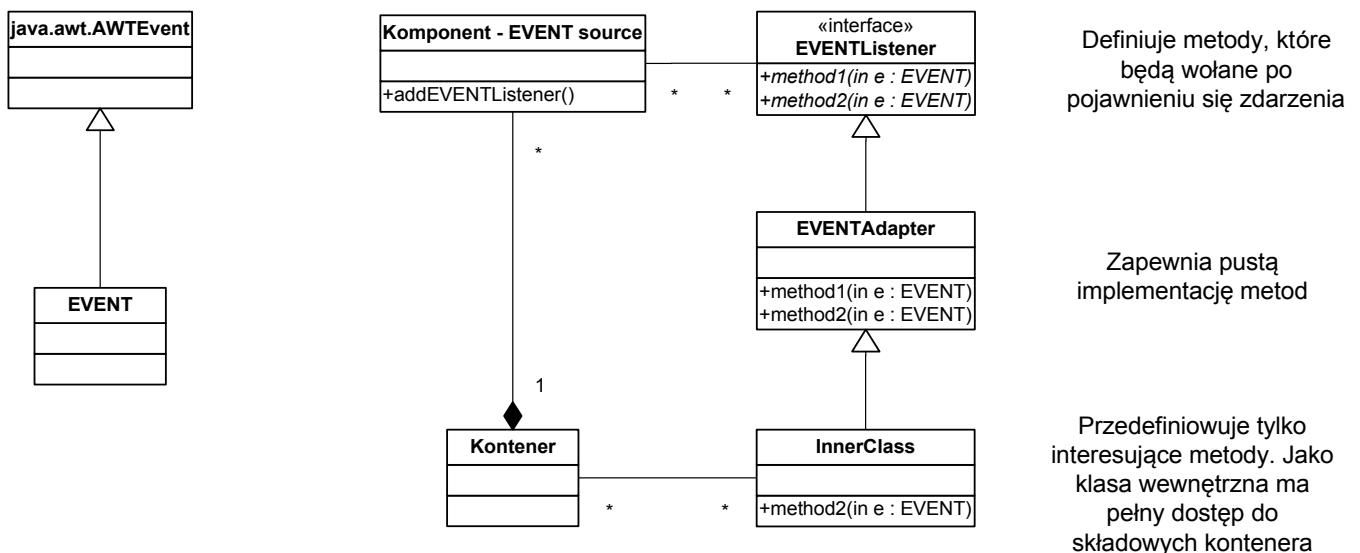
Adaptery

Interfejsy typu `Listener` definiują niejednokrotnie kilka metod (np.: `WindowListener`). W zależności od parametrów zdarzenia będzie wołana jedna z metod odbiorcy, np.:

- `windowClosing` po wybraniu przycisku zamykającego okno.
- `windowIconified` po wybraniu przycisku minimalizacji.

Część z tych zdarzeń (i metod) może być ignorowana przez aplikację, równocześnie formalne ograniczenia składniowe nakazują, aby konkretna klasa odbiorcy implementowała *pełny* interfejs.

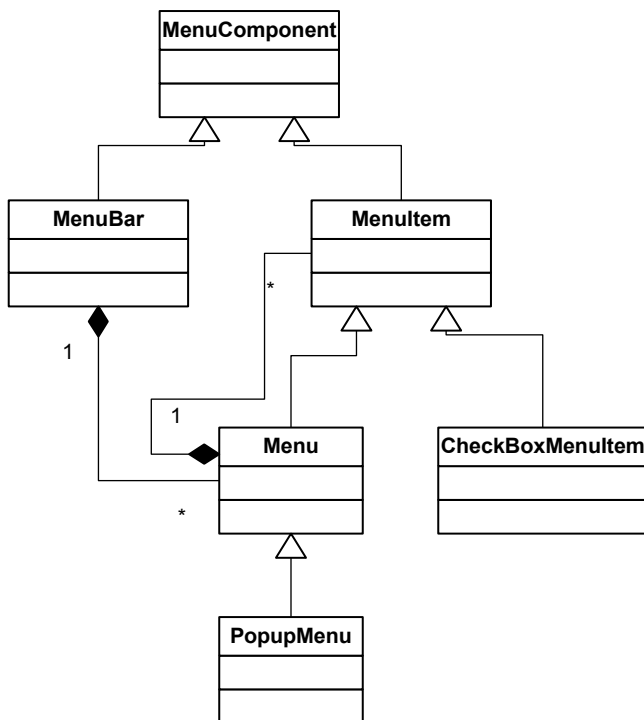
Aby nie zmuszać programistów do tworzenia pustych implementacji nieinteresujących metod, dla każdego interfejsu zdefiniowano klasę adaptera zapewniającą pustą implementację metod.



```
class MyFrame extends Frame {
    MyFrame() {
        this.addWindowListener (new WindowAdapter() {
            // tylko windowClosing, inne metody puste
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });
    }
}
```

Obsługa menu

Biblioteka AWT zawiera klasy umożliwiające tworzenie menu oraz generację zdarzeń w wyniku wybrania opcji menu.



MenuBar – kontener dla obiektów menu. Może być przypisany dla kontenera Frame za pomocą funkcji `setMenuBar()`.

MenuItem – element menu. Może być nim zwykły element lub menu niższego poziomu.

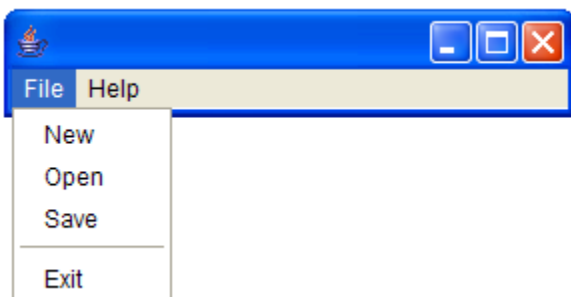
Menu jest kontenerem zawierającym elementy MenuItem.

PopupMenu jest szczególnym typem menu. Jest ono przewidziane do obsługi menu kontekstowego pojawiającego po naciśnięciu prawego klawisza myszy. Może być dodane do dowolnego komponentu za pomocą metody `add(PopupMenu m)`.

- Każdemu elementowi menu (MenuItem) jest przypisana etykieta (`label`) oraz polecenie (`command`). Standardowo polecenie jest tekstem (`String`) o takiej samej wartości, jak etykieta. Może być ono zmienione za pomocą metody `setActionCommand()`.
- Uaktywnienie elementu menu powoduje generację zdarzenia `ActionEvent` z atrybutem `command`. Zdarzenie to trafi do zarejestrowanego odbiorcy (typu `ActionListener`).
- Zaleca się, aby odbiorcami zdarzeń były obiekty wewnętrznych klas kontenera. Takie podejście stosowane jest w większości programów umożliwiających graficzne projektowanie interfejsu aplikacji.

Przykład

```
class FrameWithMenu extends Frame {  
    FrameWithMenu() {  
        //this.addWindowListener (...);  
        MenuBar mb = new MenuBar();  
        setMenuBar (mb);  
  
        Menu m = new Menu("File");  
        MenuItem mi=new MenuItem("New");  
        mi.addActionListener( new ActionListener(){  
            public void actionPerformed(ActionEvent e){  
                System.out.println("New selected");  
            } });  
        m.add(mi);  
        mi=new MenuItem("Open");  
        mi.addActionListener( new ActionListener(){  
            public void actionPerformed(ActionEvent e){  
                System.out.println("Open selected");  
            } });  
        m.add(mi);  
        mi=new MenuItem("Save");  
        mi.addActionListener( new ActionListener(){  
            public void actionPerformed(ActionEvent e){  
                System.out.println("Save selected");  
            } });  
        m.add(mi);  
        m.addSeparator();  
        mi=new MenuItem("Exit");  
        mi.addActionListener( new ActionListener(){  
            public void actionPerformed(ActionEvent e){  
                dispose();System.exit(0);  
            } });  
        m.add(mi);  
        mb.add(m);  
        m = new Menu("Help");  
        mb.add(m);  
  
        pack();  
    }  
}
```



Operacje graficzne

W momencie, kiedy komponent powinien zostać przerysowany wywoływana jest jego metoda `update(Graphics g)`. Metoda ta wypełnia obszar komponentu kolorem tła oraz wywołuje metodę `paint(Graphics g)`.

- Programowo można uaktualnić widok komponentu wywołując metodę `repaint()`.
- Kolor tła może zostać ustawiony za pomocą metody `setBackground(Color c)`.
- Wszelkie operacje graficzne zapewniające prawidłowe odtworzenie obrazu komponentu powinny zostać zaimplementowane w metodzie `paint(Graphics g)`.

Klasa `Graphics` reprezentuje *kontekst graficzny*. Może być nim:

- prostokątny fragment ekranu,
- pamięć – jeżeli operacje graficzne są przeprowadzane na utworzonym w pamięci obrazie,
- urządzenie wyjściowe typu drukarka.

Klasa `Graphics` zawiera zbiór metod umożliwiających rysowanie wektorów (linii, linii łamanych, prostokątów, wieloboków, łuków, elips, tekstów) oraz obrazów. Brak metody typu `putPixel()`!

Klasa przechowuje także informacje o stanie kontekstu graficznego – translację układu współrzędnych i obcinanie, bieżący kolor, bieżąca czcionka, sposób ustawiania koloru pikseli – kopiowanie koloru lub operacja XOR).

Typowy projekt komponentu graficznego składa się z trzech elementów:

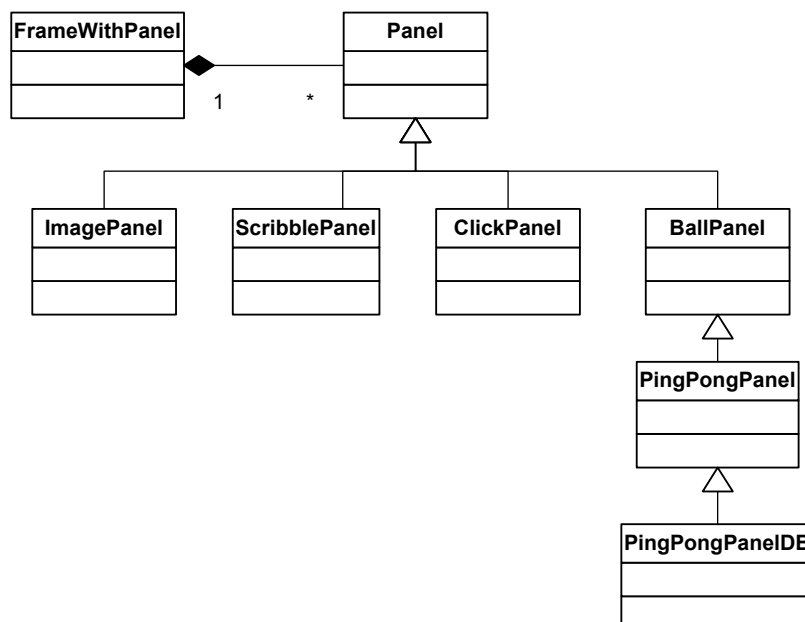
- modelu (ang. *model*) czyli zbioru danych podlegających wizualizacji
- widoku (ang. *view*) czyli kodu, który wizualizuje dane zawarte w modelu w określony sposób.
- kontrolera (ang. *controller*) – kodu zapewniającego reakcję na zdarzenia generowane w otoczeniu

Komponent reagując na zewnętrzne zdarzenia może modyfikować dane zdefiniowane w modelu i uaktualniać widok.

Źródłem zdarzeń jest w terminologii UML *aktor*, najczęściej jest to użytkownik, ale też może być to np.: wątek odpowiedzialny za animację, komunikaty odebrane za pośrednictwem połączenia sieciowego, itp.

Poniżej zostaną przedstawione kilka przykładów, w których do komponentu typu `Frame` (`FrameWithPanel`) zostaną dodane komponenty dziedziczące po klasie `Panel`:

- `ImagePanel` – komponent wyświetla załadowane z dysku obrazy JPEG
- `ScribblePanel` – komponent pozwala na rysowanie długiej linii łamanej
- `ClickPanel` – komponent w miejscu kliknięcia dodaje koła; kliknięcie na koło zmienia jego stan.
- `BallPanel`, `PingPongPanel`, `PingPongPanelDB` – pokazują przykład animacji.



Kod okna aplikacji

```
class FrameWithPanel extends Frame {
    FrameWithPanel()
    {
        this.addWindowListener (new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });
        setTitle("Frame With ImagePanel");
        add( new ImagePanel());
        //add( new XXXPanel());
        pack();
    }
}
```

Komponent `ImagePanel`

Wyświetla jeden z dwóch załadowanych obrazów JPEG. Kliknięcie na komponent powoduje zmianę stanu modelu – wybierany jest drugi z obrazów i wyświetlany w obszarze komponentu.

Obrazy są skalowane tak, aby wypełnić cały komponent.

```
class ImagePanel extends Panel
{
    Image[] img = {
        Toolkit.getDefaultToolkit().getImage("bird1.jpg"),
        Toolkit.getDefaultToolkit().getImage("bird2.jpg")
    };
    int currentImage=0;
    ImagePanel() {
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                currentImage = (currentImage+1)%img.length;
                repaint(); // przerysuj
            }
        });
    }

    public void paint(Graphics g)
    {
        Dimension d = getSize();
        g.drawImage(img[currentImage], 0, 0, d.width, d.height, this);
    }
}
```



Komponent ScribblePanel

Jego modelem są współrzędne dla linii łamanej oraz liczba punktów kontrolnych linii.

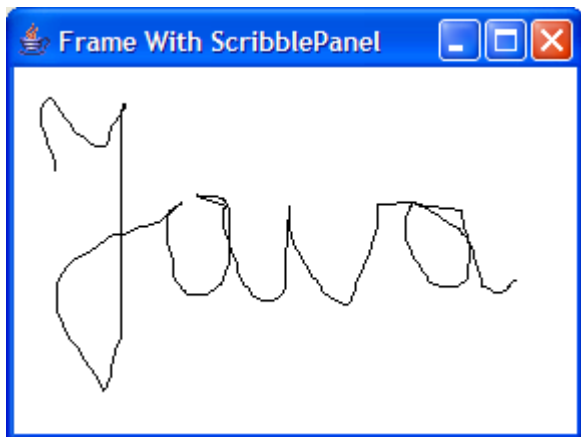
Po naciśnięciu przycisku myszy lub podczas ruchu myszy z naciśniętym przyciskiem do linii dodawane są nowe odcinki.

Klawisz ESC czyści linię. Klawisz BACKSPACE usuwa ostatni odcinek linii.

```
class ScribblePanel extends Panel
{
    int[] xpts = new int[100000];
    int[] ypts = new int[100000];
    int count=0;
    void addPoint(int x, int y){
        if(count==xpts.length) return;
        if(count<0) count=0;
        xpts[count]=x;
        ypts[count]=y;
        count++;
    }

    ScribblePanel(){
        addMouseListener(new MouseAdapter(){
            public void mousePressed(MouseEvent e){
                addPoint(e.getX(),e.getY());
                repaint();
            }
        });
        addMouseMotionListener(new MouseMotionAdapter(){
            public void mouseDragged(MouseEvent e){
                addPoint(e.getX(),e.getY());
                repaint();
            }
        });
        addKeyListener(new KeyAdapter(){
            public void keyPressed(KeyEvent e){
                if(e.getKeyCode()==KeyEvent.VK_ESCAPE) count=0;
                if(e.getKeyCode()==KeyEvent.VK_BACK_SPACE) count--;
                repaint();
            }
        });
    }
}
```

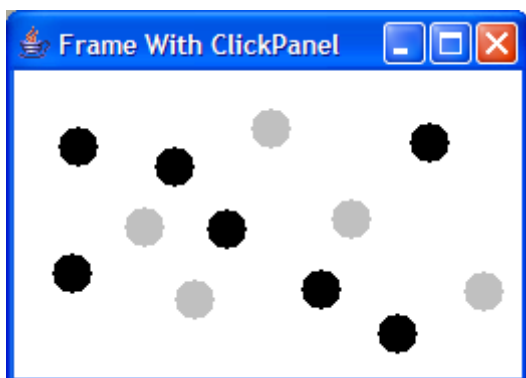
```
public void paint(Graphics g)
{
    if(count>0)g.drawPolyline(xpts,ypts,count);
}
}
```



Komponent `clickPanel`

Jego modelem jest zbiór obiektów klasy `Marker`. Obiekt `Marker` jest wizualizowany jako koło w środku współrzędnych (x,y) o promieniu `radius`. Jeżeli flaga `selected` ustawiona jest na `true` okrąg wyświetlany jest w kolorze szarym, w przeciwnym wypadku w kolorze czarnym. Wyświetlanie następuje w metodzie `void draw(Graphics g)`.

Jeżeli klikniemy myszą na pusty obszar do modelu dodawany jest nowy obiekt `Marker`; jeżeli klikniemy na obiekt `Marker`, jego flaga `selected` jest zmieniana na przeciwną.



```

class ClickPanel extends Panel
{
    static class Marker{
        int x,y;
        int radius=10;
        boolean selected=false;
        void draw(Graphics g){
            if(selected)g.setColor(Color.lightGray);
            else g.setColor(Color.black);
            g.fillOval(x-radius,y-radius,2*radius,2*radius);
        }
    }
    Marker[] m = new Marker[1000];
    int count=0;
    void addMarker(int x, int y){
        if(count==m.length) return;
        m[count]=new Marker();
        m[count].x=x;
        m[count].y=y;
        count++;
    }
    Marker getAt(int x,int y){
        for(int i=0;i<count;i++){
            if( (x-m[i].x)*(x-m[i].x)+(y-m[i].y)*(y-m[i].y)
                <=m[i].radius*m[i].radius ) return m[i];
        }
        return null;
    }

    ClickPanel(){
        addMouseListener(new MouseAdapter(){
            public void mousePressed(MouseEvent e){
                Marker mk= getAt(e.getX(),e.getY());
                if(mk!=null)mk.selected=!mk.selected;
                else addMarker(e.getX(),e.getY());
                repaint();
            }
        });
    }
    public void paint(Graphics g) {
        for(int i=0;i<count;i++){
            m[i].draw(g);
        }
    }
}

```


Komponent `BallPanel`

Komponent wyświetla piłkę odbijającą się od ścian. Jego modelem jest położenie środka piłki i jej prędkość.

Piłka jest animowana przez wątek (klasę wewnętrzną `BallThread`).

```
class BallPanel extends Panel
{
    double ballX,ballY;
    double ballVx,ballVy;
    double speed=15;
    static final int BALL_SIZE=10;
    int sleepTime=66;

    Color white=new Color(255,255,255);
    Color black=new Color(0,0,0);

    public void paint(Graphics g)
    {
        g.setColor(white);
        g.fillOval( (int)ballX-BALL_SIZE,
                  (int)ballY-BALL_SIZE,
                  2*BALL_SIZE, 2*BALL_SIZE);
        g.setColor(black);
        g.drawOval( (int)ballX-BALL_SIZE,
                  (int)ballY-BALL_SIZE,
                  2*BALL_SIZE, 2*BALL_SIZE);
    }

    public Dimension getPreferredSize() {
        return new Dimension(300,300);
    }

    class BallThread extends Thread {...}
    void detectCollision(){...}

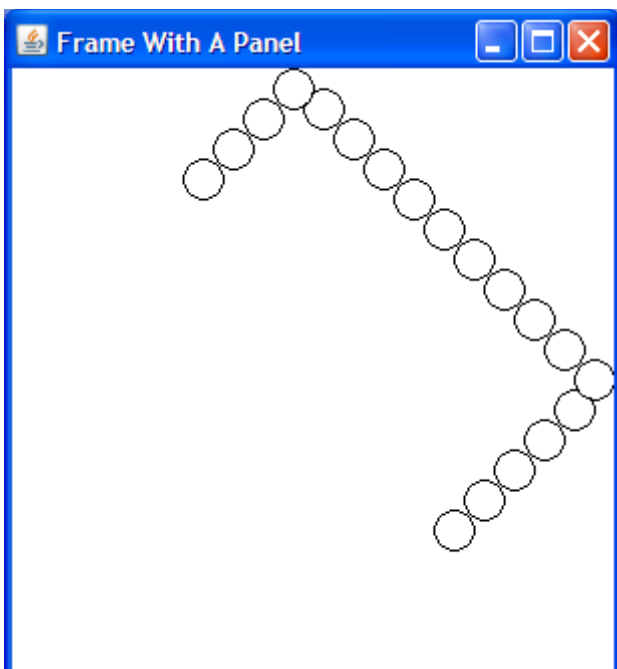
    BallPanel()
    {
        ballVx=1;
        ballVy=1;
        ballX=BALL_SIZE;
        ballY=150;

        new BallThread().start();
    }
}
```

Kod wątku odpowiedzialnego za animację

```
class BallThread extends Thread
{
    public void run()
    {
        for(;;){
            ❶ try{sleep(sleepTime);}
              catch(InterruptedException e){}
            ❷ ballX+=speed*ballVx;
              ballY+=speed*ballVy;
              detectCollision();
            ❸ repaint();
        }
    }
}
```

```
void detectCollision()
{
    Dimension d=getSize();
    if(ballX<BALL_SIZE){ballX=BALL_SIZE;ballVx*=-1;}
    if(ballX>d.width-BALL_SIZE){
        ballX=d.width-BALL_SIZE;ballVx*=-1;}
    if(ballY<BALL_SIZE){ballY=BALL_SIZE;ballVy*=-1;}
    if(ballY>d.height-BALL_SIZE){
        ballY=d.height-BALL_SIZE;ballVy*=-1;}
}
```



Komponent `PingPongPanel`

Komponent dziedziczy po nadklasie animację ruchu piłki. Dodatkowo wyświetlana jest na ekranie raketka przesuwana klawiszami kursora. Model jest rozszerzony o współrzędne środka raketki. Zdefiniowane są odpowiednio metody `paint()` oraz `detectCollision()`.

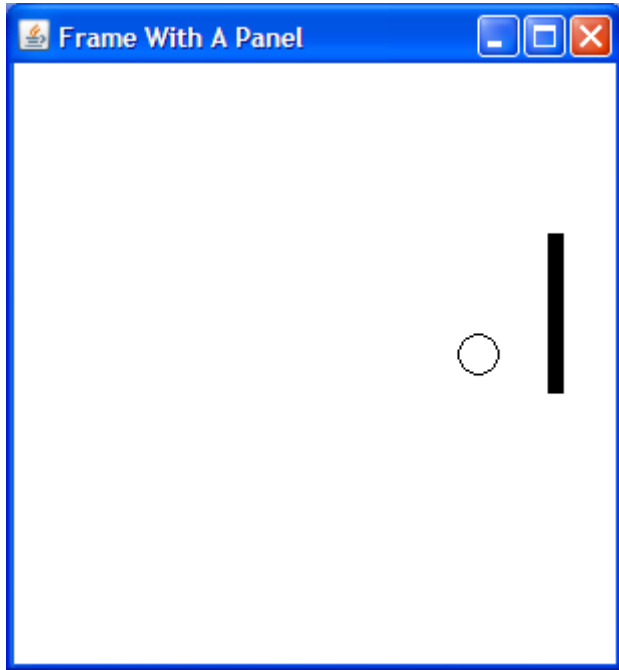
```
class PingPongPanel extends BallPanel
{
    int racketX=270;
    int racketY=150;
    static final int RSIZEX=4;
    static final int RSIZEY=40;

    PingPongPanel() {
        setFocusable(true);
        addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e) {
                if(e.getKeyCode() == KeyEvent.VK_UP) {
                    racketY-=25;repaint();}
                if(e.getKeyCode() == KeyEvent.VK_DOWN) {
                    racketY+=25;repaint();}
            }
        });
    }

    public void paint(Graphics g) {
        g.setColor(black);
        g.fillRect(racketX-RSIZEX, racketY-RSIZEY,
            2*RSIZEX, 2*RSIZEY);
        super.paint(g);
    }

    void detectCollision() {
        super.detectCollision();
        if(Math.abs(ballX-racketX)<BALL_SIZE+RSIZEX &&
            Math.abs(ballY-racketY)<RSIZEY)
        {
            ballX+=ballX-racketX + (BALL_SIZE+RSIZEX) *
                Math.signum(ballX-racketX);
            ballVx=-ballVx;
            repaint();
        }
    }
}
```

Okno aplikacji



Komponent `PingPongPanelDB`

Komponent jest najbardziej zaawansowaną wersją gry:

- Dodane jest wyświetlanie tła; jednakże tło nie powinno być wyświetlane w metodzie `paint()`, ale `update()`. Standardowo, metoda `update()` czyści tło komponentu i woła metodę `paint()`. Przerysowanie tła (bitmapa lub wypełnienie kolorem) w metodzie `paint()` wywołałoby efekt migotania przy każdej operacji odświeżania ekranu.
- Zmiany w modelu (położenie piłki) zachodzą znacznie częściej – wątek odpowiedzialny za animację jest usypiany na krótszy czas, ale ekran jest przerysowywany zgodnie z założoną częstotliwością ramek.
- Obraz jest generowany w buforze obrazu, a następnie za pomocą pojedynczego wywołania metody `drawImage()` jego zawartość jest wyświetlana na ekranie. Ta technika jest zalecana w przypadku konieczności animacji wielu elementów.

```

class PingPongPanelDB extends PingPongPanel
{
    Image bufferImage=null;
    long lastPaint=0;
    static final int frameRate=40;
    Color green=new Color(0,255,0);

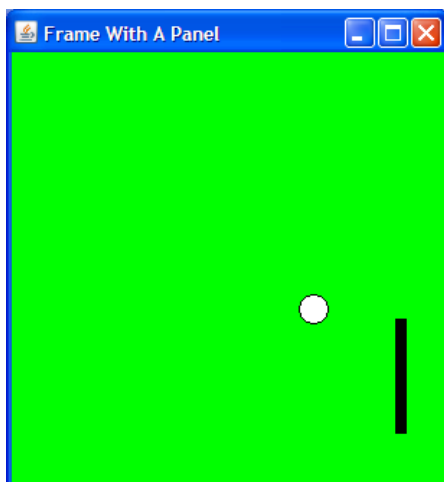
    PingPongPanelDB()
    {
        sleepTime=1;
        speed=.66;
    }

    public void update(Graphics g) {
        if(System.currentTimeMillis()-lastPaint<frameRate)
            return;
        Dimension d=getSize();
        if(bufferImage==null ||
            bufferImage.getHeight(this)!=d.height ||
            bufferImage.getWidth(this)!=d.width)
        {
            bufferImage=createImage(d.width, d.height);
        }

        Graphics bufferGraphics=bufferImage.getGraphics();
        bufferGraphics.setColor(green);
        bufferGraphics.fillRect(0, 0, d.width, d.height);
        super.paint(bufferGraphics);

        g.drawImage(bufferImage, 0, 0, this);
        lastPaint=System.currentTimeMillis();
    }
}

```



Swing

Swing stanowi kolejną generację biblioteki graficznej dla platformy Java.

- Zapewnia bogaty zbiór komponentów graficznych (JLabel, JButton, JCheckBox, JComboBox, JTree, JTextField, JTextArea, itd.)
- Został wyposażony w możliwość programowego zmieniania wyglądu aplikacji *PLAF—Pluggable Look And Feel*
- Możliwości graficzne zostały rozszerzone o Java 2D API
- Wsparcie dla internacjonalizacji

Swing przejął z AWT podstawowe zasady rozmieszczania komponentów z wykorzystaniem menadżerów układu oraz model obsługi zdarzeń. Podstawową różnicą pomiędzy komponentami Swing i AWT jest sposób implementacji komponentów

Lekkie komponenty

W AWT każdy z komponentów ma swój odpowiednik (*peer*) w okienkowym systemie platformy. W momencie, kiedy tworzymy obiekt AWT, na przykład przycisk lub pole edycyjne, system przesyła żądanie utworzenia odpowiedniego okna (kontrolki) do docelowego środowiska. Zmiana położenia lub rozmiarów komponentu, zmiana jego danych jest dokonywana przez przekierowanie do funkcji systemowej platformy. W ten sposób zachowany jest charakterystyczny dla danego środowiska wygląd (*look and feel*) i sposób działania komponentów.

W praktyce zachowanie i własności komponentów AWT pozostawały częściowo poza kontrolą biblioteki (docelowe środowisko definiowało własne ścieżki przesyłania i obsługi zdarzeń) i aplikacja napisana w AWT mogła w różny sposób zachowywać się w zależności od platformy.

Projektując Swing przyjęto, że większość komponentów zostanie całkowicie napisanych języku Java i nie będą powiązane z odpowiednikami (*peer*) docelowej platformy. Są to właśnie lekkie komponenty (*lightweight components*). Jedynymi ciężkimi komponentami są kontenery górnego poziomu: `JFrame`, `JDialog` i `JApplet`.

Lekkie komponenty są potomkami klasy `JComponent` (dziedziczącej po `java.awt.Container`). Za ich wygląd i zachowanie odpowiada kod w całości napisany w języku Java. Dzięki temu niezależnie od platformy reagują na zdarzenia w analogiczny sposób, zużywają mniej zasobów, działają szybciej, ponieważ wszelkie operacje odbywają się bez przekierowania do funkcji systemowych.

Dodatkowe zalety:

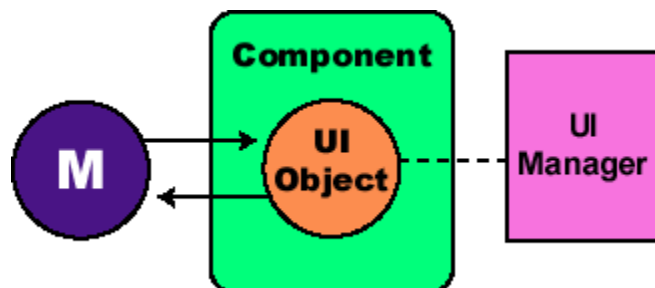
- zazwyczaj przy ich rysowaniu używa się techniki podwójnego buforowania, co pozwala uniknąć migotania
- mogą mieć inne kształty niż prostokątne
- mogą być przezroczyste
- pozwalają na programową zmianę wyglądu (*PLAF*)

Architektura komponentów Swing

Inspiracją dla architektury komponentów Swing był wzorzec MVC (*model widok kontroler*).

W architekturze komponentów Swing widok i kontroler zostały sprzężone w jeden obiekt nazywany *UI Delegate*, który skupia w sobie typowe zadania kontrolera i widoku oraz oddelegowuje część zadań widoku do obiektów specyficznych dla danej architektury *PLAF* dostarczanych przez klasę `UIManager`. Klasą bazową dla różnych klas *UI Delegate* jest klasa `ComponentUI`.

- *Model* przechowuje dane (ale też pełni część funkcji kontrolera, na przykład wysyła powiadomienia o zmianie w danych)
- *UI Delegate* jest odpowiedzialny za pobieranie danych z *Modelu* i ich wizualizację
- *Komponent* koordynuje działanie *Modelu* i *UI Delegate*, a także stanowi interfejs do funkcjonalności biblioteki AWT



Dodatkowym elementem architektury może być *Renderer*. Jest to klasa używana przez złożone komponenty (lista, lista rozwijana, drzewo, tabela) do wizualizacji danych składowych.

JComponent

`JComponent` jest klasą bazową dla wszystkich lekkich komponentów Swing. Zapewnia ona zestaw podstawowych funkcji:

- Podpowiedzi. Za pomocą funkcji `setToolTip()` można ustawić tekst, który będzie wyświetlany po zatrzymaniu kursora na komponencie.
- Wsparcie dla *PLAF*. Każdemu komponentowi jest przypisany obiekt *UI Delegate* (klasa potomna `ComponentUI`). W zależności od bieżących ustawień wyglądu, może zostać użyty jeden z obiektów *UI Delegate*. Dobór zależy od ustawienia dokonywanego w metodzie `UIManager.setLookAndFeel()`.
- Specyficzne własności. Każdemu komponentowi można przypisać dodatkowe dane w postaci par (*nazwa własności, obiekt*). Dostęp do własności może być dokonany z pomocą metod `putClientProperty()` i `getClientProperty()`. Własności mogą być używane na przykład przez menadżerów układu, aby zapewnić złożone pozycjonowanie komponentów.

- Ustalanie rozmiarów. W bibliotece AWT, aby ustalić minimalne, maksymalne i preferowane rozmiary komponentów, należy przedefiniować odpowiednie metody. W przypadku komponentów Swing zaimplementowane są funkcje: `setMinimumSize()`, `setMaximumSize()`, `setPreferredSize()`, `setAlignmentX()` oraz `setAlignmentY()`.
- Skróty klawiszowe. Komponentom można przypisać skróty klawiszowe równoważne ich aktywacji (elementy menu, przyciski)

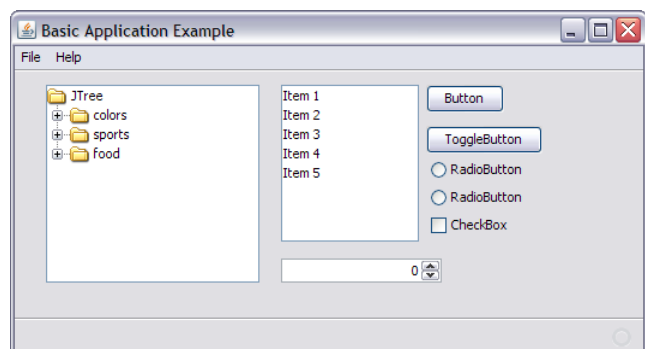
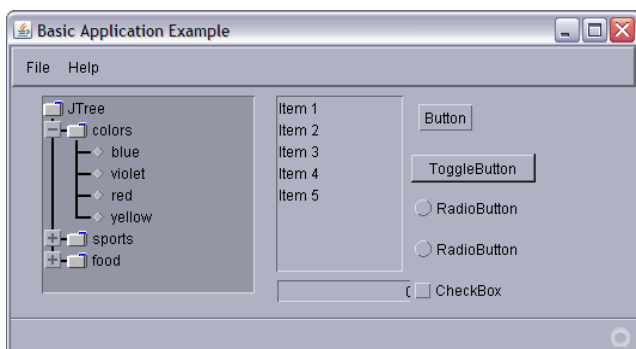
Look and Feel

Klasą odpowiedzialną za zmianę wyglądu elementów jest `UIManager`. Ustawienie wyglądu powinno być wykonane przed wyświetleniem okna aplikacji, za pomocą statycznej funkcji `setLookAndFeel()` klasy `UIManager`.

```

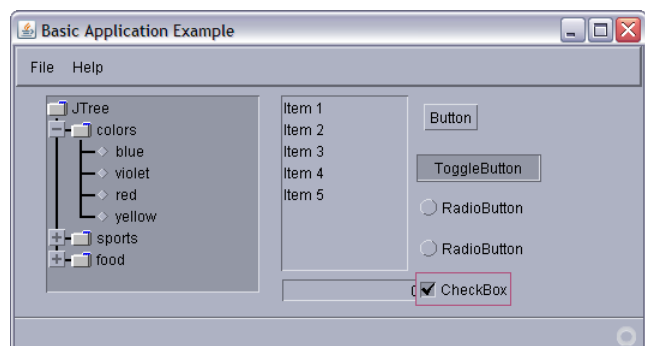
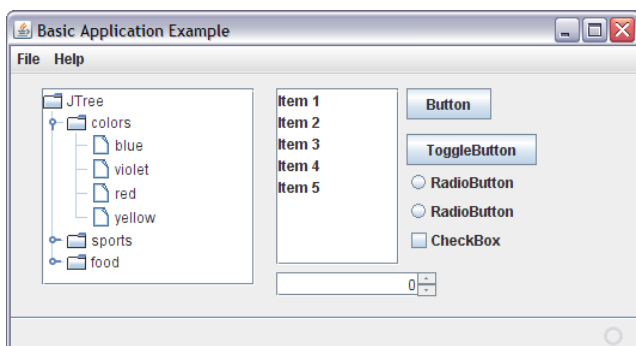
UIManager.setLookAndFeel(
    "com.sun.java.swing.plaf.motif.MotifLookAndFeel"
);

```



Java

System



Metal

Motif

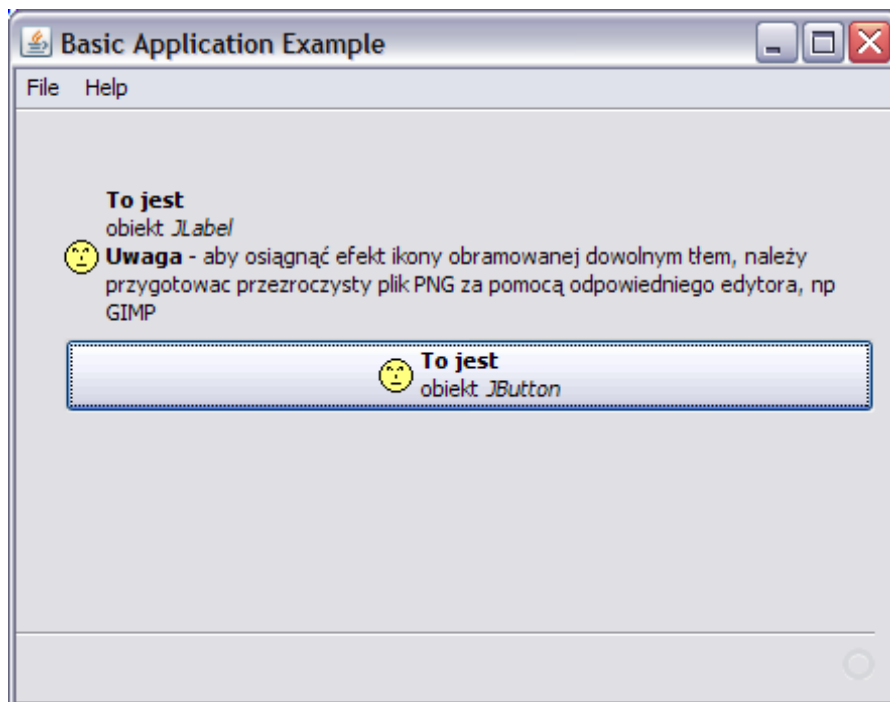
Prosty przykład – JButton i JLabel

Poniższy przykład ilustruje zastosowanie ikon i obsługę tekstu HTML.

```
ImageIcon ic = new ImageIcon(this.getClass().  
getResource("resources/face.PNG"));
```

```
labelControl.setIcon(ic);  
labelControl.setText("<html><b>To jest</b><p>obiekt  
<i>JLabel</i><p><b>Uwaga</b> - aby osiągnąć efekt ikony  
obramowanej dowolnym tłem, należy przygotować przezroczysty  
plik PNG za pomocą odpowiedniego edytora, np GIMP</html>");
```

```
buttonControl.setIcon(ic);  
buttonControl.setText("<html><b>To jest</b><p>obiekt  
<i>JButton</i></html>");
```



JComboBox

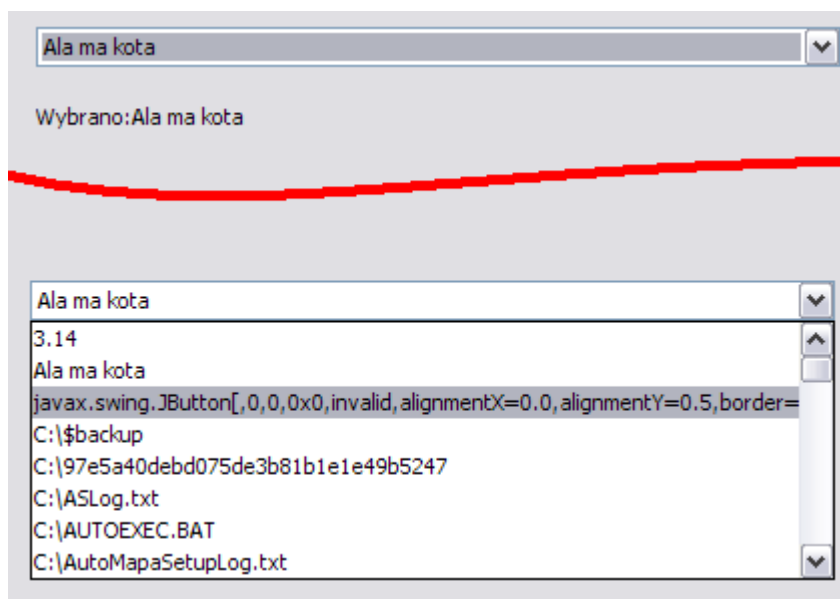
Komponent pozwala na wybór jednego lub więcej elementów z listy. Pozwala także na edycję elementu (ale, aby dodać element do listy konieczne jest dodanie własnego kodu).

Za pomocą metody `addItem()` można dodać obiekty do listy. Za pomocą metody `Object getSelectedItem()` odczytać, który element został wybrany.

Metoda `setSelectedItem(Object anObject)` pozwala programowo wybrać (i wyświetlić) wybrany obiekt.

```
comboBox.addItem(new Double(3.14));
comboBox.addItem("Ala ma kota");
comboBox.addItem(new JButton("Button"));

File[] files=new File("C:/").listFiles();
for(File f:files)comboBox.addItem(f);
comboBox.setSelectedItem("Ala ma kota");
```



Niezależnie od typów obiektów umieszczonych na liście, komponent wyświetla ich tekstową reprezentację konwertując zawartość obiektu za pomocą metody `toString()`. Jednakże struktury danych `JComboBox` (*model*) nie przechowują tekstów, ale obiekty, stąd możliwa jest manipulacja za pomocą referencji do obiektów, a nie tekstów (jak odbywa się na przykład w kontrolkach Windows).

Złożone komponenty Swing posługują się wbudowanym lub dostarczonym z zewnątrz kontenerem i na ogół nie ma potrzeby stosować równocześnie dwóch kontenerów i zarządzać ich spójnością (kontener tekstów komponentu i kontener obiektów).

Wywołanie `setSelectedItem("Ala ma kota")` wybiera obiekt posługując się referencją do elementu obecnego na liście (stałe łańcuchowe są reprezentowane przez unikalne instancje obiektów klasy `String`).

Powiadomienia

Zmianie stanu elementu listy spowodowanej akcją użytkownika lub dokonanej programowo towarzyszy wysłanie zdarzenia typu `ItemEvent`. Zdarzenie to można przechwycić implementując funkcję `itemStateChanged` zdefiniowaną w interfejsie `ItemListener`.

```
comboControl.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent evt) {
        label.setText("Wybrano:" + evt.getItem());
    }
});
```

Zmiana sposobu wizualizacji elementów

Komponenty typu listy, listy rozwijane, tabele, drzewa są zaimplementowane w większości bibliotek GUI. Zazwyczaj komponent górnego poziomu może być traktowany jak kontener zawierający elementy składowe. Kontener komunikuje się ze swoimi elementami za pomocą komunikatów – np.: z żądaniami podania rozmiarów elementu albo przerysowania zawartości.

Modyfikacja sposobu wyświetlania elementów polega na dostarczeniu przez programistę funkcji odpowiedzialnych za rysowanie (często na poziomie prymitywnych operacji graficznych).

Podejście klasyczne ma wiele wad:

- Protokół wymiany informacji pomiędzy złożonym komponentem i jego elementami jest z reguły skomplikowany
- Funkcje umożliwiające modyfikację sposobu wyświetlania elementów zawierają powtarzającą się funkcjonalność, podobną na przykład do rysowania etykiet z ikonami, a równocześnie wielokrotne wykorzystanie gotowego kodu (*reuse*) jest utrudnione

Twórcy biblioteki Swing założyli, że

- do wyświetlania elementów złożonych komponentów skorzystają z istniejącej funkcjonalności zawartej w klasie `JComponent` lub klasach potomnych;
- dla zoptymalizowania zużycia zasobów skorzystają ze wzorca projektowego Flyweight (*Renderer* w terminologii Swing).

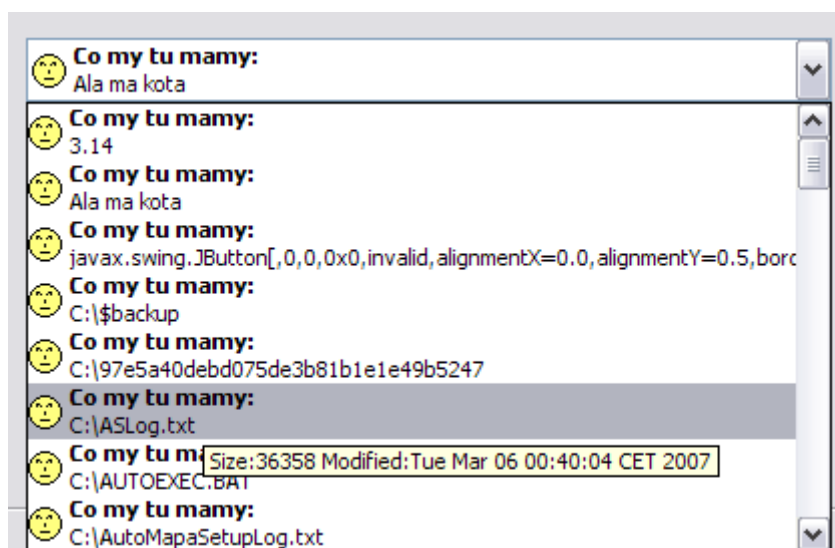
Wzorzec Flyweight

Wzorzec Flyweight polega na użyciu pojedynczego obiektu jako reprezentanta wielu obiektów. Kiedy oczekujemy od oryginalnego obiektu dostarczenia jakiś usług, w jego miejsce dostarczamy obiekt *Flyweight* modyfikując odpowiednio jego stan.

Zastosowanie wzorca pozwala na znaczną redukcję zużycia zasobów. Przykładem może być lista zawierająca kilka tysięcy elementów, z których widocznych jest 30. Lista mogłaby zostać zaimplementowana, jako olbrzymia kolekcja elementów `JLabel`. Znacznie bardziej efektywne jest jednak użycie pojedynczego obiektu `JLabel` i 30-krotne skonfigurowanie go w trakcie wyświetlania.

Wzorzec opisany, jako „Wykorzystanie współdzielenia dla efektywnej obsługi dużej liczby drobnoziarnistych obiektów” został skatalogowany w [Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995].

W bibliotece Swing występuje pod nazwą `Renderer` (`ListCellRenderer`, `TreeCellRenderer`, `TableCellRenderer`).



Przykład

```
class MyComboBoxRenderer extends JLabel
    implements ListCellRenderer {
public MyComboBoxRenderer() {
    setOpaque(true);
    setVerticalAlignment(CENTER);
}
ImageIcon icon = new ImageIcon(this.getClass().
getResource("resources/face.PNG"));

public Component getListCellRendererComponent(
    JList list, Object value, int index,
    boolean isSelected, boolean cellHasFocus) {

    if (isSelected) {
        setBackground(list.getSelectionBackground());
        setForeground(list.getSelectionForeground());
    } else {
        setBackground(list.getBackground());
        setForeground(list.getForeground());
    }

    //Set the icon and text. If icon was null, say so.
    setOpaque(true);
    setIcon(icon);
    setText(
        "<html><b>Co my tu mamy:</b><p>"+value+"</html>");

    if(value instanceof File){
        long lm = ((File)value).lastModified();
        long size=((File)value).length();
        if(((File)value).isDirectory())
            setToolTipText("Modified:" + new Date(lm));
        else
            setToolTipText(
                "Size:"+size+" Modified:" + new Date(lm));
    }else{
        setToolTipText(value.getClass()+" : "+value);
    }
    return this;
}
}
```

```
comboControl.setRenderer(new MyComboBoxRenderer());
```

Komponent JTree

Komponent przechowuje referencje do zewnętrznych obiektów w postaci węzłów reprezentowanych przez obiekty klasy implementującej interfejs `TreeNode` lub `MutableTreeNode`, zazwyczaj wykorzystywana jest biblioteczna klasa `DefaultMutableTreeNode`.

DefaultMutableTreeNode

Klasa `DefaultMutableTreeNode` zapewnia funkcjonalność pozwalającą na ustalanie powiązań pomiędzy węzłem rodzicem i węzłami potomnymi, a także metody iteracji:

`void add(MutableTreeNode newChild)` – dodaje węzeł potomny `newChild`

`void insert(MutableTreeNode newChild, int idx)` – wstawia węzeł potomny na określonej pozycji

`void remove(MutableTreeNode node)` – usuwa węzeł potomny `node`

Różne typy iteracji:

`Enumeration breadthFirstEnumeration()`

`Enumeration depthFirstEnumeration()`

`Enumeration postorderEnumeration()`

`Enumeration preorderEnumeration()`

Warto zwrócić uwagę, że obiekty klasy

`DefaultMutableTreeNode` są częścią modelu, a nie widoku (czy *UI Delegate* skupiającego widok i kontroler), dlatego nie przechowują informacji o stanie wizualnym (węzeł wybrany, węzeł rozwinięty lub zwinięty). Informacje tego typu są przechowywane w dodatkowych strukturach danych komponentu `JTree` (obiekcie klasy `TreeSelectionModel`).

Manipulacja postacią wizualną jest możliwa na dwa sposoby:

- Poprzez ścieżkę `TreePath` prowadzącą od korzenia do danego węzła.
- Poprzez numer wiersza, w którym jest wyświetlany konkretny węzeł (drzewiasta struktura jest wizualizowana w kolejnych wierszach, w których elementy są odpowiednio przesuwane w prawo i rysowane są linie pionowe łączące wiersze).

Inicjalizacja komponentu `JTree`

Najprostszą metodą inicjalizacji komponentu jest utworzenie węzła będącego korzeniem, dodanie do niego węzłów potomnych za pomocą metody `add()` klasy `MutableTreeNode`, a następnie przekazanie korzenia drzewa, jako parametru konstruktora `JTree`.

```
MutableTreeNode root = new MutableTreeNode("Root");
buildTree(root);
JTree tree = new JTree(root);
...
void buildTree(DefaultMutableTreeNode parent)
{
    parent.add(new DefaultMutableTreeNode("Child 1"));
    parent.add(new DefaultMutableTreeNode("Child 2"));
    DefaultMutableTreeNode node = new
        DefaultMutableTreeNode("Child 3");
    parent.add(node);
    node.add(new DefaultMutableTreeNode("Child 3.1"));
}
```

Ten typ inicjalizacji ma duże zalety: dane dla drzewa są przygotowywane przed rozpoczęciem wizualizacji, nie pociągają za sobą konieczności uaktualniania ekranu. Niestety, po skonstruowaniu i wyświetleniu drzewa nie jest możliwa bezpośrednia zmiana jego elementów za pomocą metod `add()` / `insert()` / `remove()` klasy `MutableTreeNode`, ponieważ zmiana struktury drzewa nie pociąga za sobą wysyłania powiadomień do komponentu `JTree`, a tym samym widok nie jest uaktualniany.

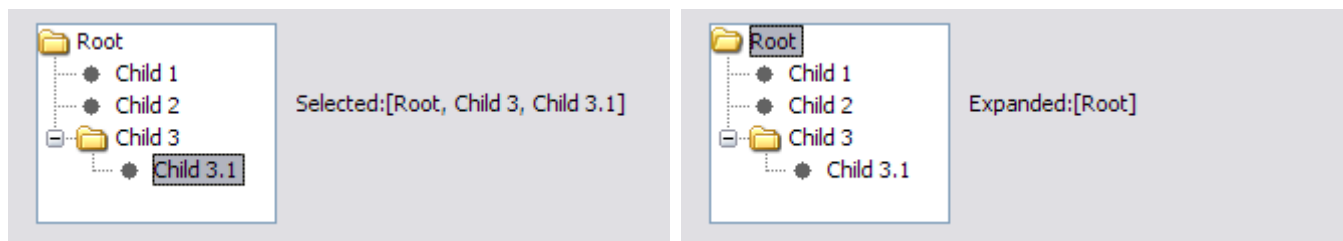
Zdarzenia generowane przez komponent JTree

JTree generuje zdarzenia typu `TreeSelectionEvent` oraz `TreeExpansionEvent`.

Zdarzenia niosą informacje o ścieżkach prowadzących od korzenia drzewa do zmodyfikowanego węzła.

```
tree.addTreeExpansionListener(new TreeExpansionListener() {  
    public void treeCollapsed(TreeExpansionEvent evt) {  
        label.setText( "Collapsed:"+evt.getPath() );  
    }  
    public void treeExpanded(TreeExpansionEvent evt) {  
        label.setText( "Expanded:"+evt.getPath() );  
    }  
});
```

```
tree.addTreeSelectionListener(new TreeSelectionListener() {  
    public void valueChanged(TreeSelectionEvent evt) {  
        label.setText( "Selected:"+evt.getPath() );  
    }  
});
```



- a. Wygląd okna po zaznaczeniu węzła *Child 3.1*
- b. Wygląd okna po kliknięciu na zwinięty węzeł *Root*

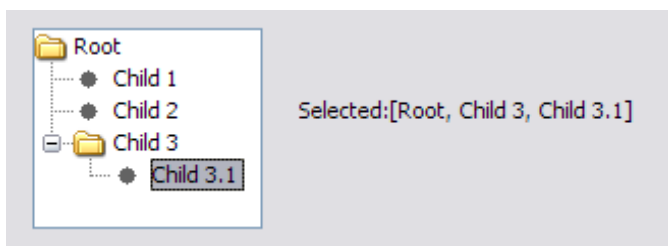
Jak programowo rozwinąć drzewo i zaznaczyć węzeł?

Najwygodniej rozwinąć lub zwinąć wszystkie węzły drzewa posługując się funkcjami `expandRow()` lub `collapseRow()`.

Aby zaznaczyć węzeł zawierający referencję do konkretnego obiektu, należy go odnaleźć, a następnie zaznaczyć ścieżkę prowadzącą do węzła.

```
void expandAllAndSelect() {
    // expand
    for(int i=0;i<tree.getRowCount();i++) tree.expandRow(i);
    // find the node
    Enumeration en = root.breadthFirstEnumeration();
    DefaultMutableTreeNode found=null;
    while(en.hasMoreElements())
    {
        DefaultMutableTreeNode n=
            (DefaultMutableTreeNode)en.nextElement();
        if(n.getUserObject()=="Child 3.1"){
            found=n;
            break;
        }
    }
    // selet the path to the node
    if(found!=null)
        tree.setSelectionPath( new TreePath(found.getPath()));
}
```

W przedstawionym kodzie porównywane są referencje do obiektów (a nie teksty!). Identyczne stałe łańcuchowe są reprezentowane przez pojedynczy obiekt `String`.



Dynamiczna modyfikacja drzewa

Modyfikacja drzewa za pomocą metod węzła drzewa

`DefaultMutableTreeNode` -- `add()`, `insert()`, `remove()` nie pociągają za sobą uaktualnienia widoku. Aby wymusić uaktualnienie należy odwołać się do funkcjonalności klasy realizującej interfejs `TreeModel` (zwykle `DefaultTreeModel`). Klasa jest odpowiedzialna za wysyłanie powiadomień do zarejestrowanych odbiorców za pośrednictwem zdarzeń `TreeModelEvent`.

Interfejs `TreeModelListener` definiuje metody wywoływane w wyniku zmian w modelu:

- `void treeNodeChanged(TreeModelEvent e)`
- `void treeNodeInserted(TreeModelEvent e)`
- `void treeNodeRemoved(TreeModelEvent e)`
- `void treeStructureChanged(TreeModelEvent e)`

Klasa `JTree`, a właściwie jej klasa wewnętrzna

`JTree.TreeModelHandler` jest zarejestrowanym odbiorcą zmian w modelu.

Klasa `DefaultTreeModel` zapewnia metody pozwalające na dokonanie zmian w drzewie oraz wysłanie powiadomień:

- `void insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index)`
- `void removeNodeFromParent(MutableTreeNode node)`
- `void nodeChanged(TreeNode node)`

Ta ostatnia powinna zostać wywołana po zmianie zawartości węzła.

Przykład

Zaprezentowany przykład pokazuje drzewo, które jest dynamicznie budowane po uruchomieniu aplikacji. Korzeniem drzewa jest sztuczny węzeł "My computer". Elementami drzewa są kolejno napędy (pod Windows), katalogi i pliki. Proces budowy drzewa (skanowanie struktury dysków) może trwać kilka minut, dlatego jest realizowany przez wątek, który dynamicznie zmienia zawartość drzewa po utworzeniu obiektu `JTree`.

Podobnie, jak dla rozwijanej listy został zaimplementowany *renderer*, klasa odpowiedzialna za wizualizację elementów drzewa, ustawianie ikon oraz wyświetlanie podpowiedzi z informacjami (*tool-tips*).

```
DefaultMutableTreeNode topTreeNode =  
new DefaultMutableTreeNode("My computer");  
treeControl = new javax.swing.JTree(topTreeNode);  
treeControl.setCellRenderer(new FileRenderer());  
new FillingTreeThread().start();
```

Renderer jest ustawiany po utworzeniu drzewa, następnie uruchamiany jest wątek, dynamicznie dodający jego elementy.

Wątek wypełniający drzewo

```
class FillingTreeThread extends Thread
{
    void listFiles(DefaultMutableTreeNode parent, File f)
    {
        statusMessageLabel.setText("Processing: "+f);
        DefaultTreeModel treeModel=
            (DefaultTreeModel)treeControl.getModel();
        File[] files =f.listFiles();
        for(int i=0;i<files.length;i++){
            DefaultMutableTreeNode child =
                new DefaultMutableTreeNode(files[i]);
            if(files[i].isDirectory()){
                treeModel.insertNodeInto(
                    child, parent, parent.getChildCount());
                listFiles(child,files[i]);
            }else{
                treeModel.insertNodeInto(
                    child, parent, parent.getChildCount());
            }
        }
    }
}
```

```
public void run() {
    DefaultTreeModel treeModel=
        (DefaultTreeModel)treeControl.getModel();
    File[] files = File.listFiles();
    for(int i=0;i<files.length;i++){
        DefaultMutableTreeNode child =
            new DefaultMutableTreeNode(files[i]);
        treeModel.insertNodeInto(child, topTreeNode,
            topTreeNode.getChildCount());
        listFiles(child,files[i]);
    }
    statusMessageLabel.setText("Done...");
}
}
```

Renderer

```
class FileRenderer extends DefaultTreeCellRenderer
```

```
{
```

```
    ImageIcon normalFile = ...  
    ImageIcon hiddenFile = ...;  
    ImageIcon readOnlyFile = ...  
    ImageIcon directoryNormal = ...  
    ImageIcon directoryOpened = ...;
```

```
public Component getTreeCellRendererComponent(  
                JTree tree,  
                Object value,  
                boolean sel,  
                boolean expanded,  
                boolean leaf,  
                int row,  
                boolean hasFocus) {
```

```
    super.getTreeCellRendererComponent(  
        tree, value, sel,  
        expanded, leaf, row,  
        hasFocus);
```

```
    Object o =  
        ((DefaultMutableTreeNode) value).getUserObject();
```

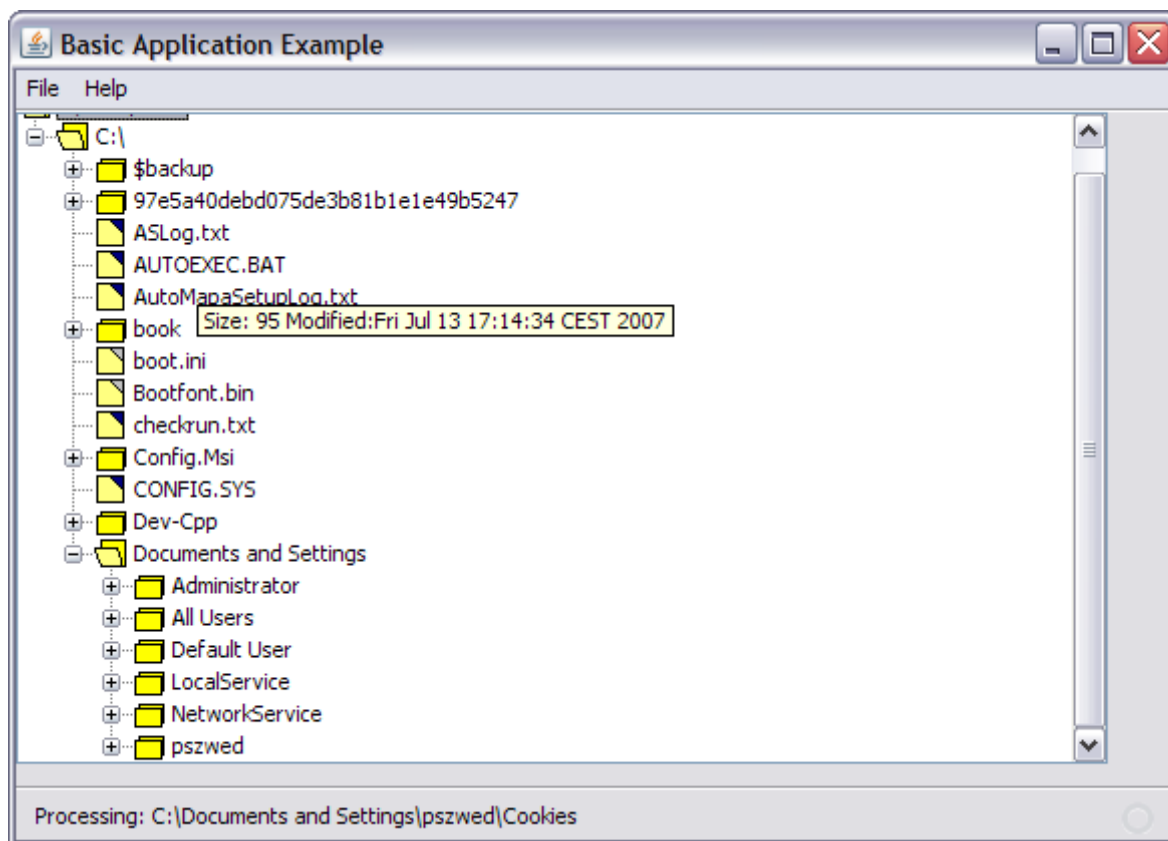
```
    if(o instanceof File){  
        File file =(File)o;  
        if(file.getParent()!=null)  
            setText(file.getName()); // disk C: has no parent  
        long lm = file.lastModified();  
        Date d=new Date(lm);  
        if(file.isDirectory()){  
            setToolTipText("Modified:"+d);  
            if(expanded)setIcon(directoryOpened);  
            else setIcon(directoryNormal);  
            return this;  
        }  
    }
```

```

        setToolTipText (
            "Size: "+file.length()+" Modified:"+d);
        if(file.isHidden()){
            setIcon(hiddenFile);
        }
        else if(!file.canWrite()){
            setIcon(readonlyFile);
        } else {
            setIcon(normalFile);
        }
        return this;
    }
}
// if not file, i.e. root
    if(expanded) setIcon(directoryOpened);
    else setIcon(directoryNormal);

    return this;
}

```



Komponent JTable

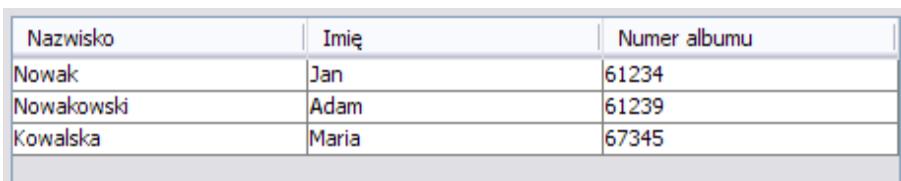
Komponent `JTable` pozwala na wyświetlanie danych w postaci tabeli komórek oraz opcjonalnie na edycję tych danych. Tabela jest

jedynie widokiem danych, które są przechowywane w kontenerze realizującym interfejs `TableModel`. Model tabeli może być wbudowany, można wybrać standardowy model: `DefaultTableModel` lub dostarczyć własny.

Inicjalizacja tabeli

Najprostszy kod dokonujący inicjalizacji tabeli definiuje tablicę tekstów, które będą nazwami kolumn oraz dwuwymiarową tablicę obiektów umieszczonych w tabeli.

```
String []cols={"Nazwisko", "Imię", "Numer albumu"};
String [][]data={
    {"Nowak", "Jan", "61234"},
    {"Nowakowski", "Adam", "61239"},
    {"Kowalska", "Maria", "67345"}};
JTable table = new JTable(data,cols);
```



Nazwisko	Imię	Numer albumu
Nowak	Jan	61234
Nowakowski	Adam	61239
Kowalska	Maria	67345

Niestety, taka inicjalizacja ma wady, ponieważ użytym modelem danych jest wówczas wewnętrzna klasa `JTable`, która nie pozwala na manipulację jej zawartością.

Bardziej elastycznym wyborem jest przypisanie tabeli, jako modelu obiektu klasy `DefaultTableModel`:

```
JTable table = new JTable();
table.setModel(new DefaultTableModel(data, cols));
```

Alternatywnie, można przypisać tabeli pusty model, a dokonać inicjalizacji poprzez manipulację modelem.

Klasa `DefaultTableModel`

Klasa definiuje szereg metod pozwalających na zmianę danych przechowywanych w tabeli. Zmiany w modelu powodują przesyłanie

zdarzeń typu `TableModelEvent` do zarejestrowanych odbiorców (`TableModelListener`) informujących o konieczności uaktualnienia widoku. Jednym z odbiorców jest oczywiście obiekt `JTable`.

Podstawowe metody:

- `void addColumn(Object columnName)`
- `void addRow(Object[] rowData)` **lub**
`void addRow(Vector rowData)`
- `void insertRow(int row, Object[] rowData)` **lub**
`void insertRow(int row, Vector rowData)`
- `void removeRow(int row)`
- `int getRowCount()`
- `int getColumnCount()`

Konfiguracja tabeli

```
JTable table = new JTable();
table.setModel(new DefaultTableModel(data, cols));

...

DefaultTableModel m= (DefaultTableModel)table.getModel();
m.addColumn("Nazwisko");
m.addColumn("Imię");
m.addColumn("Numer albumu");

m.addRow(new Object[]{"Nowak", "Jan", "61234"});
m.addRow(new Object[]{"Nowakowski", "Adam", "61239"});
m.addRow(new Object[]{"", "", ""});
m.setValueAt("Kowalska", 2, 0);
m.setValueAt("Maria", 2, 1);
m.setValueAt("67345", 2, 2);
```

Usuwanie danych z tabeli

```
DefaultTableModel model =
    (DefaultTableModel) table.getModel();
int[] rowIndices = table.getSelectedRows();
for(int i=rowIndices.length-1;i>=0;i--)
    model.removeRow(rowIndices[i]);
```

Iteracja po zawartości tabeli

```
for(int i=0;i<model.getRowCount();i++ ){
    String na = (String) model.getValueAt(i, 0);
    String im = (String) model.getValueAt(i, 1);
    String nr = (String) model.getValueAt(i, 2);
    if(na.isEmpty() && im.isEmpty() &&
        nr.isEmpty()) continue;
    System.out.println(na+";"+im+";"+nr);
}
```

Implementacja własnego modelu tabeli

Jeżeli użytkownik chciałby wyświetlić zawartość pewnego zbioru danych w postaci tabeli (i ewentualnie przeprowadzić jego edycję), może:

- utworzyć kopie indywidualnych obiektów i dodać je do tabeli,
- edytować bezpośrednio dane tabeli
- na zakończenie dokonać iteracji po zawartości tabeli i na tej podstawie uaktualnić zawartość danych w oryginalnym kontenerze.

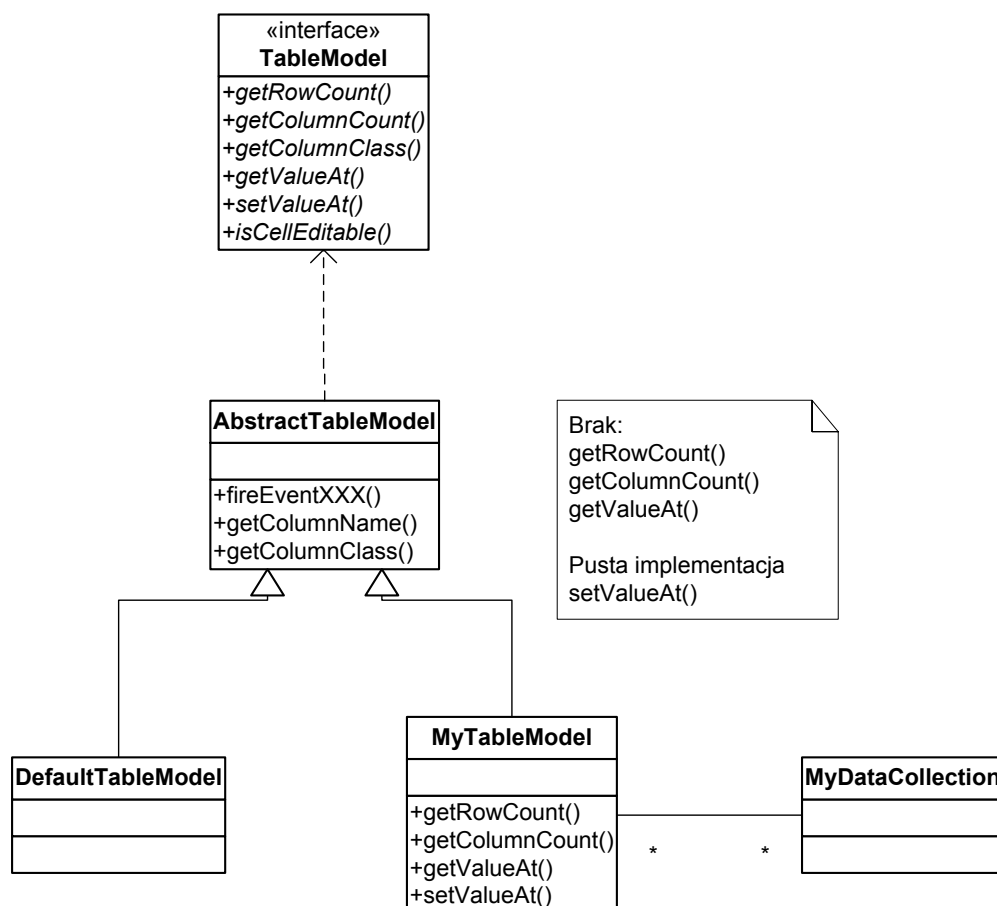
Taka implementacja może być uzasadniona w przypadku małych zbiorów danych (dochodzi dodatkowy etap zatwierdzania zmian, np. w dialogu po naciśnięciu przycisku OK dane będą przepisywane do oryginalnego kontenera). Jednak w przypadku naprawdę dużych zbiorów danych taka implementacja może być nieefektywna lub wręcz nierealizowalna.

Przedstawiony zostanie przykład, w którym dane dla dużej tabeli będą przechowywane w postaci rzadkiej macierzy wypełnionej w około 5%. W tym celu zostanie utworzona klasa `SparseMatrixTableModel` i zastosowana, jako model danych dla tabeli.

Własny model tabeli

- Przygotowanie własnego modelu dla tabeli wymaga implementacji interfejsu `TableModel`. Interfejs ten definiuje podstawowe metody realizujące dostęp do danych (odczyt lub modyfikacja).
- Dodatkowym zadaniem modelu jest wysyłanie powiadomień o zmianach w danych do zarejestrowanych odbiorców

Z tego powodu warto wykorzystać klasę abstrakcyjną `AbstractTableModel`, która implementuje część funkcjonalności związanej z zarządzaniem listą odbiorców `TableModelListener`, wysyła powiadomienia – `fireEventXXX()`, zapewnia standardowe implementacje wybranych metod, zarządza listą kolumn, itp.



Klasa `SparseMatrixTableModel`

Klasa przechowuje dane przyjmujące postać odwzorowania:
(*numer wiersza, numer kolumny*) → *wartość*.

W celu implementacji tego odwzorowania utworzona zostanie zagnieżdżona klasa `class KeyType {int row; int col}`.

Kontenerem przechowującym dane odwzorowania będzie `TreeMap` (drzewo), stąd zostanie on zadeklarowany jako:

```
Map<KeyType, Object> storage =  
    new TreeMap<KeyType, Object> ();
```

Aby móc umieszczać obiekty w drzewie, muszą istnieć mechanizmy *porównywania* ich wartości. Podczas wyznaczania położenia obiektów w drzewie wołana jest metoda `compareTo (Object o)` interfejsu `Comparable`, stąd definicja klasy `KeyType` wygląda następująco;

```
static class KeyType implements Comparable{  
    int row;  
    int col;  
    KeyType(int r,int c){  
        row =r;  
        col=c;  
    }  
  
    public int compareTo (Object o) {  
        KeyType other = (KeyType)o;  
        if(row>other.row) return 1;  
        if(row<other.row) return -1;  
        return col-other.col;  
    }  
}
```

Liczba wierszy i kolumn

Metody `getRowCount()` i `getColumnCount()` zdefiniowane w interfejsie `TableModel` mogą być wołane bardzo często. Należy unikać iteracji po danych, aby wyznaczyć te wartości. Dlatego będą przechowywane w prywatnych atrybutach.

```
private int rowCount=0;
private int columnCount=0;

void setRowCount(int c){rowCount=c;}
void setColumnCount(int c){columnCount=c;}

int getRowCount(){return rowCount;}
int getColumnCount(){return columnCount;}
```

Edycja komórek

Chcąc zezwolić na edycję danych bezpośrednio w komórce tabeli, należy przedefiniować metodę `isCellEditable()`.

```
public boolean isCellEditable(int rowIndex,int
columnIndex){
    return true;
}
```

Zmiana wartości komórki

Aby programowo zmienić zawartość komórki tablicy, należy przeddefiniować pustą metodę `setValueAt()`.

```
public void setValueAt(Object aValue,int rowIndex,int
columnIndex)
{
    if(rowIndex>rowCount-1)rowCount=rowIndex+1;
    if(columnIndex>columnCount-1)
        columnCount=columnIndex+1;
    storage.put(new KeyType(rowIndex,columnIndex), aValue);
    fireTableCellUpdated(rowIndex, columnIndex);
}
```

Wywołanie `fireTableCellUpdated()` powoduje rozesłanie zdarzenia typu `TableModelEvent` do wszystkich zarejestrowanych odbiorców. Bez tego wywołania nie nastąpiłoby uaktualnienie widoku w `JTable`.

Odczyt wartości komórki

Metoda `getValueAt()` ma zwrócić referencję obiektu przypisanego danej komórce. Przy założonej rzadkiej implementacji macierzy:

Poszukujemy klucza w postaci pary (wiersz, kolumna) w drzewie przechowującym odwzorowanie. Jeśli taka para wystąpi, zwracany jest obiekt jej przypisany, w przeciwnym wypadku zwracana jest standardowa wartość `defaultReturnValue`, która jest konfigurowalnym atrybutem klasy.

```
KeyType reusableKey=new KeyType(-1, -1);
Object defaultReturnValue="";

public Object getValueAt(int rowIndex, int columnIndex){
    reusableKey.row=rowIndex;
    reusableKey.col=columnIndex;

    Object o = storage.get(reusableKey);
    if(o==null)return defaultReturnValue;
    return o;
}
```

Przypisanie kolumnom klasy

Implementacja funkcji `getColumnClass()` daje możliwość przekazania do widoku informacji, w jaki sposób należy interpretować wprowadzane dane. Jeżeli wprowadzony tekst nie będzie mógł zostać odpowiednio przetworzony, wówczas wyrzucane i przechwytywane w widoku wyjątki uniemożliwią wyjście z edycji komórki z niepoprawnymi danymi.

```
Class defaultClassForColumn=String.class;

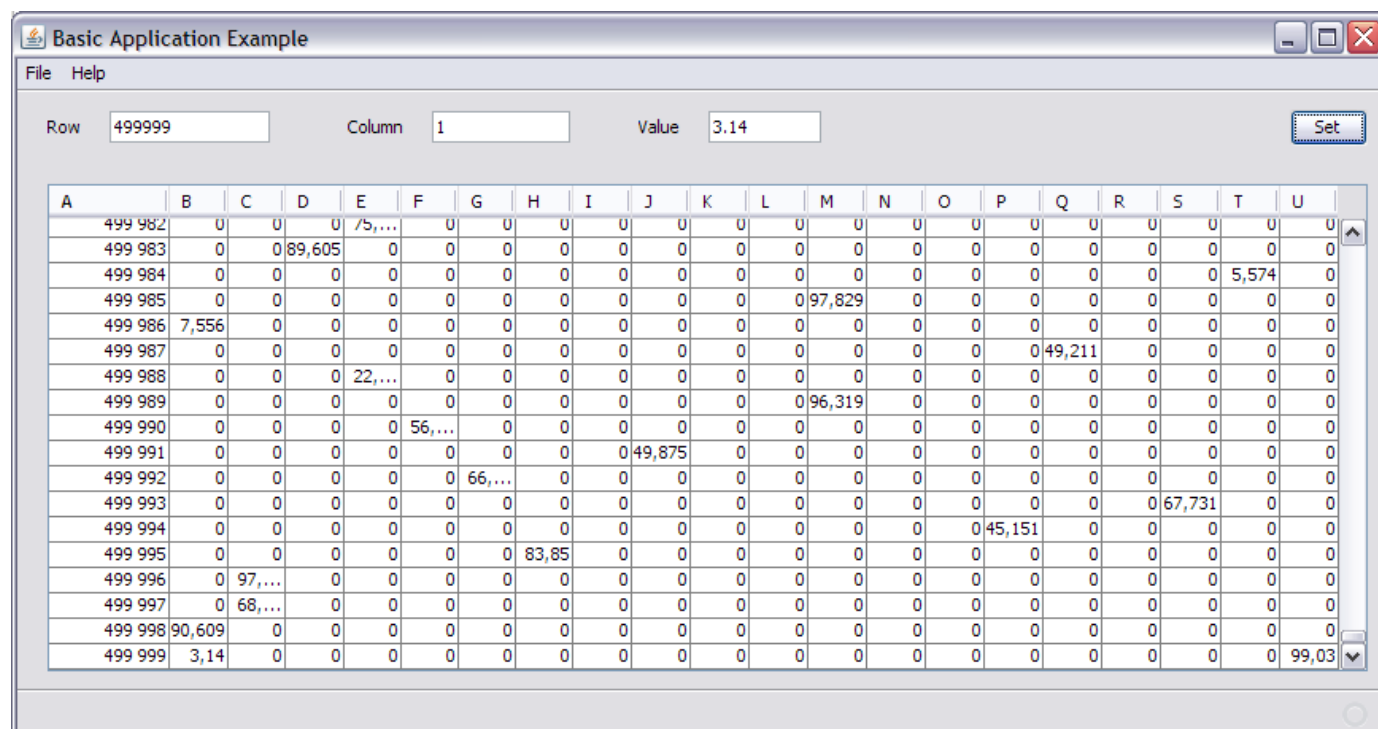
public Class getColumnClass(int columnIndex)
{
    return defaultClassForColumn;
}
```


Kod inicjujący tabelę

```
table = JTable();
table.setModel(new SparseMatrixTableModel());

SparseMatrixTableModel m =
(SparseMatrixTableModel) table.getModel();
m.defaultReturnValue=new Double(0.0);
m.defaultClassForColumn=Double.class;

for(int i=0;i<500000;i++){
    m.setValueAt(new Double(i),i,0);
    int col = (int)(20*Math.random()+1);
    m.setValueAt(new Double(100*Math.random()),i,col);
}
// update afterwards
m.fireTableStructureChanged();
```



Basic Application Example

File Help

Row: 499999 Column: 1 Value: 3.14 [Set]

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
499 982	0	0	0	75,...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
499 983	0	0	89,605	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
499 984	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5,574	0
499 985	0	0	0	0	0	0	0	0	0	0	0	97,829	0	0	0	0	0	0	0	0
499 986	7,556	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
499 987	0	0	0	0	0	0	0	0	0	0	0	0	0	0	49,211	0	0	0	0	0
499 988	0	0	0	22,...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
499 989	0	0	0	0	0	0	0	0	0	0	0	96,319	0	0	0	0	0	0	0	0
499 990	0	0	0	0	56,...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
499 991	0	0	0	0	0	0	0	0	49,875	0	0	0	0	0	0	0	0	0	0	0
499 992	0	0	0	0	0	66,...	0	0	0	0	0	0	0	0	0	0	0	0	0	0
499 993	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	67,731	0	0
499 994	0	0	0	0	0	0	0	0	0	0	0	0	0	0	45,151	0	0	0	0	0
499 995	0	0	0	0	0	0	83,85	0	0	0	0	0	0	0	0	0	0	0	0	0
499 996	0	97,...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
499 997	0	68,...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
499 998	90,609	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
499 999	3,14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	99,03

Komponent JList

Komponent pozwala na wyświetlenie grupy elementów rozmieszczonych w wielu wierszach i jednej lub kilku kolumnach.

Układ wierszy i kolumn przypomina `BoxLayout` – komórki rozmieszczone w wierszach i kolumnach mają stałą wysokość i szerokość ustalaną na podstawie najdłuższego tekstu przypisanego elementowi listy.

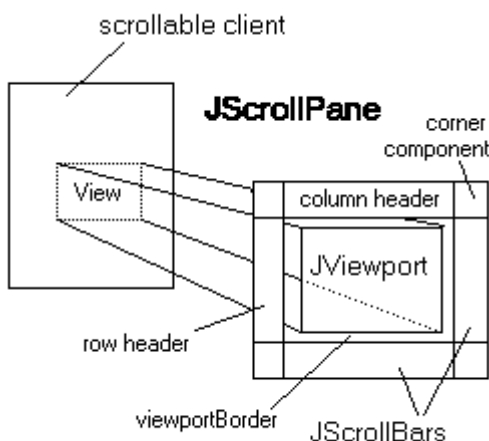
Podobnie jak dla pozostałych komponentów, umieszczamy w nich obiekty. Standardowo, są one reprezentowane przez napisy zwrócone przez metodę `toString()`.

Inicjalizacja listy

Najprostszą formą inicjalizacji listy jest dostarczenie jej danych w konstruktorze.

```
Object[] data = {
    "Anna", "Maria", "Jan", "Tomasz",
    "Zenobia", "Barbara", "Teofil"};
JList list = new JList(data);
```

Zazwyczaj lista zawiera zbyt wiele elementów, aby mogła zmieścić się w przeznaczonym obszarze. Dlatego często podczas konstrukcji listy tworzy się dodatkowy komponent `JScrollPane`, który będzie odpowiedzialny za wizualizację fragmentu listy oraz obsługę suwaków.



[Źródło: <http://java.sun.com/j2se/1.4.2/docs/api/javawx/swing/JScrollPane.html>]

```
JList list = new JList(data);
```

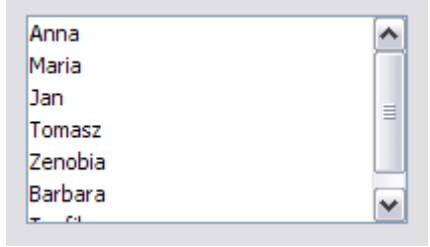
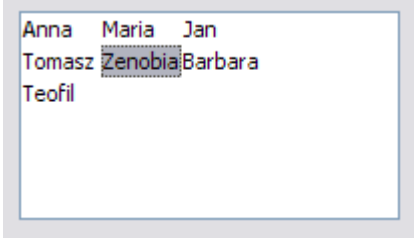
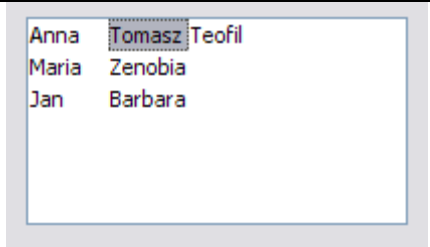
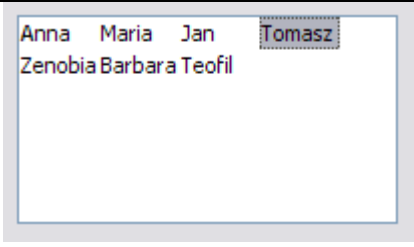
```
JScrollPane listScrollPane;  
listScrollPane = new JScrollPane(list);  
add(listScrollPane);
```

lub

```
JScrollPane listScrollPane;  
listScrollPane = new JScrollPane();  
listScrollPane.setViewportView(list);  
add(listScrollPane);
```

Komponent `JList` zapewnia funkcje do konfiguracji sposobu wyświetlania elementów.

```
list.setVisibleRowCount(3);  
list.setLayoutOrientation(  
    JList.HORIZONTAL_WRAP);
```

	
Tryb standardowy	Liczba wierszy: 3 HORIZONTAL_WRAP
	
Liczba wierszy: 3 Orientacja: VERTICAL_WRAP	Liczba wierszy: -1 HORIZONTAL_WRAP

Wybór elementów listy

Lista przechowuje informacje o stanie zaznaczenia elementów w specjalnym modelu `ListSelectionMode`. Użytkownik ma możliwość ustalenia trybu selekcji za pomocą metody:

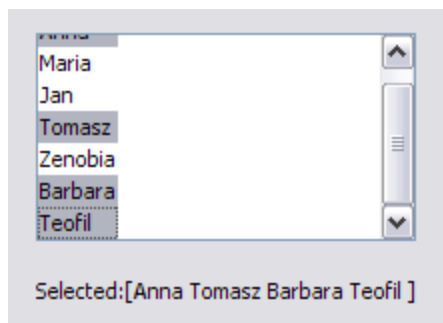
`setSelectionMode()` przekazując jedną ze stałych:

- `SINGLE_SELECTION`,
- `SINGLE_INTERVAL_SELECTION`,
- `MULTIPLE_INTERVAL_SELECTION`.

Lista generuje zdarzenia z informacją o zmianie stanu elementów typu `ListSelectionEvent`. Interfejs odbiorcy `ListSelectionListener` definiuje tylko jedną metodę `void valueChanged()`. Zdarzenie nie przynosi szczególnie cennych informacji, ale można w reakcji na nie odczytać stan listy.

Przykład

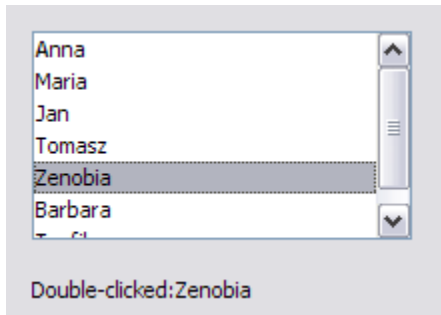
```
list.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent evt) {
        StringBuffer msg=new StringBuffer("Selected: [");
        for(Object o:list.getSelectedValues()){
            msg.append(o.toString());
            msg.append(' ');
        }
        msg.append( ']');
        label.setText(msg.toString());
    }
});
```



Reakcja na podwójne kliknięcie

Komponent `JList` nie zapewnia wbudowanej obsługi zdarzeń typu: podwójne kliknięcie na jeden z jej elementów. Możliwe jest jednak przechwycenie zdarzeń generowanych przez mysz.

```
list.addMouseListener(
    new MouseAdapter() {
        public void mouseClicked(MouseEvent evt) {
            if(evt.getClickCount() == 2) {
                int index = list.locationToIndex(evt.getPoint());
                label.setText("Double-clicked:"
                    +list.getModel().getElementAt(index));
            }
        }
    });
```



Dynamiczna zmiana zawartości listy

Chcąc zezwolić na modyfikację listy w trakcie działania programu, najwygodniej zainicjować ją wybierając, jako model `DefaultListModel`. Korzystając z metod tej klasy:

- `void add(int index, Object element)`
- `void addElement(Object obj)`
- `Object set(int index, Object element)`
- `Object remove(int index)`
- `void removeAllElements()`
- `boolean removeElement(Object obj)`

można zmieniać jej zawartość (i uaktualniać widok).

Inicjalizacja listy:

```
JList list = new JList(new DefaultListModel());
```

lub

```
JList list = new JList();
```

```
list.setModel(new DefaultListModel());
```

...

```
DefaultListModel m = (DefaultListModel)list.getModel();
```

```
for(File f:new File("some directory").listFiles())
```

```
    m.addElement(f);
```

Przykład

Przedstawiony przykład będzie wyświetlał listę, której elementami będą pliki. Sposób wyświetlania plików będzie zmodyfikowany przez użycie odpowiedniego obiektu *renderer* (wzorzec Flyweight).

Renderer zostanie skonfigurowany w ten sposób, aby w przypadku, gdy plikiem jest obraz w formacie JPG (dla innych typów plików graficznych zachowanie można analogicznie oprogramować) wyświetlać jego miniaturę (*thumbnail*). W tym celu w locie dokona generacji miniatury z obrazu oraz zapisze ją w pamięci w postaci odwzorowania *File*→*ImageIcon* realizowanego przez atrybut okna zdefiniowany jako:

```
Map<File, ImageIcon> map = new HashMap<File, ImageIcon>();
```

Przepis na działanie obiektu *Renderer* wydaje się prosty:

1. Jeśli obrazek dla danego obiektu *File* *f* został utworzony, czyli wywołanie `icon = map.get(f)` zwróci wartość różną od `null`, wówczas ustaw: `setIcon(icon)`
2. W przeciwnym przypadku:
 - a. Wygeneruj obrazek `icon` dla pliku `f`
 - b. Dodaj obrazek: `map.put(f, icon)`
 - c. Powrót do punktu (1)

Niestety takie rozwiązanie jest zbyt naiwne, ponieważ czas ładowania z dysku obrazu (zwłaszcza fotografii) jest wielokrotnie większy od czasu przeznaczonego na wyświetlanie. Dlatego obrazy w Java API zawsze ładowane są *asynchronicznie*.

- Typowe wywołanie funkcji pozwalającej na załadowanie obrazu z dysku:

```
Image img = list.getToolkit().getImage(f.getPath());
```

tworzy **pusty** obiekt typu *Image* i równocześnie tworzy i uruchamia działający w tle wątek *Image Fetcher*, który stopniowo ładuje obraz.

- Wątek *Image Fetcher* wysyła powiadomienia do obiektu realizującego interfejs *ImageObserver* za pomocą wywołań metody interfejsu:

```
boolean imageUpdate().
```

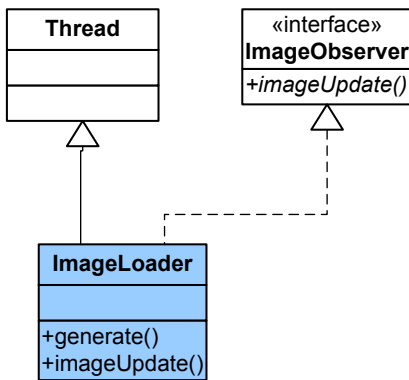
Metoda powinna zwrócić wartość `true`, jeżeli należy kontynuować proces ładowania lub `false`, gdy wątek ma zakończyć działanie (ponieważ obraz został załadowany lub nie jest już potrzebny).

- Sytuacja staje się jeszcze bardziej skomplikowana: *ImageObserver* nie jest parametrem funkcji, która ładuje obraz, ale pojawia się w wywołaniach `Image.getWidth()`, `Image.getHeight()` oraz `Graphics.drawImage()`. Pojawienie się wywołania tych funkcji może obudzić nieaktywny wątek *Image Fetcher*.
- Komponent AWT (*Component*) realizuje interfejs *ImageObserver*. Załadowanie obrazu standardowo przerywa działanie wątku *Image Fetcher* oraz powoduje przerysowanie komponentu.
- Warto zdać sobie sprawę z rozmiarów danych. Załadowanie 100 fotografii do pamięci jest w praktyce niemożliwe (jeden obraz może zająć kilkadziesiąt MB – podczas ładowania są one dekompresowane). Równocześnie 100 ikon o rozmiarach 64x64 to jedynie 400 kB.

Architektura aplikacji

Zadanie generacji ikon będzie realizowane przez wątek

`ImageLoader`. Klasa realizuje interfejs `ImageObserver` – czyli jest odbiorcą komunikatów o stanie ładowania obrazka przesyłanych za pomocą `imageUpdate()`.



Wątek `ImageLoader` będzie przetwarzał kolejne pliki, dlatego jeżeli metoda `imageUpdate()` zostanie wywołana z flagą informującą o tym, że obraz jest częściowo załadowany, będzie usypiany na ok. 500 ms. Jeżeli metoda `imageUpdate()` przekaże flagę `ALLBITS` (obraz załadowany), wówczas wątek wygeneruje ikonę.

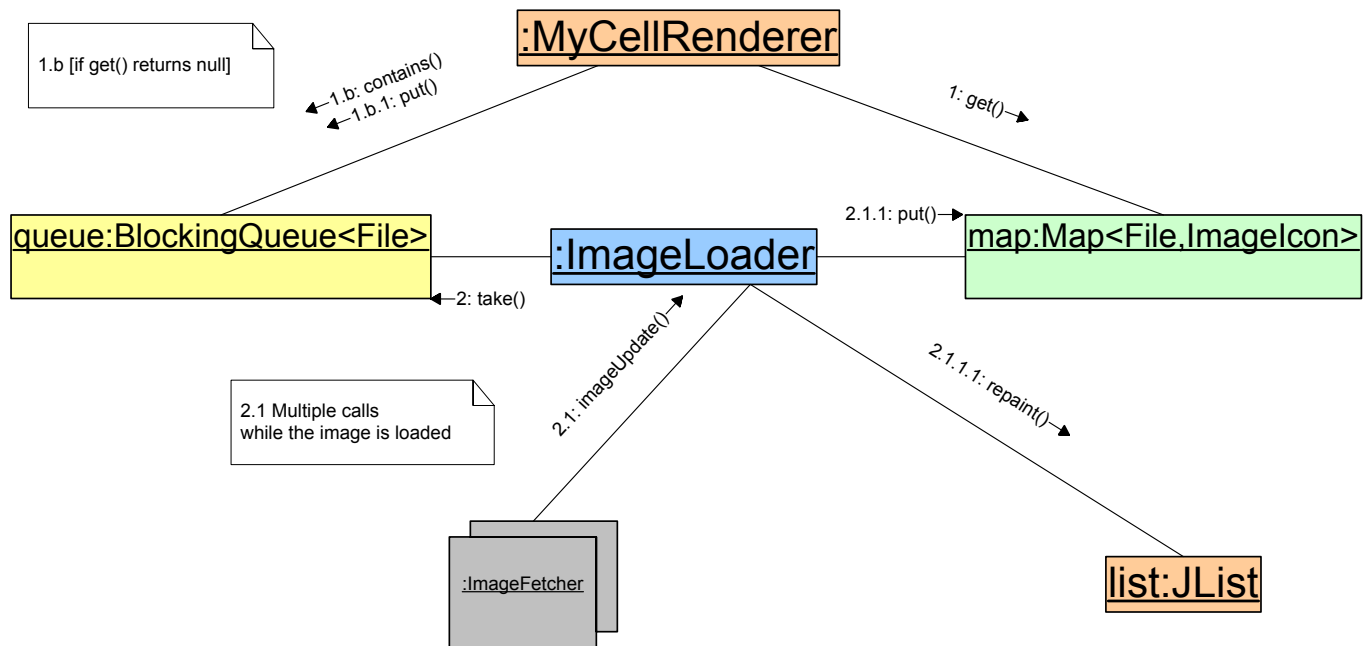
Wątek pełni w architekturze rolę *konsumenta* odczytującego zlecenia z plikami do przekonwertowania z *bufora*.

Rolę *bufora* będzie pełniła kolejka zdefiniowana jako:

```
BlockingQueue<File> queue = new LinkedBlockingQueue<File>();
```

Metoda `take()` kolejki blokującej zawiesza wykonujący ją wątek, jeżeli kolejka jest pusta.

Rolę *producenta* umieszczającego pliki w kolejce będzie pełnił `Renderer`. Teoretycznie wywołanie metody `put()` umieszczającej dane w kolejce może zawiesić wywołujący ją wątek, ale kolejka typu `LinkedBlockingQueue` jest praktycznie nieograniczona. Jej pojemnością jest `Integer.MAX_VALUE`.



Na diagramie komunikacji pokazano scenariusze działania producenta *MyCellRenderer* (wykonywanego w wątku przetwarzania zdarzeń Swing) i konsumenta *ImageLoader*.

MyCellRenderrer ma skonfigurować komponent dla pliku `File f`

1. Wykonuje `icon=map.get(f)`
 - 1.a Jeżeli `icon!=null` ustawia `setIcon(icon)`.
 - 1.b. W przeciwnym przypadku sprawdza, czy kolejka nie zawiera elementu `f`
 - 1.b.1 Dodaje `f` do kolejki: `queue.put(f)`

Jeżeli zwrócona w (1) referencja `icon` ma wartość `null`, używana jest zastępcza ikona `defaultIcon`.

Wątek ImageLader

2. Odczytuje plik z kolejki: `File f=queue.take()` i inicjuje ładowanie obrazka

2.1 Kiedy otrzyma informację, że plik został załadowany (flaga `ALLBITS` przekazana w wywołaniu `imageUpdate()`) wygeneruje ikonę `icon`.

2.1.1 Doda parę `(f, icon)` do obiektu `map` – `map.put(f, icon)`

2.1.1.1 Wywoła przerysowanie komponentu `list`.

Kod klasy okna

W klasie okna zdefiniowane są odpowiednie atrybuty i klasy wewnętrzne:

```
class JListTestView extends FrameView{
    BlockingQueue<File> queue =
        new LinkedBlockingQueue<File>();
    Map<File, ImageIcon> map = new HashMap<File, ImageIcon>();
    ImageIcon defaultIcon = new ImageIcon(
        this.getClass().
        getResource("resources/placeholder.PNG"));
    JScrollPane listScrollPane = JScrollPane();
    JList list = new JList();

    public JListTestView(){...}
    ...
    class ImageLoader extends Thread implements ImageObserver{...}
    class MyCellRenderer extends DefaultListCellRenderer {...}
}
```

Konstruktor okna

W konstruktorze okna tworzona jest lista i konfigurowana tak, aby wyświetlać elementy w kilku kolumnach. Ustawiany jest *renderer* i uruchamiany wątek *ImageLoader*. Do listy dodawane są pliki.

```
public JListTestView() {
    list.setModel(new DefaultListModel() );
    listScrollPane.setViewportView(list);

    list.setSelectionMode(
        ListSelectionMode.SINGLE_INTERVAL_SELECTION);
    list.setVisibleRowCount(-1);
    list.setLayoutOrientation(JList.HORIZONTAL_WRAP);
    list.setCellRenderer(new MyCellRenderer() );
    new ImageLoader() .start();

    DefaultListModel m = (DefaultListModel)list.getModel();
    for(File f:new File("directory").listFiles())
        m.addElement(f);

    ...
}
```

Kod klasy wewnętrznej okna MyCellRender

```
class MyCellRenderer extends DefaultListCellRenderer {

    Dimension preferredSize=new Dimension(130,130);

    public Component getListCellRendererComponent(
        JList list,
        Object value,    // value to display
        int index,      // cell index
        boolean iss,    // is the cell selected
        boolean chf)    // cell have the focus?
    {
// ustawienie standardowych atrybutów
        super.getListCellRendererComponent(
            list, value, index, iss, chf);

// konfiguracja sposobu wyświetlania
        setPreferredSize(preferredSize);
        setHorizontalAlignment(CENTER);
        setVerticalAlignment(CENTER);
        setVerticalTextPosition(JLabel.BOTTOM);
        setHorizontalTextPosition(JLabel.CENTER);

        if(value instanceof File){
            File f = (File)value;
            setText(f.getName());
            ImageIcon icon = defaultIcon;
            setToolTipText(f.getName());

            if(f.getName().toLowerCase().endsWith(".jpg") ) {
                icon = map.get(f);
                if(icon==null){
                    icon = defaultIcon;
                    if(!queue.contains(f))
                        try {queue.put(f);}
                        catch (InterruptedException ex) { }
                }
            }
            setIcon(icon);
        }
        return this;
    }
}
```

Kod wątku ImageLoader (klasa wewnętrzna okna)

```
class ImageLoader extends Thread implements ImageObserver{  
  
    boolean imageLoaded=false;  
  
    public void run(){  
        for(;;){  
            File f=null;  
            try {  
                f = queue.take();  
                ImageIcon icon= generate(f);  
                map.put(f, icon);  
                list.repaint();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            } catch (Exception e){  
                // OutOfMemory?? invalid format ??  
                map.put(f, defaultIcon);  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```
ImageIcon generate(File f) throws InterruptedException{  
    imageLoaded=false;  
    Image img = list.getToolkit().getImage(f.getPath());  
  
    img.getWidth(this); // obudz ImageFetcher  
    while(!imageLoaded) sleep(500);  
  
    // generacja ikony  
    BufferedImage bi =  
        new BufferedImage(64,64,BufferedImage.TYPE_INT_ARGB);  
    Graphics g=bi.getGraphics();  
    g.drawImage(img, 0, 0, 64,64,this);  
    g.dispose(); // zwolnij zasoby!  
  
    ImageIcon icon = new ImageIcon(bi);  
    return icon;  
}
```

```
// metoda imageUpdate modyfikuje flagę imageLoaded
// i odblokowuje wątek uśpiony w metodzie generate

public boolean imageUpdate(Image img, int infoflags,
    int x, int y, int width, int height)
{
    if (infoflags != ALLBITS) { // not finished
        return true;
    } else {
        imageLoaded = true;
        return false;
    }
}
}
```

Okno aplikacji

Rysunek przedstawia okno aplikacji w trakcie generacji ikon (thumbnail).

