Java Beans

1 Wprowadzenie

Ziarnem Javy (Java Beans) jest fragmentem kodu, który „raz napisany może być używany wielokrotnie”. Zalety ziaren określa się jako:

- *Write Once, Run Anywhere* - pakiety java.beans są częścia core API.
- *Wieloużywalność* komponentów - używane wszędzie (platformy/narzędzia/wdrożenia, np. 3D Charting Bean – wstawiany do kontenera, podobnie jak standardowe komponenty)
- *Interoperacyjność* - komunikacja z innymi architekturami komponentów (Beans-ActiveX Bridge, Beans-OpenDoc)

Aby jakiś twór można było nazwać ziarnem, musi on spełniać kilka warunków:

- Musi istnieć możliwość utworzenia egzemplarza tego tworu, tzn. ziarna tworzy się z klas, które nie są interfejsami ani klasami abstrakcyjnymi.
- Klasa, z której powstaje ziarno musi posiadać bezparametrowy konstruktor.
- Klasa, z której powstaje ziarno musi zawierać implementacje interfejsów Serializeable lub Externalizeable (interfejsy te umożliwiają zapisanie obiektu do strumienia w postaci szeregu bajtów).
- Schemat klasy ziarna powinien spełniać wymagania wzorca projektowego (tzn. musi spełniać reguły związane z tworzeniem i nazywaniem metod)
- W klasie powinny istnieć metody typu get i set, umożliwiające użytkownikom manipulowanie właściwościami ziarna (właściwościami prostymi). W ogólności ziarno może posiadać:
    - właściwości proste (Simple Properties)
    - właściwości wiązane (Bound Properties)
    - właściwości ograniczane (Constrained Properties)
    - właściwości indeksowe (Indexed Properties)

Do ziaren bywa dodawana klasa opisująca BeanInfo oraz klasa Customizer, umożliwiająca edycję właściwości ziarna. Ziarna, skompilowane lub w postaci źródłowej, rozprowadzane są w plikach JAR.

Trwałość jest zdolnością obiektów do zachowania jego stanu w celu późniejszego odtworzenia. Ziarna Javy są trwałe. Ich trwałość zapewnia mechanizm serializacji. Jeśli jakiś obiekt posiada referencje do innych obiektów, to podczas serializacji zostają one zachowane wraz z nim. Przykładem może być serializacja obiektu klasy TreeNode.

```
TreeNode top = new TreeNode("top");
top.addChild(new TreeNode("left child"));
top.addChild(new TreeNode("right child"));

// Zapisanie obiektu:
FileOutputStream fOut =  new FileOutputStream("test.out");
ObjectOutput out = new ObjectOutputStream(fOut);
out.writeObject(top);
out.flush();
out.close();

// Odtworzenie obiektu:
FileInputStream fIn = new FileInputStream("test.out");
ObjectInputStream in = new ObjectInputStream(fIn);
TreeNode n = (TreeNode)in.readObject();
```

Porównanie technologii JavaBeans z ActiveX/COM:

- JavaBeans jest szkieletem do budowy aplikacji z komponentów Javy (Beans)
- ActiveX jest szkieletem do budowy złożonych dokumentów z kontrolkami ActiveX

- Beans są bardzo podobne do kontrolek ActiveX. Dodatkowo Beans są napisane w Javie, dzięki czemu posiadają jej bezpieczność i systemową niezależność
- Bardzo często kontrolki ActiveX są pisane w Visual Basic lub Visual C++ i następnie kompilowane z natywnymi bibliotekami

Więcej informacji można znaleźć pod adresami:
http://www.javaworld.com/javaworld/jw-02-1997/jw-02-activex-beans.html
http://www.javaworld.com/javaworld/jw-03-1997/jw-03-avb-tech.html

2 Architektura Java Beans

Każde ziarno Javy składa się z trzech elementów:

- właściwości
- metod
- zdarzeń

Aby stworzyć ziarno najpierw należy określić jego przeznaczenie, zdefiniować zdarzenia, oraz określić właściwości razem z metodami dostępu do właściwości.

2.1 Właściwości

Cechy charakterystyczne konkretnego ziarna opisane są przez jego właściwości. Właściwości są publicznymi atrybutami ziarna, zwykle reprezentowanymi przez nie-publiczne parametry instancji. Ziarno można zmieniać, manipulując właśnie jego właściwościami. Właściwości mogą być do zapisu i odczytu, tylko do zapisu lub tylko do odczytu.

2.1.1 Właściwości proste

W najprostszym przypadku manipulacja właściwościami dokonywana jest za pomocą metod zdefiniowanych według wzorów:

```
public void set<i>PropertyName</i>(PropertyType value);
public <i>PropertyType</i> get<i>PropertyName</i>();
```

gdzie:

PropertyName jest nazwą właściwości,
PropertyType jest typem właściwości.

przy czym nazwa właściwości wcale nie musi pokrywać się z nazwą parametru klasy odpowiadającego właściwości.

Aby uzyskać właściwość tylko do odczytu należy zadeklarować jedynie metodę get. Stworzenie tylko metody set powoduje powstanie właściwości tylko do zapisu.
W przypadku właściwości typu Boolean można stosować zamiast metody get metodę o nazwie is*PropertyName*.

Przykład:

Niech ziarnem będzie obiekt graficzny, który może mieć określony kolor (domyślnie niech to będzie Color.red). Wtedy w klasie ziarna powinna pojawić się deklaracja parametru:

```
private Color kolor = Color.green;
```

Ponieważ kolor jest właściwością ziarna, należałoby jeszcze dostarczyć metody do manipulacji tą właściwością:

```
public Color getKolor() {
        return kolor;
}
public void setColor(Color nowyKolor) {
kolor = nowyKolor;
repaint();
}
```

Przykład:

Niech będzie ziarno Employee z właściwościami salary (typu float) oraz trained (typu boolean). Wtedy w klasie Employee mamy:

```
float salary;
public void setSalary (float newSalary) {
  salary = newSalary;
}
public float getSalary () {
  return salary;
}

boolean trained;
public void setTrained (boolean trained) {
  this.trained = trained;
}
public boolean isTrained () {
  return trained;
}
```

W Javie ziarna mogą być komponentami widocznymi lub nie. Jeśli jest to pierwszy przypadek, to ziarno powinno mieć metodę pozwalającą na jego rysowanie, np.:

```
public void paint(Graphics g) {
g.setColor(kolor);
....
}
```

2.1.2 Właściwości wiązane

Właściwości wiązane ziarna są to takie cechy, których zmiana pociąga za sobą reakcje w innych ziarnach. Mechanizm, który o tych zmianach informuje oparty jest na zdarzeniach. Jest on realizowany za pomocą klas PropertyChangeSupport, PropertyChangeEvent, PropertyChangeListener (w kodzie źródłowym pojawia się import java.beans.*;).

Klasa ziarna posiadającego właściwości wiązane powinna implementować interfejs PropertyChangeSupport, tzn. powinna dostarczać metod na dodawanie i usuwanie słuchaczy zdarzeń. Aktualna lista słuchaczy zdarzeń zawiera referencje do obiektów, którym sygnalizowany będzie fakt modyfikacji właściwości wiązanej. Sygnalizacja ta odbywa się przez zgłoszenie zdarzenia PropertyChangeEvent, generowanego po zmianie właściwości wiązanej, każdemu ze słuchaczy. Odbiór odbywa się w metodzie propertyChange() (tzn. że słuchacze powinni implementować interfejs PropertyChangeListener).

Przykład:

Ziarno Eployee, w którym właściwość salary jest własnością wiązaną

```
private PropertyChangeSupport changes = new PropertyChangeSupport
(this);
public void addPropertyChangeListener (
    PropertyChangeListener p) {
  changes.addPropertyChangeListener (p);
}
public void removePropertyChangeListener (
    PropertyChangeListener p) {
  changes.removePropertyChangeListener (p);
}
public void setSalary (float salary) {
```

```
  Float oldSalary = new Float (this.salary);
  this.salary = salary;
  changes.firePropertyChange (
    "salary", oldSalary, new Float (this.salary));
}
```

Po stronie odbierającej (innego ziarna) należy zdefiniować metodę propertyChange:

```
public void propertyChange(PropertyChangeEvent e);
```

Metoda firePropertyChange wymaga podania starej i nowej wartości modyfikowanej właściwości. Metoda ta przekazuje swoje parametry do obiektu typu PropertyChangeEvent i wywołuje propertyChange(PropertyChangeEvent pce) na każdym zarejestrowanym słuchaczu. Przekazywanymi parametrami mogą być tylko obiekty. Dlatego parametry o typach prostych muszą być przekonwertowane do obiektów odpowiednich klas (np. parametr parametr typu int do obiektu klasy java.lang.Integer).

Składnia metody firePropertyChange jest następująca:

```
public void firePropertyChange(String propertyName, Object oldValue, Object
newValue)
```

2.1.2.1 Implementacja słuchaczy wartości wiązanych.

Aby ziarno mogło nasłuchiwać zmian, musi ono implementować interfejs PropertyChangeListener. W interfejsie tym istnieje metoda public abstract void propertyChange(PropertyChangeEvent evt). Metoda ta jest wywoływana na każdym zarejestrowanym słuchaczu przez ziarno, którego własność wiązana została zmodyfikowana. Jej implementacja powinna być reakcją na dokonane zmiany. Stąd deklaracja klasy ziarna-słuchacza powinna wyglądać następująco:

```
public class MyClass implements java.beans.PropertyChangeListener,
java.io.Serializable
```

Rejestracja słuchaczy (tj. obiektów posiadających metodę propertyChange) w obiekcie mającym właściwość wiązaną odbywa się przez wywołanie jego metody addPropertyChangeListener().

2.1.3 Właściwości ograniczane

Właściwości ograniczane podobne są do właściwości wiązanych. Tym razem jednak modyfikacje własności mogą być zawetowane. Aby to umożliwić, oprócz słuchaczy PropertyChangeListeners, ziarno dodatkowo przechowuje listę słuchaczy VetoableChangeListeners.

Modyfikacja wartości ograniczonej odbywa się w trzech krokach:

1. Słuchacze VetoableChangeListeners pytani są o pozwolenie na dokonanie zmiany własności ograniczonej (wykorzystywana jest wtedy metoda fireVetoableChange klasy VetoableChangeSupport)
2. W przypadku odmowy wyrzucany jest wyjątek PropertyVetoException (który również powinien być zadeklarowany) i nie ma modyfikacji własności. Przy braku wyjątku dokonywana jest modyfikacja własności.
3. O dokonanej modyfikacji informowani są słuchacze PropertyChangeListeners.

Obiektami mogącymi zgłosić *veto* mogą być: obiekt słuchający, jak i samo ziarno. Klasami wykorzystywanymi w tym wypadku są: VetoableChangeListener, PropertyChangeEvent, VetoableChangeSupport.

Przykład:

Implementacja własności salary jako własności ograniczonej – w czasie veta wyrzucany będzie wyjątek.

```
private VetoableChangeSupport vetoes =
   new VetoableChangeSupport (this);
```

```
public void addVetoableChangeListener (VetoableChangeListener v) {
  vetoes.addVetoableChangeListener (v);
}
public void removeVetoableChangeListener (VetoableChangeListener v) {
  vetoes.removeVetoableChangeListener (v);
}

public void setSalary (float salary) throws PropertyVetoException {
  Float oldSalary = new Float (this.salary);
  vetoes.fireVetoableChange ( "salary", oldSalary, new Float (salary));
  this.salary = salary;
  changes.firePropertyChange ( "salary", oldSalary, new Float (this.salary));
}
```

Po stronie odbiorczej należy zdefiniować metodę vetoableChange.

```
public void vetoableChange(PropertyChangeEvent e)
  throws PropertyVetoException;
```

Zaleca się, aby własności ograniczane były jednocześnie właściwościami powiązanymi. Zamiast dostarczać osobnych słuchaczy reagujących na zmiany jakichś właściwości (PropertyChangeListener) i osobnych słuchaczy zgłaszających veto do zmian (VetoableChangeListener), można utworzyć osobne listy słuchaczy dla każdej z właściwości. Wzorzec projektowy takiego rozwiązania wygląda następująco:

```
public void add<PropertyName>Listener ( PropertyChangeListener p);
public void remove<PropertyName>Listener ( PropertyChangeListener p);

public void add<PropertyName>Listener ( VetoableChangeListener v);
public void remove<PropertyName>Listener ( VetoableChangeListener v);
```

Klasa JComponent posiada pewne możliwości związane z zarządzaniem obiektami nasłuchującymi właściwości ograniczonym, ale nie tak rozbudowaną jak w przypadku właściwości powiązanych. Klasa JComponent utrzymuje wspólną listę dla wszystkich obiektów nasłuchujących właściwości ograniczonych, a nie osobną dla każdej właściwości.

2.1.4 Właściwości indeksowane

Właściwości indeksowane to kolekcje właściwości prostych, do których dostęp odbywa się poprzez indeks (jak w tablicach). Wzorzec projektowy takich właściwości wygląda następująco:

metody dostępu do całej tablicy właściwości:	metody dostępu do pojedynczych właściwości
public *PropertyType*[] get*PropertyName* () public void set*PropertyName* (*PropertyType*[] list)	public *PropertyType* get*PropertyName* (int position) public void set*PropertyName* (*PropertyType* element, int position)

Stosowanie się do tego schematu informuje wszelkie graficzne programy wykorzystujące ziarna że to ziarno zawiera właściwości indeksowe.

Przykład:

Modyfikacja komponentu List (z biblioteki AWT), polegająca na dodaniu właściwości indeksowanej o nazwie item:

```
public class ListBean extends List {
  public String[] getItem () {
    return getItems ();
  }
  public synchronized void setItem (String item[]) {
    removeAll();
    for (int i=0;i<item.length;i++)
      addItem (item[i]);
  }
  public void setItem (String item, int position) {
    replaceItem (item, position)
  }
}
```

gdzie metoda String getItem (int position) jest zdefiniowana w klasie List.

Inne przykłady (właściwości indeksowane wiązane):

```
public void setItems(String[] indexprop) {
        String[] oldValue=fieldIndexprop;
        fieldIndexprop=indexprop;
        populateListBox();
        support.firePropertyChange("items",oldValue, indexprop);
}
public void setItems(int index, String indexprop) {
        String[] oldValue=fieldIndexprop;
        fieldIndexprop[index]=indexprop;
        populateListBox();
        support.firePropertyChange("Items",oldValue, fieldIndexprop);
}
public String[] getItems() {
        return fieldIndexprop;
}
public String getItems(int index) {
        return getItems()[index];
}
```

2.2 Metody

Metody ziaren to operacje, które dostarczane są po to, aby użytkownicy mogli na ziarna oddziaływać. Metody ziaren typu public są dostępne dla wszystkich użytkowników. Metody typu private wspierają działanie ziaren (jednak użytkownicy nie mogą już z nich korzystać).

W ziarnach występują metody związane z właściwościami, bezargumentowe metody wspierające działanie Beans lub metody pobierające jeden argument – zdarzenie, na które się oczekuje.

Każdy Ziarno może wspierać klasa ZiarnoBeanInfo, która zawiera informacje na jego temat. Dzięki BeanInfo istnieje możliwość kontroli (ograniczenia) dostępu do metod publicznych dla narzędzi służący do budowy aplikacji na podstawie ziaren.

Narzędzia do budowy aplikacji na podstawie ziaren przeglądają ziarna używając metody getMethodDescriptors. Dzięki niej szybko identyfikowane są metody, które odpowiadają wzorcowi projektowemu, a które stanowią o indywidualnych cechach ziarna. Metody publiczne, które nie odpowiadają wzorcowi projektowego, mogą zostać nierozpoznane.

2.3 Zdarzenia

Zdarzenia pozwalają ziarnom na komunikację – informują zajściu jakiejś interesującej sytuacji w systemie (np. upływ czasu, pojawienie się nowej informacji, wybranie kontrolki ziarna przez użytownika(). W ziarnach Javy zastosowano model zdarzeń zapożyczony z biblioteki AWT, a wprowadzony w Javie 1.1. Model ten składa się z trzech części:

- `EventObjects` – zdarzenia
  - `AWTEvent` z biblioteki AWT
- `EventListeners` – słuchacze
  - `ActionListener`, `ItemListener`, ...
- Event Sources (Beans) – źródła zdarzeń
  - Component (Bean)

Każdy może zarejestrować słuchacza dla obiektu Component, pod warunkiem, że komponent rozumie zbiór zdarzeń (np. nie można zarejestrować ActionListener dla TextArea, ale dla TextField tak). Kiedy coś się stanie wewnątrz obiektu Component, słuchacz jest zawiadamiany za pomocą EventObject wysłanego do odpowiedniej metody słuchacza.

2.3.1 EventObject

Podstawę dla wszystkich zdarzeń w ziarnach jest klasa java.util.EventObject

```
public class java.util.EventObject
    extends Object implements java.io.Serializable {
  public java.util.EventObject (Object source);
  public Object getSource();
  public String toString();
}
```

Dla potrzeb własnego ziarna można stworzyć własną klasę zdarzeń, musi jednak ona dziedziczyć po EventObject.

Przykład: klasa służąca do tworzenia instancji zdarzeń, które pojawiać się będą w chwilach zatrudnienia nowego pracownika. Zdarzenia te nieść będą ze sobą informacje o czasie:

```
public class HireEvent extends EventObject {
  private long hireDate;
  public HireEvent (Object source) {
    super (source);
    hireDate = System.currentTimeMillis();
  }
  public HireEvent (Object source, long hired) {
    super (source);
    hireDate = hired;
  }
  public long getHireDate () {
    return hireDate;
  }
}
```

2.3.2 EventListener

Interfejs EventListner (słuchacz zdarzeń) jest pusty. Wszyscy słuchacze implementowani dla potrzeb ziarna muszą z tego interfejsu dziedziczyć. Słuchacz jest obiektem, który zostaje poinformowany, gdy zachodzi zdarzenie. Otrzymuje on jako parametr obiekt klasy, dziedziczącej z EventObject (tj. instancję reprezentującą zdarzenie). Nazwa nowego interfejsu powstaje według wzoru *EventType*Listener. Dla nowego zdarzenia HireEvent, nazwą słuchacza powinna być HireListener. Nazwy metod wewnątrz interfejsu słuchacza nie są wyspecyfikowane, ale powinny opisywać zdarzenie.

```
public interface HireListener
   extends java.util.EventListener {
  public abstract void hired (HireEvent e);
}
```

2.3.3 Event Source

Instancje klas HireEvent i HireListener są bezużyteczne, jeśli nie utworzone zostanie źródło zdarzeń. Źródło zdarzeń określa, kiedy i gdzie zachodzi zdarzenie.

Źródło zdarzeń powinno pozwalać na rejestrację słuchaczy (tj. obiektów zainteresowanych otrzymywaniem zdarzeń), aby w przypadku zajścia zdarzenia móc je słuchaczom przekazać. Grupa wzorców metod reprezentujących proces takiej rejestracji:

```
public synchronized void add<ListenerType>(<ListenerType> l);
public synchronized void remove<ListenerType>(<ListenerType> l);
```

Mechanizm rejestracji można zaimplementować jak w przykładzie poniżej:

```
private Vector hireListeners = new Vector();
public synchronized void addHireListener (HireListener l) {
  hireListeners.addElement (l);
}
public synchronized void removeHireListener (HireListener l) {
  hireListeners.removeElement (l);
}
```

Zdarzenia AWT oraz komponenty AWT oraz Swing mają już zdefiniowane powyższe zachowanie. W ich przypadku tworzenie słuchaczy konieczne jest tylko dla nowych typów zdarzeń lub dodawania słuchaczy tam gdzie ich wcześniej nie było.

Ilość zarajestrowanych słuchaczy można kontrolować poprzez wyrzucanie wyjątków. W tym celu metoda add*ListenerType* powinna wyrzucać wyjątek java.util.TooManyListenersException w momencie dodawania nadmiarowego słuchacza.

Powiadamianie słuchaczy o zajściu zdarzenie można zaimplementować jak w przykładzie poniżej (przeglądanie listy słuchaczy i odpalenie zdarzenia dla każdego z nich):

```
protected void notifyHired () {
  Vector l;

  // Utworzenie zdarzenia
  HireEvent h = new HireEvent (this);

  // skopiowanie wektora słuchaczy, aby nie dopuścić
  // do jego zmiany podczas odpalania zdarzeń
  synchronized (this) {
    l = (Vector)hireListeners.clone();
  }
 // odpalenie zdarzenie
  for (int i=0;i<l.size();i++) {
    HireListener hl = (HireListener)l.elementAt (i);
    hl.hired(h);
  }
}
```

3 BeanInfo

3.1 Tworzenie klasy informacyjnej

Introspekcja jest procesem określania dostępnych właściwości, metod i zdarzeń Ziarna. Proces ten wykonywany jest często w aplikacjach służących do graficznego projektowania aplikacji. Introspekcja może być zrobiona za pomocą klasy Introspector, (bazującej na Reflection API). Daje to możliwość poznania wszystkie tajników ziarna.
Aby ograniczyć zakres widzialnych szczegółów ziarna można wyłączyć domyślny mechanizm bazujący na Reflection API tworząc klasę informacyjną ziarna. Klasa taka powinna implementować interfejs BeanInfo. Interfejs ten zawiera 8 metod. Można też alternatywnie rozszerzyć klasę SimpleBeanInfo, która posiada puste implementacje metod interfejsu BeanInfo, przysłaniając tylko niezbędne metody.

Kontrola dostępu do cech ziarna w klasie typu BeanInfo realizowana jest przez metody zwracające opisy (deskryptory) ujawnionych metod, właściwości i zdarzeń. Aby system odnalazł klasę typu BeanInfo dla konkretnego Ziarna, musi ona mieć nazwę ZiarnoBeanInfo. Całość operacji przy ujawnianiu lub urywaniu cech jest bardzo prosta. Jeżeli jakaś cecha ma zostać ujawniona, należy w klasie BeanInfo napisać dla niej deskryptor. W przypadku braku deksryptora cechy pozostaną ukryte. Klasę *BeanInfo* należy umieszczać w tym samym katalogu, co ziarno, gdyż właśnie od niej rozpoczyna się odkrywanie cech ziarna. Dokładniej mówiąc, do znalezienia klasy BeanInfo wykorzystywana jest metoda getBeanInfo klasy Introspector. Jeśli BeanInfo nie zostanie znaleziona w ścieżce, wtedy przeszukiwana jest domyślna lokalizacja (sun.beans.infos). Jeżeli i tam poszukiwanie nie da rezultatu, na ziarnie zostanie przeprowadzona analiza niskopoziomowa (w przypadku braku BeanInfo wykorzystywane jest Reflection API).

```
TextField tf = new TextField ();
BeanInfo bi = Introspector.getBeanInfo (tf.getClass());
```

Klasa BeanInfo:

- pozwala ujawnić tylko te cechy, które mają być ujawniane, podczas gdy niskopoziomowa analiza ujawni inne
- umożliwia przypisanie ikony dla ziarna
- pozwala określić edytory właściwości ziarna używane w okienku inspektora
- pozwala wyspecyfikować klasę indywidualizacji ziarna, *customizer*
- pozwala posegregować cechy w kategoriach Normal i Expert
- daje możliwość dołączania dodatkowych informacji na temat ziarna

Metody pozwalające na utworzenie opisu ziarna (które należy zaimplementować w klasie BeanInfo):

BeanInfo[] getAdditionalBeanInfo()	pozwala obiektowi BeanInfo zwrócić arbitralną kolekcję innych obiektów BeanInfo, które dostarczają dodatkowych informacji o bieżącym ziarnie
BeanDescriptor getBeanDescriptor()	zwraca BeanDescriptor
int getDefaultEventIndex()	ziarno może mieć "domyślne" zdarzenie, które jest zdarzeniem najczęściej używanym przez użytkowników ziarna.
int getDefaultPropertyIndex()	ziarno może mieć "domyślną" właściwość, która jest najczęściej wybierana przez użytkowników podczas modyfikacji ziarna.
EventSetDescriptor[] getEventSetDescriptors()	zwraca EventSetDescriptors.
Image getIcon(int iconKind)	zwraca obiekt obrazka, który będzie użyty do reprezentacji ziarna w toolboxes, toolbars, etc.
MethodDescriptor[] getMethodDescriptors()	zwraca MethodDescriptors.
PropertyDescriptor[] getPropertyDescriptors()	zwraca PropertyDescriptors.

Deskryptory zwracane przez metody muszą być utworzone w BeanInfo. Deskryptory te mają różne konstruktory (do przekazania w parametrach ziarna, nazwy własności, metod set i get).

Nazwa klasy	Opis
FeatureDescriptor	jest klasą bazową dla innych deskryptorów
BeanDescriptor	opisuje typ klasy ziarna i jego nazwę oraz klasę Customizer jeżeli taka istnieje
PropertyDescriptor	opisuje właściwości ziarna (tj. nazwę właściwości ziarna oraz klasę ziarna). Za pomocą metody tej klasy setPropertyEditorClass można określić edytor właściwości (zobacz Edycja właściwości ziarna).
IndexedPropertyDescriptor	jest podklasą PropertyDescriptor i opisuje właściwości indeksowe ziarna
EventSetDescriptor	opisuje zdarzenia obsługiwane przez ziarno
MethodDescriptor	opisuje metody zawarte w ziarnie
ParameterDescriptor	opisuje parametry metod

Przykład (w klasie ZiarnoBeanInfo):

```
public BeanDescriptor getBeanDescriptor() {
// beanClass – parametr, który został przekazany w konstruktorze ZiarnoBeanInfo
     BeanDescriptor bd = new BeanDescriptor(beanClass);
    bd.setDisplayName("Uneasy Text");
    return bd;
  }

public PropertyDescriptor[] getPropertyDescriptors() {
    try {
      PropertyDescriptor textPD = new PropertyDescriptor("text", beanClass);
      PropertyDescriptor rv[] = {textPD};
      return rv;
    } catch (IntrospectionException e) {
      throw new Error(e.toString());
    }
  }

public MethodDescriptor[] getMethodDescriptors() {
  // First find the "method" objects.
  Method startMethod, stopMethod, changeDirectionMethod;
  Method propertyChangeMethod;
  Class args[] = { };
  Class actionEventArgs[] = { java.awt.event.ActionEvent.class };
  Class propertyChangeEventArgs[] = { PropertyChangeEvent.class };

  try {
    startMethod = Ziarno.class.getMethod("start", args);
    stopMethod = Ziarno.class.getMethod("stop", args);

    // metoda obsługi zdarzeń ActionEvent:  public void changeDirection(ActionEvent x)
    changeDirectionMethod =  Ziarno.class.getMethod("changeDirection", actionEventArgs);

    // metoda obsługi zdarzeń PropertyChangeEvent: public void makeChange ( PropertyChangeEvent evt)
    propertyChangeMethod = Ziarno.class.getMethod("makeChange", propertyChangeEventArgs);

  } catch (Exception ex) {
    // "should never happen"
    throw new Error("Missing method: " + ex);
  }

  // Utworzenie tablicy MethodDescriptor z widocznymi metodami obsługi zdarzeń
  MethodDescriptor result[] = {
    new MethodDescriptor(startMethod),
    new MethodDescriptor(stopMethod),
    new MethodDescriptor(changeDirectionMethod),
    new MethodDescriptor(propertyChangeMethod)
  };
```

```
  return result;
}
```

Konstruktory klas wszystkich deskryptorów działają w podobny sposób. W przypadku deskryptorów zdarzeń najczęściej używany jest konstruktor z 4 parametrami:

- klasa ziarna,
- podstawowa nazwa zdarzenia,
- klasa implementująca interfejs EventListener dla danego zdarzenia,
- metody interfejsu EventListener wyzwalane przez zdarzenie.

Inne konstruktory pozwalają określić metody ziarna do rejestrowania i werejestrowywania słuchaczy. Po utworzeniu deskryptorów zdarzeń należy jeszcze określić nazwy zdarzeń. Dzieje się tak dlatego, iż nazwy zdarzeń nie są zgodne ze wzorcem projektowym (bo nie ma w nich nazwy ziarna z przyrostkiem Listener, oraz nazwy zdarzeń wyświetlanych w okienku inspektora nie muszą pokrywać się z nazwami rzeczywistych zdarzeń)

```
public EventSetDescriptor[] getEventSetDescriptors() {
   try {
      EventSetDescriptor push = new EventSetDescriptor(beanClass,
               "actionPerformed",
               java.awt.event.ActionListener.class,
               "actionPerformed");

      EventSetDescriptor changed = new EventSetDescriptor(beanClass,
               "propertyChange",
               java.beans.PropertyChangeListener.class,
               "propertyChange");

      push.setDisplayName("button push");
      changed.setDisplayName("bound property change");

      EventSetDescriptor[] rv = { push, changed};
      return rv;
   } catch (IntrospectionException e) {
      throw new Error(e.toString());
   }
}
```

W ogólności wszędzie tam, gdzie cechy ziarna nie odpowiadają wzorcom projektowym należy:

- utworzyć deskryptory wszystkich cech danego zbioru (zdarzeń, właściwości, metod)
- zwrócić tablicę deskryptorów za pomoca odpowiedniej metody w BeanInfo

Należy tutaj zaznaczyć jeszcze dwie istotne rzeczy:

- Jeżeli opuścisz deskryptor właściwość, metoda albo zdarzenie nie zostanie ujawnione.
- Jeżeli metoda get danej cechy zwraca null, to do tej cechy wykorzystywana jest analiza niskopoziomowa

3.2 Tryb pracy i tryb projektowania ziaren

Ziarno musi być przygotowane do pracy w aplikacji jak i podczas jej graficznego projektowania. W czasie projektowania Ziarno musi dostarczyć informacji niezbędnych do edycji jego właściwości i zachowań. Środowisko musi też mieć informacje na temat dostępnych metod i zdarzeń, by móc wygenerować kod, który będzie współdziałał z Ziarnem w aplikacji. Na sprawdzenia aktualnego trybu pracy pozwala metoda Beans.isDesignTime.

3.2.1 Właściwości

Metoda getPropertyDescriptors dostarcza wszystkie dostępne właściwości ziarna.

Przykład:

```
PropertyDescriptor pd[] = bi.getPropertyDescriptors();
for (int i=0;i<pd.length;i++)
  System.out.print (pd[i].getName() + " ");
System.out.println ();
```

Dla bi będącego ziarnem typu TextField zostanie wydrukowane:
selectionStart enabled text preferredSize foreground visible background selectedText echoCharacter font columns echoChar name caretPosition selectionEnd minimumSize editable

Przykład:

Niech będzie klasa SizedTextField, oraz klasa SizedTextFieldBeanInfo, która będzie dostarczać BeanInfo pozwalające rozpoznać tylko jedną właściwość (length) ziarna narzędziom graficznego tworzenia aplikacji:

```
import java.beans.*;
public class SizedTextFieldBeanInfo
    extends SimpleBeanInfo {

  private final static Class beanClass = SizedTextField.class;

  public PropertyDescriptor[] getPropertyDescriptors() {
    try {
      PropertyDescriptor length = new PropertyDescriptor("length", beanClass);
      PropertyDescriptor rv[] = {length};
      return rv;
    } catch (IntrospectionException e) {
      throw new Error(e.toString());
    }
  }
}
```

Powyższy kod sprawia, iż w narzędziu dostępna jest tylko właściwość length (zamiast 17 zwykle dostarczanych przez TextField). Wszystkie niewidoczne w narzędziu metody są nadal dostępne (nie przestały być publiczne) - można się do nich odwołać albo po nazwie albo przez Reflection API.
W celu nadania właściwości jakiejś nazwy zawierającej znaki spacji należy umieścić w SizedTextFieldBeanInfo co następuje:

```
public BeanDescriptor getBeanDescriptor() {
  BeanDescriptor bd = new BeanDescriptor(beanClass);
  bd.setDisplayName("Sized Text Field");
  return bd;
}
```

Ikonę dla tworzonego ziarna dodaje się przez umieszczenie w SizedTextFieldBeanInfo poniższego kodu:

```
public Image getIcon (int iconKind) {
  if (iconKind == BeanInfo.ICON_COLOR_16x16) {
    Image img = loadImage("sized.gif");
    return img;
  }
  return null;
}
```

3.2.2 Zdarzenia

Metoda getEventSetDescriptors dostarcza listę zdarzeń, które generuje ziarno (dla przypomnienia: słuchaczy rejestruje się i wyrejestrowuje się metodami add/remove*ListenerType*Listener).

```
EventSetDescriptor[] esd = bi.getEventSetDescriptors();
for (int i=0;i<esd.length;i++)
  System.out.print (esd[i].getName() + " ");
System.out.println ();
```

Dla bi będącego ziarnem typu TextField zostanie wydrukowane:
text mouse key component action focus mouseMotion

3.2.3 Metody

Metoda getMethodDescriptors dostarcza wszystkie metody ziarna. Jest to pełna lista wszystkich publicznych metod, które można wywołać bez uprzedniej znajomości ich nazw.
getParameterDescriptors pozwala na podstawie deskryptora metody poznać jej nazwę i parametry.

```
MethodDescriptor md[] = bi.getMethodDescriptors();
for (int i=0;i<md.length;i++)
  System.out.print (md[i].getName() + " ");
System.out.println ();
```

Dla bi będącego ziarnem typu TextField zostaną wydrukowane nazwy wszystkich 155 dostępnych metod, większość z nich jest odziedziczona z klasy Component.

3.3 Edycja właściwości ziarna

Właściwości o typach standardowych edytowane są w narzędziach graficznego tworzenia aplikacji za pomocą domyślnych edytorów. Edytory te zazwyczaj są edytowalnym polem tekstowym. W przypadku ziaren o właściwościach zaimplementowanych przez użytkownika należy dostarczyć własny edytor.

Okienko inspektora zazwyczaj jest niewielkie, więc edytor nie ma zbyt wiele miejsca na wyświetlanie wartości własności (bądź reprezentacji własności). Aby z poziomu okienka inspektora można było edytować złożone właściwości ziaren dostarcza się, oprócz edytora wyświetlającego reprezentację ziarna w okienku inspektora, komponent umożliwiający edycję złożonych własności. Komponent ten pojawia się po kliknięciu w polu własności okienka inspektora.

Domyślne edytory dla właściwości tego samego typu można określać, korzystając z metody statycznej registerEditor klasy PropertyEditorManager, np.:

```
PropertyEditorManager.registerEditor(Date.class, CalendarSelector.class)
```

Wybór domyślnego edytora należy do narzędzia tworzenia aplikacji. Same ziarna nie powinny wywoływać metody registerEditor.

O tym, czy narzędzie tworzenia aplikacji dysponuje edytorem właściwości określonego typu można przekonać się wywołując metodę findEditor klasy PropertyEditorManager. Metoda ta:

1. sprawdza, jakie edytory zostały zarejestrowane (czyli jakie edytory dostarczyło samo narzędzie tworzenia aplikacji oraz jakie edytory zostały do tej pory zarejestrowane)
2. poszukuje klas, których nazwa odpowiada nazwie typu z dodanym przedrostkiem Editor.
3. zwraca null, jeśli edytor nie został znaleziony

W celu utworzenia własnych edytorów należy:

- zaimplementować interfejs PropertyEditor (posiadającej 12 metod) lub rozszerzyć klasę PropertyEditorSupport, lub

- dokonać indywidualizacji ziarna, tj. dostarczyć klasę Customizer (konwencja nazw, choć nie jest wymagana, zakłada nazwę BeanCustomizer, tj. Customizer z nazwą ziarna na początku) która jest czymś w rodzaju podprogramu służącego do kontroli wyglądu i zachowania się ziarna.

W ogólności instancje edytorów tworzone są przez narzędzia tworzenia aplikacji dla każdej właściwości ziarna. Narzędzia te żądają od ziarna podania bieżącej wartości właściwości, aby zlecić edytorom ich wyświetlenie. Dany edytor wiąże się z właściwością ziarna w klasie BeanInfo, dzięki czemu narzędzia tworzenia aplikacji mogą rozpoznać, co mają wyświetlać w okienku inspektora właściwości ziarna.

3.3.1 PropertyEditor

Do umożliwienia edycji właściwości prostych wystarczy zaimplementować metody getAsText oraz setAsText interfejsu PropertyEditor. W poniższym przykładzie skorzystano z klasy PropertyEditorSupport, która ma zaimplementowane wszystkie metody interfejsu. Aby uzyskać klasę edytora wystarczyło tylko przysłonić wybrane metody.

```
public class TitlePositionEditor extends PropertyEditorSupport{
 public String getAsText(){
    int i = ((Integer)getValue().intValue());
    return options[value];
 }
 public void setAsText(String s) {
   for(int i=0; i<options.length;i++) {
      if(options[i].equals(s)){
      setValue(new Integer(i));
      return;
      }
   }
 }
  private String[] options = {"Left", "Center", "Right"};
  public String[] getTags() { return options; }
}
```

oraz umieścić w klasie BeanInfo co następuje:

```
public PropertyDescriptor[] getPropertyDescriptors(){
 try{
       PropertyDescriptor titlePositionDescriptor = new PropertyDescriptor("titlePosition",
Ziarno.class);
       titlePositionDescriptor.setPropertyEditorClass(TitlePositionEditor.class);
      ....
 } catch(IntrospectionException e)
{
...}
}
```

Aby utworzyć własny edytor właściwości należy wykonać następujące kroki:

1. poinformować narzędzie, że sami będziemy tworzyć reprezentację graficzną wartości przysłaniając dwie metody interfejsu PropertyEditor:

```
public String getAsText() { return null;}
public boolean isPaintable() { return true;}
```

2. stworzyć reprezentację graficzną wartości implementując metodę paintValue

```
public void paintValue(Graphics g, Rectangle box){
// g – kontekst graficzny, box- obszar rysowania
.....
}
```

3. poinformować narzędzie, że będziemy korzystać z graficznego edytora właściwości zastępując metodę supportCustomEditor interfejsu PropertyEditor

```
public boolean supportCustomEditor(){ return true;}
```

4. utworzyć graficzny interfejs właściwości i związać go z edytorem (nie możemy zapomnieć o zdarzeniu PropertyChanged, które odpalane jest dla listy słychaczy przechowywanej przez PropertyEditor po zmianie właściwości)

```
public class CustomEditorPanel extends JPanel{
 public CustomEditorPanel(PropertyEditorSupport ed){
   // przechowaj referencję do edytora właściwości ziarna, aby móc zmodyfikować
właściwość ziarna
    PropertyEditorSupport editor = ed;

    // utworz panel z polami, przyciskami, itd.
   ......
   // jeśli modyfikacja własności związana jest z obsługą zdarzenia naciśnięcia przycisku, to
    ActionListener bl = new ActionListener(){
          public void actionPerformed(ActionEvent e){
                editor.setValue( .....);
                 editor.firePropertyChange();
          }
         };
     przycisk.addActionListener(bl);
 }
}
```

```
// w klasie rozszerzającej PropertyEditorSupport
public Component getCustomEditor(){
     return new CustomEditorPanel(this);
}
```

5. zaprogramować reguły walidacji wartości wprowadzonych przez użytkownika

3.3.1.1 Implementowanie edytora właściwości indeksowych.

```
// Implementacja interfejsu PropertyEditor
public class IndexPropertyEditor extends Panel implements PropertyEditor, ActionListener {

// Wskazanie edytora w spokrewnionej klasie BeanInfo
        itemsprop.setPropertyEditorClass(IndexPropertyEditor.class);

// Zrobienie z edytora wartości źródła dla zdarzeń wartości wiązanych. Edytor będzie rejestrował
// słuchaczy i generował dla nich PropertyChangeEvent.
// Dlatego IndexPropertyEditor musi utworzyc nowy obiekt klasy PropertyChangeSupport
        private PropertyChangeSupport support = new PropertyChangeSupport(this);

// Daje to obiektom możliwość rejestrowania swojego zainteresowania edycją właściwości.
        public void addPropertyChangeListener(PropertyChangeListener l) {
              support.addPropertyChangeListener(l);
        }
       public void removePropertyChangeListener(PropertyChangeListener l) {
              support.removePropertyChangeListener(l);
        }
```

```
// I generuje dla tych słuchaczy PropertyChangeEvent
        public void actionPerformed(ActionEvent evt) {
            if (evt.getSource() == addButton) {
                listBox.addItem(textBox.getText());
                textBox.setText("");
                support.firePropertyChange("", null, null);
             } else if (evt.getSource()== textBox) {
                        listBox.addItem(textBox.getText());
                        textBox.setText("");
                        support.firePropertyChange("",null,null);
        }
...
}
```

3.3.2 Customizers

Customizers to klasy służące do definiowana okienek edycyjnych właściwości ziarna, które można dostarczyć wraz z nim. Dzięki zastosowaniu tego rozwiązania można mieć pełny wpływ na to, jak będzie wyglądała karta właściwości naszego komponentu w narzędziu do graficznego tworzenia aplikacji.
Klas tych używa się zamiast PropertyEditor, jeśli PropertyEditor okazuje się zbyt prymitywny czy też ograniczony. Wszystkie obiekty pełniące role *customizera* muszą:

- Dziedziczyć po jawa.awt.Component lub jednej z jego podklas (JPanel, JTabbedPane – choć dla komponentów Swing mogą pojawić się problemy).
- Implementować interfejs java.beans.Customizer zawierający 3 metody:
    - setObject z parametrem określającym ziarno
    - addPropertyChangeListener (rejestracja słuchaczy zdarzeń generowanych przy zmianie właściwości ziarna)
    - removePropertyChangeListener (wyrejestrowywanie słuchaczy)

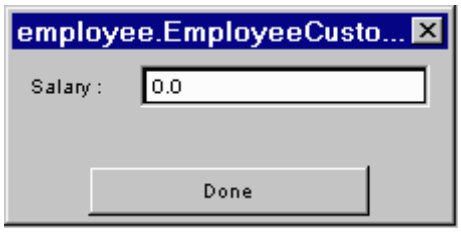
  Ostatnie dwie metody związane są z tym, że po każdej zmianie właściwości powinno odświeżyć się wygląd ziarna przez wysłanie zdarzenia PropertyChangeEvent.
- Być dołączane do swojej klasy poprzez BeanInfo.getBeanDescriptor.

W przypadku Customizers okna indywidualizacji (tj. okna z edytowanymi cechami) nie są otwierane automatycznie. W narzędziach tworzenia aplikacji zazwyczaj należy wybrać z menu pozycję Edit/Customize, która spowoduje wykonanie metody setObject.
Dla przykładu zbudujmy Customizer dla ziarna Employee, który posiada tylko jedną właściwość salary. Niech okienko wprowadzania tej właściwości przyjmuje tylko znaki numeryczne.

```
package employee;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;

public class EmployeeCustomizer extends Panel
   implements Customizer, KeyListener {
  private Employee target;
  private TextField salaryField;
  private PropertyChangeSupport support = new PropertyChangeSupport(this);
  public void setObject(Object obj) {
    target = (Employee) obj;
    Label t1 = new Label("Salary :");
    add(t1);
    salaryField = new TextField( String.valueOf(target.getSalary()), 20);
    add(salaryField);
```

```
    salaryField.addKeyListener(this);
  }
  public Dimension getPreferredSize() {
    return new Dimension(225,50);
  }
  public void keyPressed(KeyEvent e) {}
  public void keyTyped(KeyEvent e) {}
  public void keyReleased(KeyEvent e) {
    Object source = e.getSource();
    if (source==salaryField) {
      String txt = salaryField.getText();
      try {
        target.setSalary( (new Float(txt)).floatValue() );
      } catch (NumberFormatException ex) {
        salaryField.setText( String.valueOf(target.getSalary()) );
      }
      support.firePropertyChange("", null, null);
    }
  }
  public void addPropertyChangeListener(PropertyChangeListener l) {
    support.addPropertyChangeListener(l);
  }
  public void removePropertyChangeListener(PropertyChangeListener l) {
    support.removePropertyChangeListener(l);
  }
}
```

W narzędziu do tworzenia aplikacji edycja właściwości naszego komponentu będzie wyglądać w następujący sposób:



Oczywiście przy założeniu, została zdefiniowana klasa EmployeeBeanInfo:

```
package employee;
import java.beans.*;
public class EmployeeBeanInfo extends SimpleBeanInfo {
  public BeanDescriptor getBeanDescriptor() {
    return new BeanDescriptor( beanClass, customizerClass);
  }
  private final static Class beanClass =  Employee.class;
  private final static Class customizerClass =  EmployeeCustomizer.class;
}
```

3.4 Serializacja Bean

To persist, your Beans must support serialization by implementing either the java.io.Serializable interface, or the java.io.Externalizable interface. These interfaces offer you the choice between automatic serialization, and "roll your own". As long as one class in a class's inheritance hierarchy implements Serializable or Externalizable, that class is serializable.

Controlling Serialization

You can control the level of serialization that your Beans undergo:

- Automatic: implement Serializable. Everything gets serialized by the Java serialization software.
- Selectively exclude fields you do not want serialized by marking with the transient (or static) modifier.
- Writing Beans to a specific file format: implement Externalizable, and its two methods.

Seralizacja obiektów następuje przez wywołanie ObjectOutput.writeObject z serializowanym obiektem jako parametrem. Deserializacja uzyskiwana jest przez wywołanie metody ObjectInput.readObject. W celu upewnienia się, że budowane ziarno nadaje się do serializacji należy odpowiedzieć na pytania

- Czy stworzona klasa jest serializowana?
- Czy wszystkie instancje zmiennych są serializowane?
- Czy aby wszystko co nie jest zadeklarowane jako transient lub static zostało zapisane?
- Czy obiekt powinien zostać zserializowany w swojej aktualnej strukturze?
- Jak powinny zostać zainicjowane zmienne transient i static po deserializacji?
- Czy chcesz dodać walidator do procesu serializacji?

3.5 Rekonstrukcja Bean

Normalnie do stworzenia nowej instancji obiektu używany jest operator new. W Beans można też użyć metody Beans.instantiate która jest metodą używaną przez narzędzia tworzenia aplikacj do odtworzenia zserializowanych ziaren. To rozwiązania ma na celu umożliwienie narzędziom tworzenie obiektów Bean, które nie były znane podczas tworzenia narzędzia

```
Component c =  (Component)Beans.instantiate(null, "java.awt.TextField")
```

3.6 Wersje w Beans

Kiedy powstaje kolejna wersja Bean to w pewnym momencie może dojść do sytuacji kiedy ktoś będzie próbował odzyskać zserializowaną starą wersję do wersji nowej komponentu. Może to spowodować konflikt i pojawienie się wyjątku java.io.InvalidClassException. Rozwiązaniem jest wprowadzenie w definicji naszej klasy zmiennej Stream Unique Identifier (SUID).

```
private static final long serialVersionUID =    -2966288784432217853L;
```

Wartość zmiennej nie jest liczbą losową lecz wynikiem działania specjalnego algorytmu haszujacego. Do wygenerowania jej może posłużyć program serialver który jako parametr przyjmuje nazwę klasy. Można też uruchomić ten program z GUI - służy do tego przełącznik –show.

3.6.1.1 Tworzenie klasy BeanInfo

Nazwę klasy tworzymy przez dodanie do nazwy klasy słów *BeanInfo*. wiec jeżeli nazwa klasy to *MojPrzycisk* to klasa BeanInfo będzie się nazywać *MojPrzyciskBeanInfo*.

Podklasa *SimpleBeanInfo* jest wygodną klasą implementującą metody klasy *BeanInfo* zwracające wartości *null* i równoważną *no-op*. Użycie tej klasy jest dużym ułatwieniem, ponieważ zaoszczędza nam implementowania wszystkich metod BeanInfo i zmusza nas tylko do przykrycia metod, które są nam potrzebne.

public class MojPrzyciskBeanInfo extends SimpleBeanInfo {

Na tym przykładzie pokaże w jaki sposób można przykryć metodę aby wykonywała oczekiwane przez nas zadania. MojPrzyciskBeanInfo przykrywa metodę *getPropertyDescriptors()* tak aby zwracała ona cztery wartości :

```
public PropertyDescriptor[] getPropertyDescriptors() {
try {
PropertyDescriptor background = new PropertyDescriptor("background", beanClass);
PropertyDescriptor foreground = new PropertyDescriptor("foreground", beanClass);
PropertyDescriptor font = new PropertyDescriptor("font", beanClass);
PropertyDescriptor label = new PropertyDescriptor("label", beanClass);

background.setBound(true);
```

```
foreground.setBound(true);
font.setBound(true);
label.setBound(true);

PropertyDescriptor rv[] = {background, foreground, font, label};
return rv;
} catch (IntrospectionException e) {
throw new Error(e.toString());
}
}
```

3.6.1.2 Dodawanie ikony do ziarna

Aby ziarno można było dodać do paska zadań w jakimkolwiek edytorze graficznym należy dołączyć do niego ikonę

```
public java.awt.Image getIcon(int iconKind) {
if (iconKind == BeanInfo.ICON_MONO_16x16 || iconKind == BeanInfo.ICON_COLOR_16x16 ) {
java.awt.Image img = loadImage("MojPrzyciskIcon16.gif");
return img;
}
if (iconKind == BeanInfo.ICON_MONO_32x32 || iconKind == BeanInfo.ICON_COLOR_32x32 ) {
java.awt.Image img = loadImage("MojPrzyciskIcon32.gif");
return img;
}
return null;
}
```

3.6.1.3 Specyfikowanie klasy opisywanej przez BeanInfo

```
public BeanDescriptor getBeanDescriptor() {
return new BeanDescriptor(beanClass, customizerClass);
}
...
private final static Class beanClass = MojPrzycisk.class;
private final static Class customizerClass = MojPrzyciskCustomizer.class;
```

4 Przykłady

Przyklad 1 – Ziarno

```
import java.awt.*;
import javax.swing.JPanel;
import java.io.*;
import java.lang.String;
import java.awt.Color;

public class proste extends JPanel implements Serializable {

  private java.awt.Color kolor = java.awt.Color.yellow;
  BorderLayout borderLayout1 = new BorderLayout();
  int a,b;

  public proste(){  }

  public void paint(Graphics gDC) {
    a=getWidth();
    b=getHeight();
    gDC.setColor(kolor);
```

```
    gDC.fillRect(0,0,a,b);
  }

  public void setKolor( java.awt.Color kolor ) {
    this.kolor = kolor;
  }

  public java.awt.Color getKolor() {
    return kolor;
  }

  private void writeObject(ObjectOutputStream oos) throws IOException {
    oos.defaultWriteObject();
  }

  private void readObject(ObjectInputStream ois)
        throws ClassNotFoundException, IOException {
    ois.defaultReadObject();
  }
}
```

Przykład 1 – Program

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.lang.String;
import java.awt.Color;

public class prostep {
 public static void main(String[] args) {
  Okno tProgram = new Okno("Właściwości Proste");
  tProgram.setSize(300,300);
  tProgram.setBackground(java.awt.Color.lightGray);
  tProgram.show();
 }
}

class Okno extends Frame {
 private Button zielony = new Button("Zielony");
 private Button czerwony = new Button("Czerwony");
 private Button niebieski = new Button("Niebieski");
 private Button jaki = new Button("Jaki Kolor?");
 private Panel przyciski = new Panel();
 private Panel skladowe = new Panel();
 private proste bean = new proste();
 private Label rgb = new Label("Tu pojawia sie skladowe koloru.",Label.CENTER);

 public Okno(String nazwa) {
  super(nazwa);
  setLayout(new BorderLayout());
  add(przyciski,BorderLayout.NORTH);
  przyciski.setLayout(new FlowLayout());
  przyciski.add(zielony);
  przyciski.add(czerwony);
```

```
  przyciski.add(niebieski);
  add(bean, BorderLayout.CENTER);
  add(skladowe, BorderLayout.SOUTH);
  skladowe.add(jaki);
  skladowe.add(rgb);

  zielony.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    bean.setKolor(java.awt.Color.green);
    bean.repaint();
   }
  });
  czerwony.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    bean.setKolor(java.awt.Color.red);
    bean.repaint();
   }
  });
  niebieski.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    bean.setKolor(java.awt.Color.blue);
    bean.repaint();
   }
  });
  jaki.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    java.awt.Color a;
    int r,g,b;
    a=bean.getKolor();
    r=a.getRed();
    g=a.getGreen();
    b=a.getBlue();
    rgb.setText("Red="+ r +" Green=" + g + " Blue=" + b);
   }
  });

  addWindowListener(new ProgramWindowAdapter());
 }
}
class ProgramWindowAdapter extends WindowAdapter {
 public void windowClosing(WindowEvent e) {
  System.exit(0);
 }
}
```

Przyklad 2 - Ziarno 1

```
import java.awt.*;
import java.io.*;
import java.beans.*;
import java.awt.event.*;

public class wiazane1 extends TextField{
  private String zmiennawiazana;
  private transient PropertyChangeSupport propertyChangeListeners = new
```

```
PropertyChangeSupport(this);

  public wiazane1() {
  }

  private void jbInit() throws Exception {
    this.setText("Tu wpisz tekst");
  }

  public void setZmiennawiazana(String newZmiennawiazana) {
    String  oldZmiennawiazana = zmiennawiazana;
    this.zmiennawiazana = newZmiennawiazana;
    propertyChangeListeners.firePropertyChange("zmiennawiazana", oldZmiennawiazana,
newZmiennawiazana);
  }

  public String getZmiennawiazana() {
    return zmiennawiazana;
  }

  public synchronized void removePropertyChangeListener(PropertyChangeListener l) {
    super.removePropertyChangeListener(l);
    propertyChangeListeners.removePropertyChangeListener(l);
  }

  public synchronized void addPropertyChangeListener(PropertyChangeListener l) {
    super.addPropertyChangeListener(l);
    propertyChangeListeners.addPropertyChangeListener(l);
  }

  private void writeObject(ObjectOutputStream oos) throws IOException {
    oos.defaultWriteObject();
  }

  private void readObject(ObjectInputStream ois) throws ClassNotFoundException, IOException
{
    ois.defaultReadObject();
  }
}
```

Przyklad 2 - Ziarno 2

```
import java.awt.*;
import java.beans.*;

public class wiazane2 extends Label implements PropertyChangeListener{

  public wiazane2() { }

  private void jbInit() { }

  public void propertyChange (PropertyChangeEvent evt) {
    this.setText(evt.getNewValue().toString());
  }
```

```
  private void writeObject(ObjectOutputStream oos) throws IOException {
    oos.defaultWriteObject();
  }

  private void readObject(ObjectInputStream ois)
        throws ClassNotFoundException, IOException {
    ois.defaultReadObject();
  }
}
```

Przyklad 2 - Program

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.lang.String;
import java.awt.Color;

public class wiazanep {
 public static void main(String[] args) {
  Okno tProgram = new Okno("Właściwości Wiązane");
  tProgram.setSize(300,100);
  tProgram.setBackground(java.awt.Color.lightGray);
  tProgram.show();
 }
}

class Okno extends Frame {
 Button przycisk = new Button();
 wiazane1 bean1 = new wiazane1();
 wiazane2 bean2 = new wiazane2();

 public Okno(String nazwa) {
  super(nazwa);
  przycisk.setLabel("Pobierz Text");
  bean1.setText("Tu wpisz tekst");
  bean2.setText("Tu pojawi sie tekst");
  setLayout(new BorderLayout());
  add(bean1, BorderLayout.NORTH);
  add(przycisk, BorderLayout.CENTER);
  add(bean2, BorderLayout.SOUTH);
  bean1.addPropertyChangeListener(bean2);

  przycisk.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    bean1.setZmiennawiazana(bean1.getText());
   }
  });

  addWindowListener(new ProgramWindowAdapter());
 }
}
class ProgramWindowAdapter extends WindowAdapter {
 public void windowClosing(WindowEvent e) {
  System.exit(0);
```

```
 }
}
```

Właściwości Ograniczane - Ziarno 1

```
import java.awt.*;
import java.beans.*;
import java.lang.String;

public class Ogranicz1 extends TextField {

  public Ogranicz1() { }

  private void jbInit() { }

  public void setNapis(String newNapis) throws PropertyVetoException {
    String  oldNapis = napis;
    veto.fireVetoableChange("napis", oldNapis, newNapis);
    this.napis = newNapis;
    zmiany.firePropertyChange("napis", oldNapis, newNapis);
  }

  public String getNapis() {
    return napis;
  }

  public void removePropertyChangeListener(PropertyChangeListener l) {
    zmiany.removePropertyChangeListener(l);
  }

  public void addPropertyChangeListener(PropertyChangeListener l) {
    zmiany.addPropertyChangeListener(l);
  }

  public void removeVetoableChangeListener(VetoableChangeListener l) {
    veto.removeVetoableChangeListener(l);
  }

  public void addVetoableChangeListener(VetoableChangeListener l) {
    veto.addVetoableChangeListener(l);
  }

  private String napis;
  private PropertyChangeSupport zmiany = new PropertyChangeSupport(this);
  private VetoableChangeSupport veto = new VetoableChangeSupport(this);
}
```

Właściwości ograniczane - Ziarno 2

```
import java.awt.*;
import java.beans.*;
import java.lang.String;

public class Ogranicz2 extends Checkbox implements VetoableChangeListener {
  String a = new String("a");
  String b = new String("");
```

```
  public Ogranicz2() { }

  private void jbInit() { }

  public void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException {

    if (evt.getNewValue().toString() != a)
      throw new PropertyVetoException("",evt);
    else
      b = evt.getNewValue().toString();
  }
}
```

Przyklad 4 – Ziarno

```
package ziarna;

import java.awt.*;
import java.io.*;
import java.beans.*;
import java.lang.String;

public class indeks extends Label implements Serializable {
  String tablica[] = new String[10];

  public indeks() { }

  private void jbInit() { }

  private void writeObject(ObjectOutputStream oos) throws IOException {
    oos.defaultWriteObject();
  }

  private void readObject(ObjectInputStream ois)
      throws ClassNotFoundException, IOException {
    ois.defaultReadObject();
  }

  public void setItem(int index, String tekst) {
    tablica[index]=tekst;
  }

  public String getItem(int index) {
    return tablica [index];
  }
  public String[] getTablica() {
    return tablica;
  }
  public void setTablica(String[] tablica) {
    this.tablica = tablica;
  }
}
```

Przyklad 4 – Program

```
import java.awt.*;
```

```
import java.awt.event.*;
import javax.swing.*;
import java.lang.Integer;

public class indeksp {
 public static void main(String[] args) {
  Okno tProgram = new Okno("WŚa□ciwo□ci Indeksowe");
  tProgram.setSize(300,150);
  tProgram.setBackground(java.awt.Color.lightGray);
  tProgram.show();
 }
}

class Okno extends Frame {
 private Button get = new Button("Get");
 private Button set = new Button("Set");
 TextField textField1 = new TextField();
 indeks bean = new indeks();
 TextField textField2 = new TextField();

 public Okno(String nazwa) {
  super(nazwa);
  setLayout(new BorderLayout());
  textField1.setText("Tu wpisz numer elementu tablicy");
  bean.setText("Wyniki");
  textField2.setText("Tu wpisz Tekst");
  add(textField2,BorderLayout.NORTH);
  add(bean, BorderLayout.CENTER);
  add(textField1, BorderLayout.SOUTH);
  add(get, BorderLayout.EAST);
  add(set, BorderLayout.WEST);

  get.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    int index;
    index = java.lang.Integer.parseInt(textField1.getText());
    bean.setText(bean.getItem(index));
   }
  });
  set.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    int index;
    index = java.lang.Integer.parseInt(textField1.getText());
    bean.setItem(index, textField2.getText());
   }
  });

  addWindowListener(new ProgramWindowAdapter());
 }
}
class ProgramWindowAdapter extends WindowAdapter {
 public void windowClosing(WindowEvent e) {
  System.exit(0);
 }
}
```

Przykład klasy BeanInfo dla ziarna wykorzystanego w przykładzie 1.

```
import java.beans.*;

public class prosteBeanInfo extends SimpleBeanInfo {
  Class beanClass = proste.class;
  String iconColor16x16Filename;
  String iconColor32x32Filename;
  String iconMono16x16Filename;
  String iconMono32x32Filename;

  public prosteBeanInfo() { }
  public PropertyDescriptor[] getPropertyDescriptors() {
    try {
      PropertyDescriptor _a = new PropertyDescriptor("a", beanClass, null, null);
      PropertyDescriptor _b = new PropertyDescriptor("b", beanClass, null, null);
      PropertyDescriptor _borderLayout1 = new PropertyDescriptor("borderLayout1", beanClass,
null, null);
      PropertyDescriptor _kolor = new PropertyDescriptor("kolor", beanClass, "getKolor",
"setKolor");
      PropertyDescriptor[] pds = new PropertyDescriptor[] { _a, _b, _borderLayout1, _kolor};
      return pds;
    }
    catch(IntrospectionException ex) {
      ex.printStackTrace();
      return null;
    }
  }
  public java.awt.Image getIcon(int iconKind) {
    switch (iconKind) {
    case BeanInfo.ICON_COLOR_16x16:
        return iconColor16x16Filename != null ? loadImage(iconColor16x16Filename) : null;
    case BeanInfo.ICON_COLOR_32x32:
        return iconColor32x32Filename != null ? loadImage(iconColor32x32Filename) : null;
    case BeanInfo.ICON_MONO_16x16:
        return iconMono16x16Filename != null ? loadImage(iconMono16x16Filename) : null;
    case BeanInfo.ICON_MONO_32x32:
        return iconMono32x32Filename != null ? loadImage(iconMono32x32Filename) : null;
        }
    return null;
  }
}
```

5 Przykład

Nasz Bean MsgCanvas będzie dziedziczył z klasy java.awt.Canvas i będzie umieszczał na środku napis. Napis będzie jedną ze dodatkowych właściwości bean'a. Będzie również możliwe wysyłanie zdarzenia.

5.1 MsgCanvas

```
// MsgCanvas.java

import java.awt.*;
```

```
import java.util.Vector;
import java.util.*;

public class MsgCanvas extends Canvas {

    private String  msg;
    private int     width, height;

    public MsgCanvas() {
        this("Message");
    }

    public MsgCanvas(String s) {
        this(s, 200, 200);
    }

    public MsgCanvas(String s, int width, int height) {
        super();
        this.width = width;
        this.height = height;
        this.msg = s;
        setForeground(SystemColor.controlText);
        setFont(new Font("Serif", Font.ITALIC, 24));
        setSize(getPreferredSize());
    }

    public void paint(Graphics g) {
        Dimension d = getSize();
        FontMetrics fm = g.getFontMetrics();
        int len = fm.stringWidth(msg);
        int x = Math.max(((d.width - len) / 2), 0);
        int y = d.height / 2;
        g.drawString(msg, x, y);
    }

    public Dimension getPreferredSize() {
        return new Dimension(width, height);
    }

    private int     foo = 0;
    public int      getFoo()        { return foo; }
    public void     setFoo(int foo) { this.foo = foo; }

    public String   getMessage()            { return msg; }
    public void     setMessage(String msg)  { this.msg = msg; }


    private Vector<FooListener> fooListeners    = new Vector<FooListener>();

    public synchronized void addFooListener(FooListener l) {
        fooListeners.addElement(l);
    }

    public synchronized void removeFooListener(FooListener l) {
        fooListeners.removeElement(l);
```

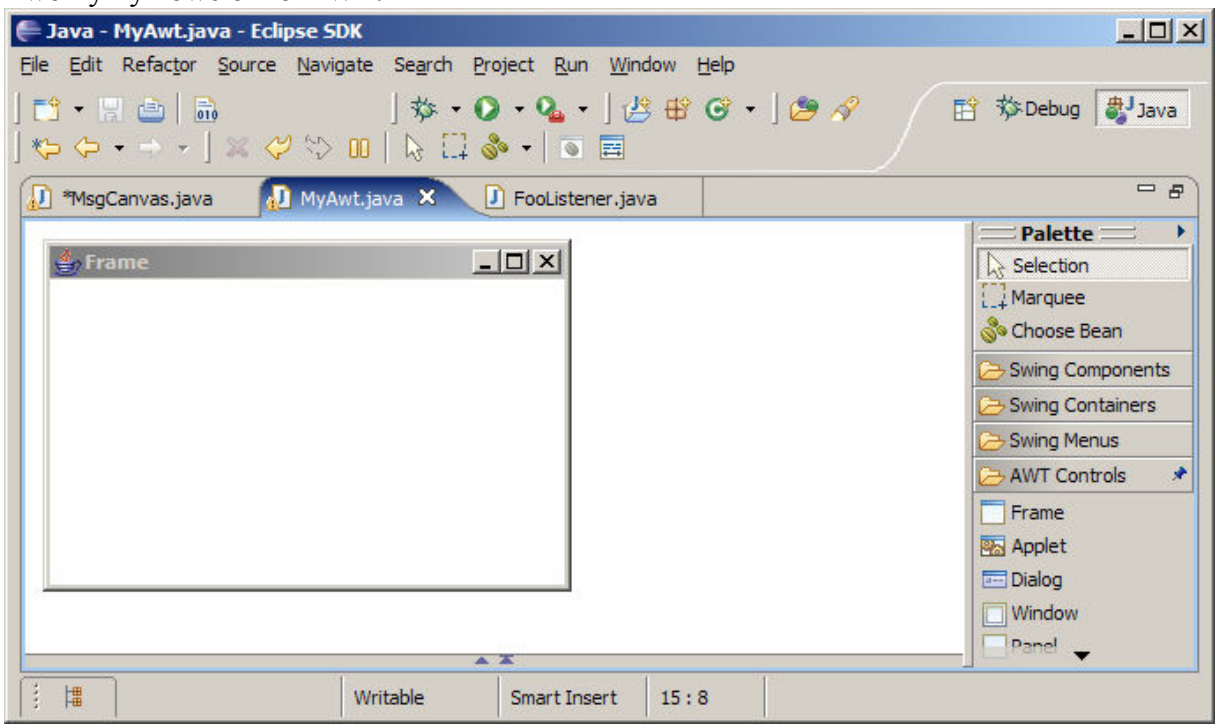
```
    }

    protected void notifyFoos() {
        Vector<EventListener> l;
        EventObject e = new EventObject(this);
        synchronized (this) {
            l = (Vector<EventListener>) fooListeners.clone();
        }
        for (int i = 0; i < l.size(); i++) {
            FooListener fl = (FooListener) l.elementAt(i);
            fl.foo(e);
        }
    }
}
```

5.2 Interejs FooListener

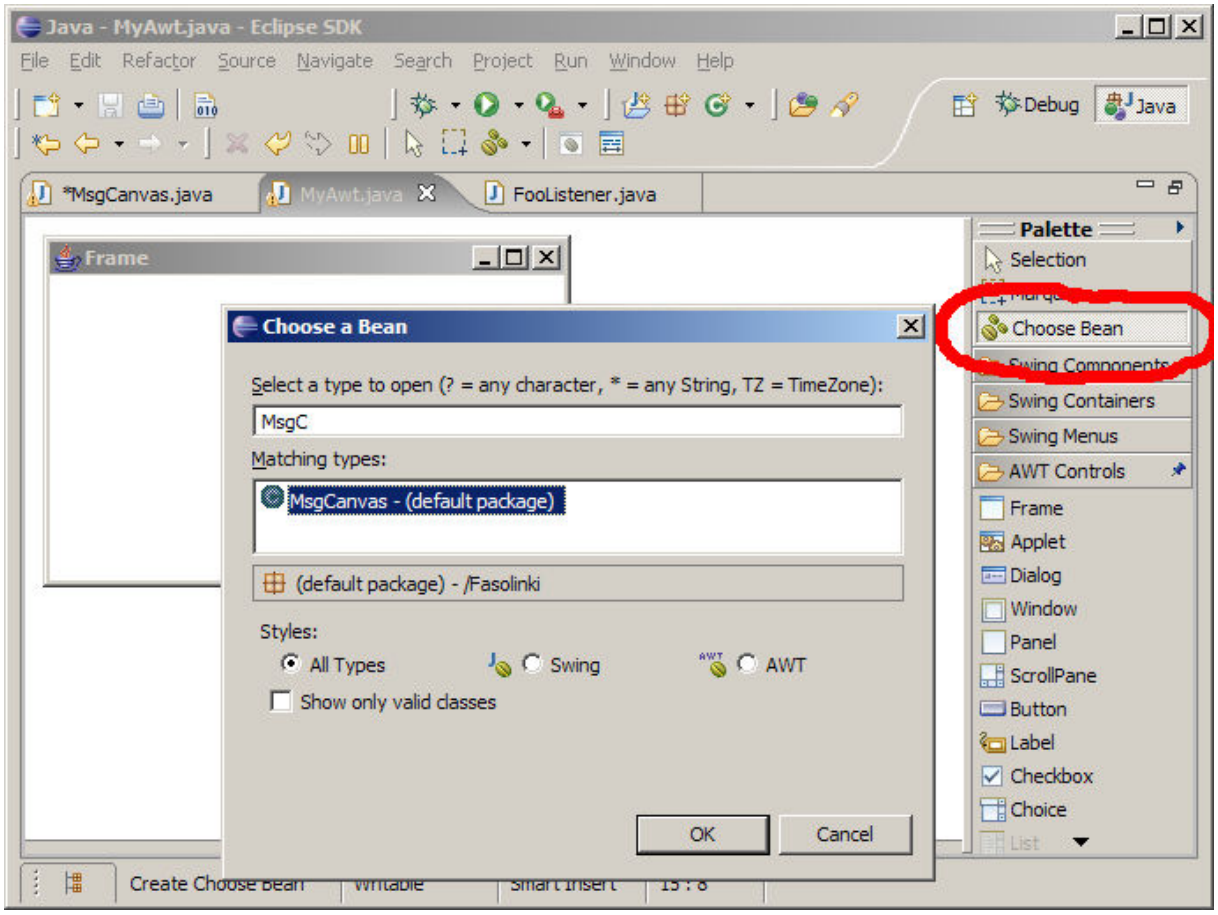
```
// FooListener.java
import java.util.EventListener;
import java.util.EventObject;

public interface FooListener extends EventListener {
    public void foo(EventObject e);
}
```

5.3 Aplikacja

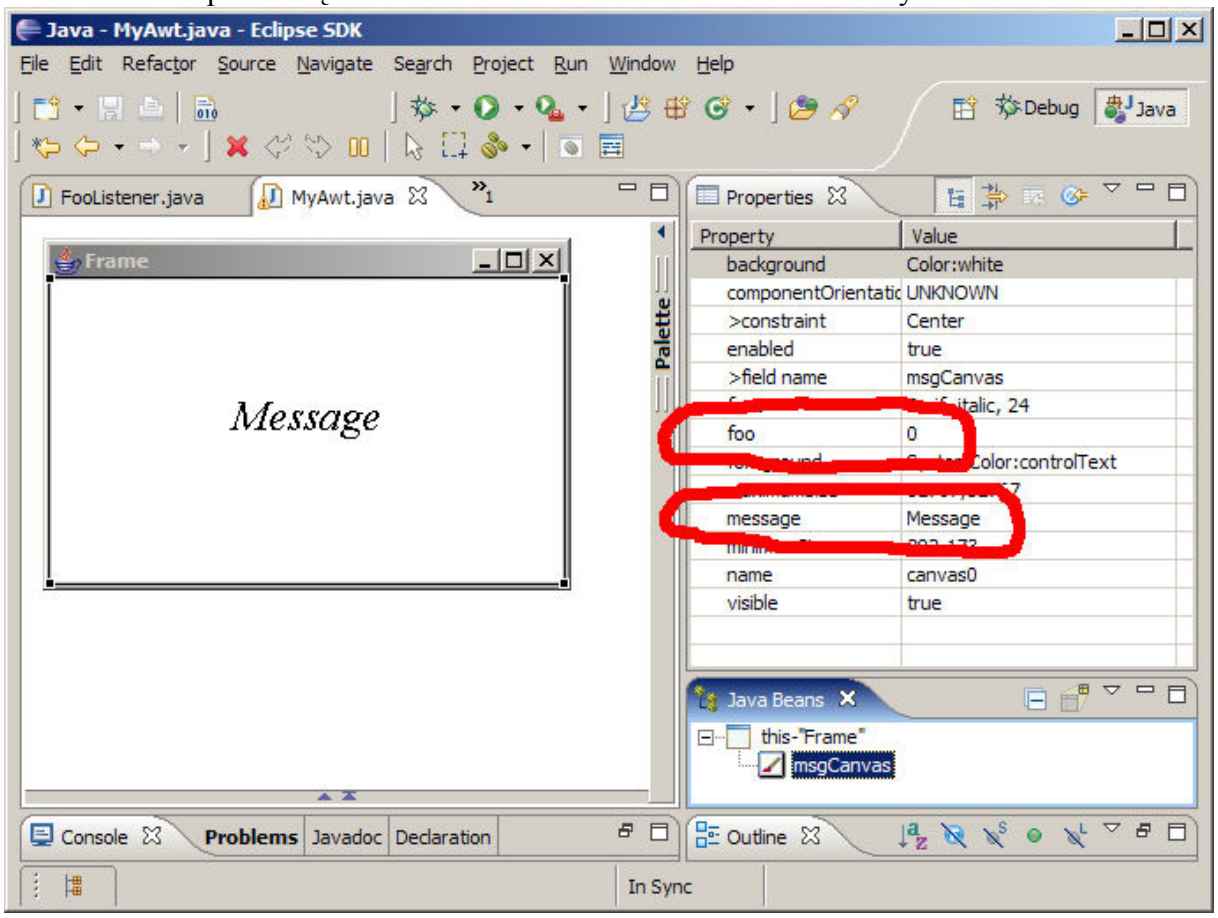
Tworzymy nowe okno AWT.



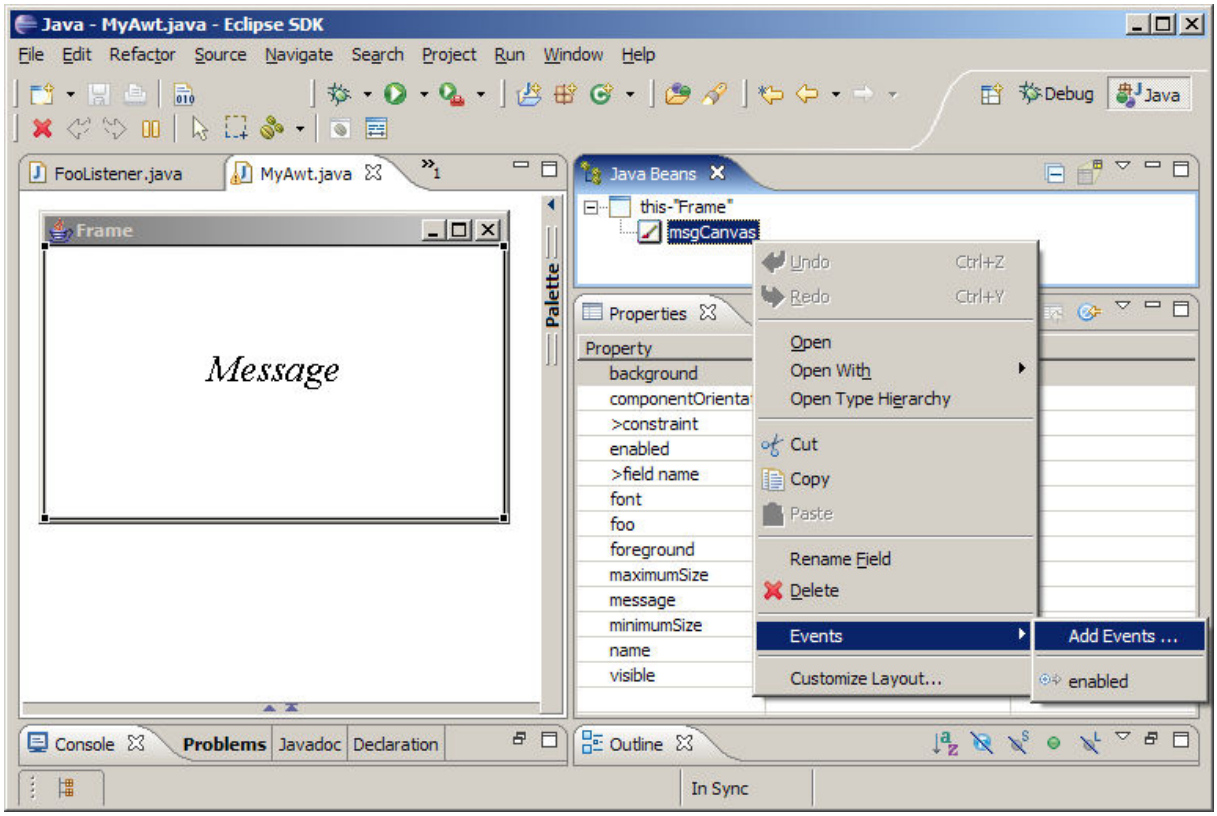
Dodajemy nasz Bean:

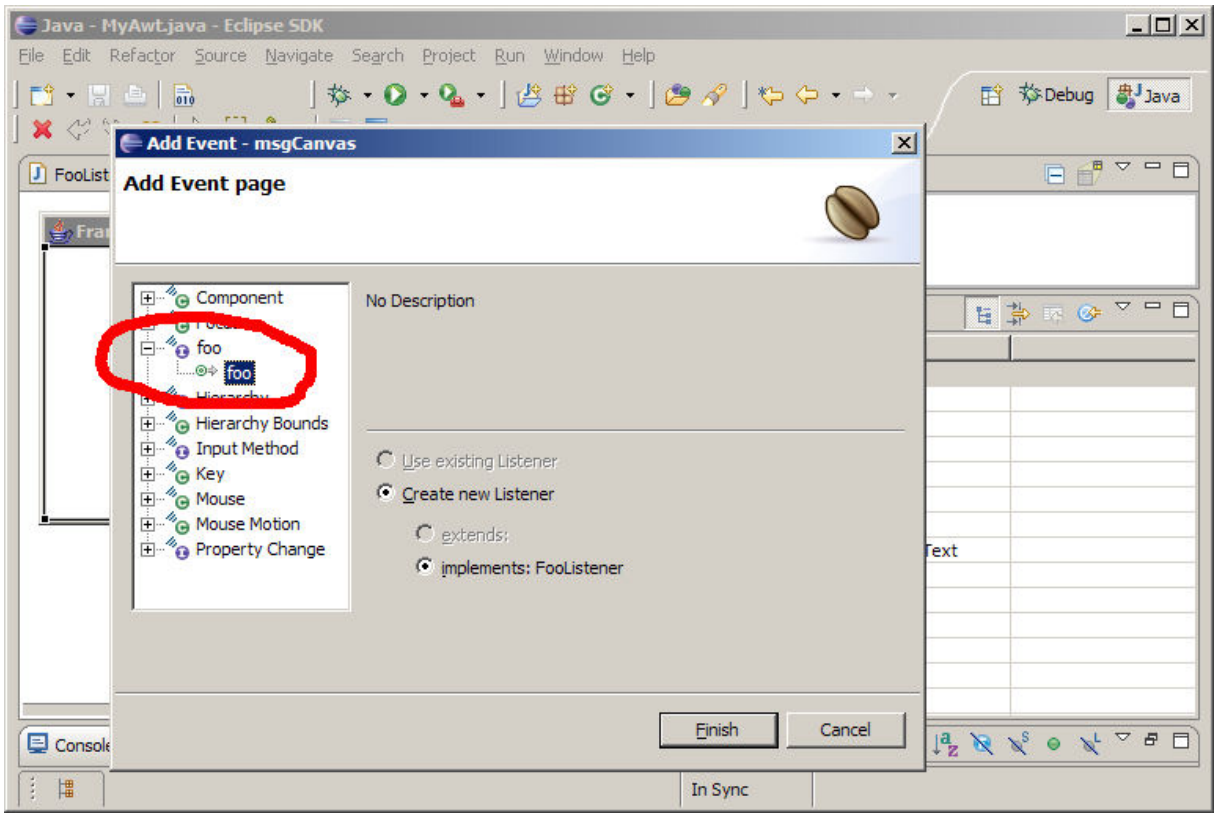


W zakładce Properties są widoczne właściwości zdefiniowane w naszym Bean'ie:



Można też dodać zdefiniowany Event:


Java - MyAwt.java - Eclipse SDK
File Edit Refactor Source Navigate Search Project Run Window Help
Debug
Java
FooListener.java
MyAwt.java
Frame
Message
Palette
Java Beans
this-"Frame"
msgCanvas
Properties
Property
background
componentOrienta
>constraint
enabled
>field name
font
foo
foreground
maximumSize
message
minimumSize
name
visible
Undo Ctrl+Z
Redo Ctrl+Y
Open
Open With
Open Type Hierarchy
Cut
Copy
Paste
Rename Field
Delete
Events
Customize Layout...
Add Events ...
enabled
Console
Problems
Javadoc
Declaration
Outline
In Sync


Java - MyAwt.java - Eclipse SDK
File Edit Refactor Source Navigate Search Project Run Window Help
Debug
Java
Add Event - msgCanvas
Add Event page
Component
foo
foo
Hierarchy Bounds
Input Method
Key
Mouse
Mouse Motion
Property Change
No Description
Use existing Listener
Create new Listener
extends:
implements: FooListener
Finish
Cancel
In Sync