ИНТУИТ
НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ

Национальный Открытый Университет "ИНТУИТ": www.intuit.ru
Вадим Монахов
Лекция 12. Компонентное программирование

12.1. Компонентная архитектура JavaBeans

Компонент – это:

- автономный элемент программного обеспечения, предназначенный для многократного использования, который может распространяться для использования в других программах в виде скомпилированного кода класса;
- подключение к этим программам осуществляется с помощью интерфейсов;
- взаимодействие с программной средой осуществляется по событиям, причем в программе, использующей компонент, можно назначать обработчики событий, на которые умеет реагировать компонент.

Технология *JavaBeans* предоставляет возможность написания компонентного программного обеспечения на языке *Java*. *Beans* по-английски означает "зерна" – обыгрывается происхождение названия "*Java*" от любимого создателями языка *Java* сорта кофе. Компоненты *JavaBeans* в литературе по языку *Java* часто упоминаются просто как *Beans* .

Компонент *JavaBeans* может быть включен в состав более сложных (составных) компонентов, приложений, *сервлетов*, пакетов, модулей. Причем обычно это делается с помощью сред визуального проектирования.

Компоненты *JavaBeans* предоставляют свои общедоступные методы и события для режима визуального проектирования. *Доступ* к ним возможен в том случае, когда их названия соответствуют особым *шаблонам проектирования* (*bean design patterns*). Для задания свойства требуется, чтобы существовали геттер и сеттер для этого свойства. Пример будет приведен в следующем параграфе.

Компонент может быть установлен в среду разработки, в этом случае кнопки доступа к компонентам выносятся на палитру (*palette*) или *панель инструментов* (*toolbox*). Вы можете создать экземпляр компонента на проектируемой экранной форме в режиме Design ("*дизайн*") путем выбора его кнопки на панели и перетаскивания на форму. Затем можно изменять его свойства , писать обработчики событий, включать в состав других компонентов и т. д.

Компонент *JavaBeans* является классом *Java* и имеет три типа атрибутов:

- **Методы** компонента *JavaBeans* не отличаются от других методов объектов в Java. Они описывают поведение компонента. Общедоступные методы компонента могут вызываться из других компонентов или из обработчиков событий.
- **Свойства** (Properties) компонента *JavaBeans* характеризуют его внешний вид и поведение и могут быть изменены в процессе визуального проектирования. Это можно сделать с помощью редактора свойств (Property Editor), а некоторые из свойств – вручную (положение компонента, его размер, текст). Свойство задается комбинацией геттера и сеттера (метода по чтению и метода по записи).
- **События** (Events) используются для связи между компонентами. При помещении компонента на *экранную форму* среда разработки исследует компоненты и определяет, какие программные события данный компонент может порождать (рассылать) и какие – получать (обрабатывать).

При окончании работы со средой разработки состояние компонентов сохраняется в файле с помощью механизма *сериализации* – представления объектов *Java* в виде потока байтов. При последующей загрузке проекта сохраненное состояние компонентов считывается из файла.

В NetBeans существует несколько способов создания компонента *JavaBeans* .

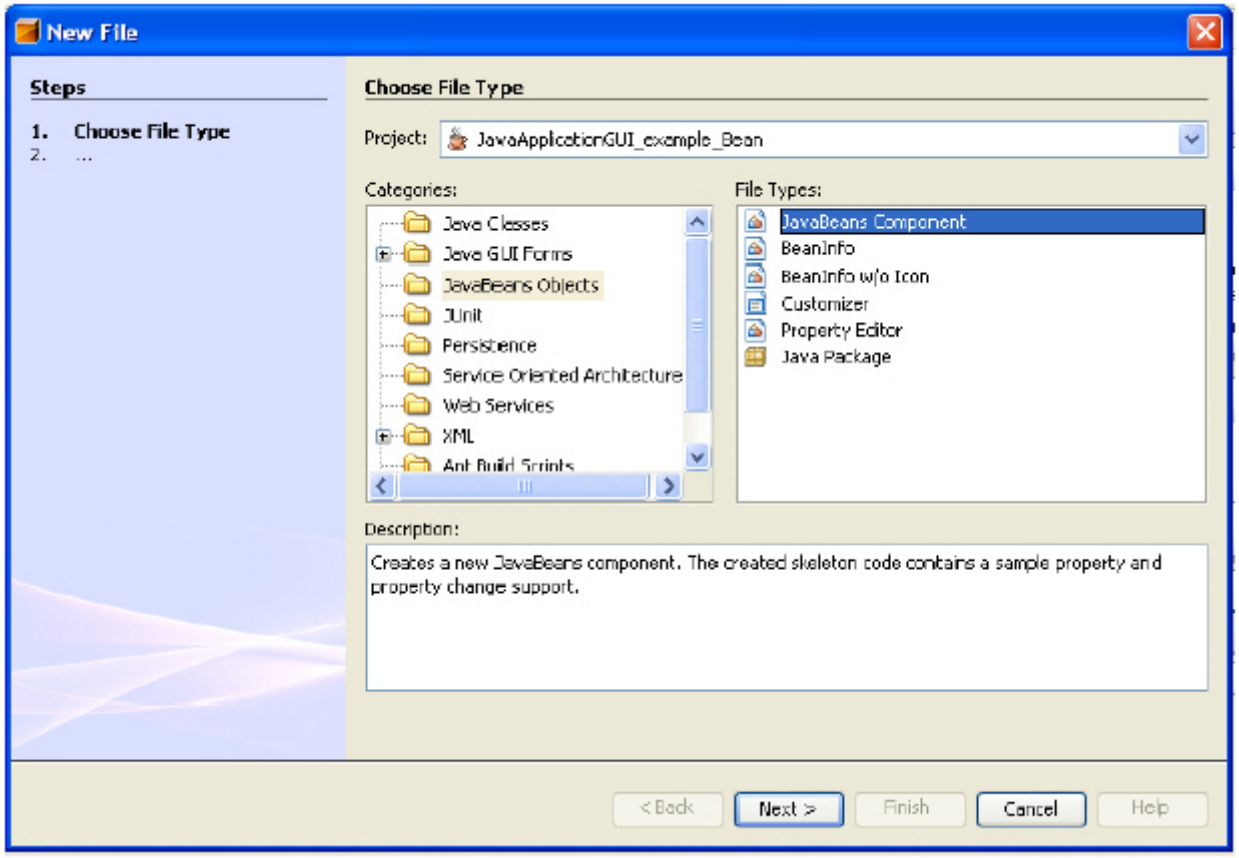
Наиболее простым является использование мастера создания компонента. О нем будет сказано в следующем параграфе.

Другой способ – создать для компонента *класс* BeanInfo , обеспечивающий поставку необходимой информации о компоненте. Этот *класс* должен реализовывать *интерфейс* BeanInfo , и его имя должно состоять из имени компонента и названия интерфейса. Например, для компонента MyComponent это будет MyComponentBeanInfo . Существует *класс-адаптер* SimpleBeanInfo для интерфейса BeanInfo . В его наследнике достаточно переписать методы, подлежащие модификации. Среда NetBeans позволяет с помощью мастера создать заготовку такого класса.

Для редактирования некоторых свойств в среде визуального проектирования требуется специальный *объект* – редактор свойств (*Property Editor*). Среда NetBeans позволяет с помощью мастера создать заготовку и для такого класса.

12.2. Мастер создания компонента в NetBeans

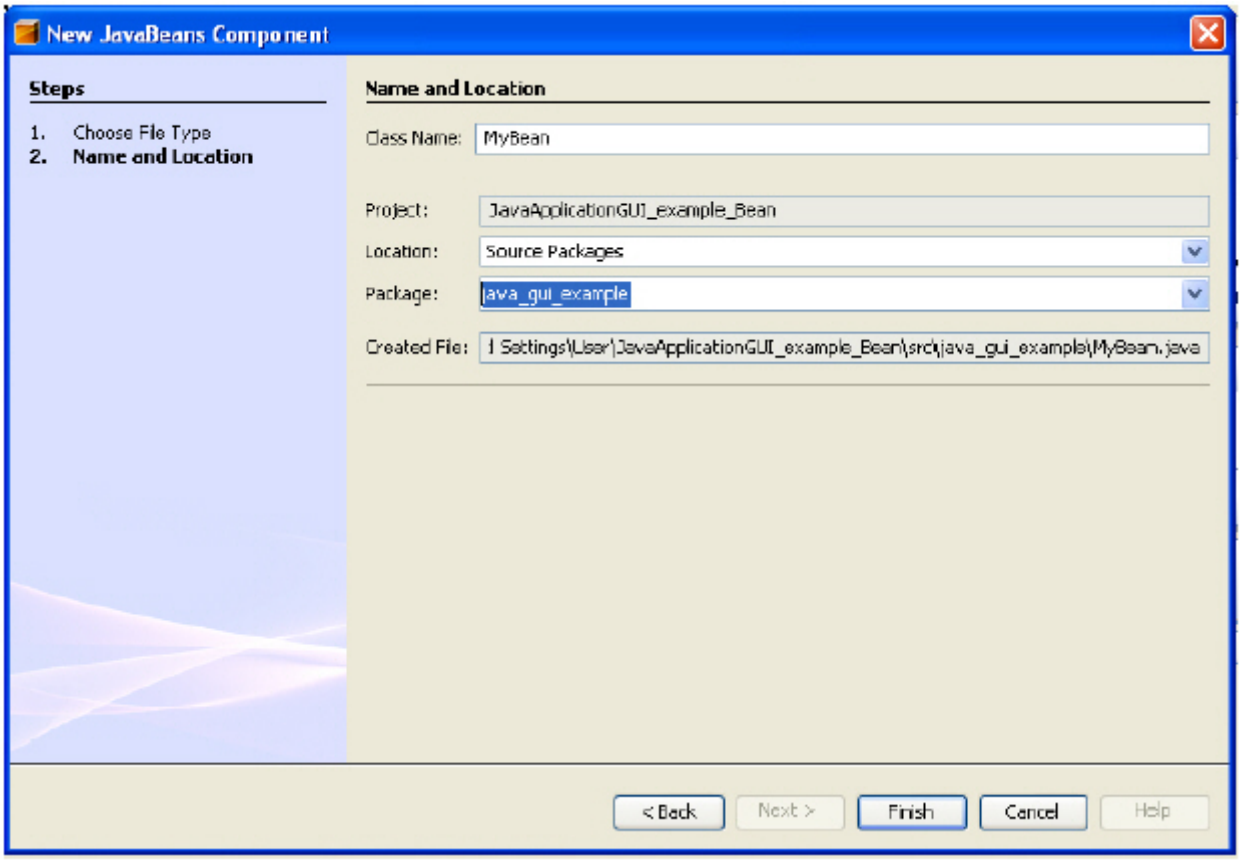
Рассмотрим подробнее процесс создания собственного компонента. В NetBeans для этого необходимо выбрать в *меню* **File/New File…/JavaBeans Objects/JavaBeans Component** и нажать кнопку **Next>**.



увеличить изображение

Рис. 12.1. Создание компонента JavaBeans. Шаг 1

Далее в *поле* **Class Name** надо ввести имя компонента. В качестве примера мы введем MyBean. Затем обязательно следует выбрать пакет, в котором мы будем создавать *компонент* – мы выберем пакет нашего приложения. После чего следует нажать на кнопку **Finish**.



увеличить изображение

Рис. 12.2. Создание компонента JavaBean. Шаг 2

Приведем код получившейся заготовки:

```
/*
 * MyBean.java
 *
 * Created on 30 Октябрь 2006 г., 23:16
 */

package java_gui_example;

import java.beans.*;
import java.io.Serializable;

/**
 * @author В.Монахов
 */
public class MyBean extends Object implements Serializable {

    public static final String PROP_SAMPLE_PROPERTY = "sampleProperty";

    private String sampleProperty;

    private PropertyChangeSupport propertySupport;

    public MyBean() {
        propertySupport = new PropertyChangeSupport(this);
    }

    public String getSampleProperty() {
        return sampleProperty;
    }

    public void setSampleProperty(String value) {
        String oldValue = sampleProperty;
        sampleProperty = value;
        propertySupport.firePropertyChange(PROP_SAMPLE_PROPERTY,
                                           oldValue, sampleProperty);
    }

    public void addPropertyChangeListener(PropertyChangeListener listener) {
        propertySupport.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener
                                                                listener) {
        propertySupport.removePropertyChangeListener(listener);
    }
```

```
}
```

В данном компоненте создана заготовка для строкового свойства `sampleProperty`. Геттер `public String getSampleProperty()` обеспечивает чтение значения свойства, а сеттер `public void setSampleProperty(String value)` обеспечивает установку нового значения.

Служебный *объект* `private PropertyChangeSupport propertySupport` обеспечивает поддержку работы с обработчиком события `PropertyChange`. Отметим, что `"property change"` означает "изменение свойства". Это событие должно возникать при каждом изменении свойств нашего компонента.

Как уже говорилось, в каждом объекте, поддерживающем работу с неким событием (в нашем случае это событие `PropertyChange`), имеется *список* объектов-слушателей событий (*listeners*). Иногда их называют зарегистрированными слушателями. Методы с названием fire *ИмяСобытия* ("fire" – "стрелять", в данном случае – "выстрелить событием") осуществляют поочередный вызов зарегистрированных слушателей из списка для данного события, передавая им событие на обработку. В нашем случае это метод `propertySupport.firePropertyChange`. Сначала он обеспечивает создание объекта-события, если *значение* свойства действительно изменилось, а потом поочередно вызывает слушателей этого события для его обработки.

Методы

```
public void addPropertyChangeListener(PropertyChangeListener listener)
```

и

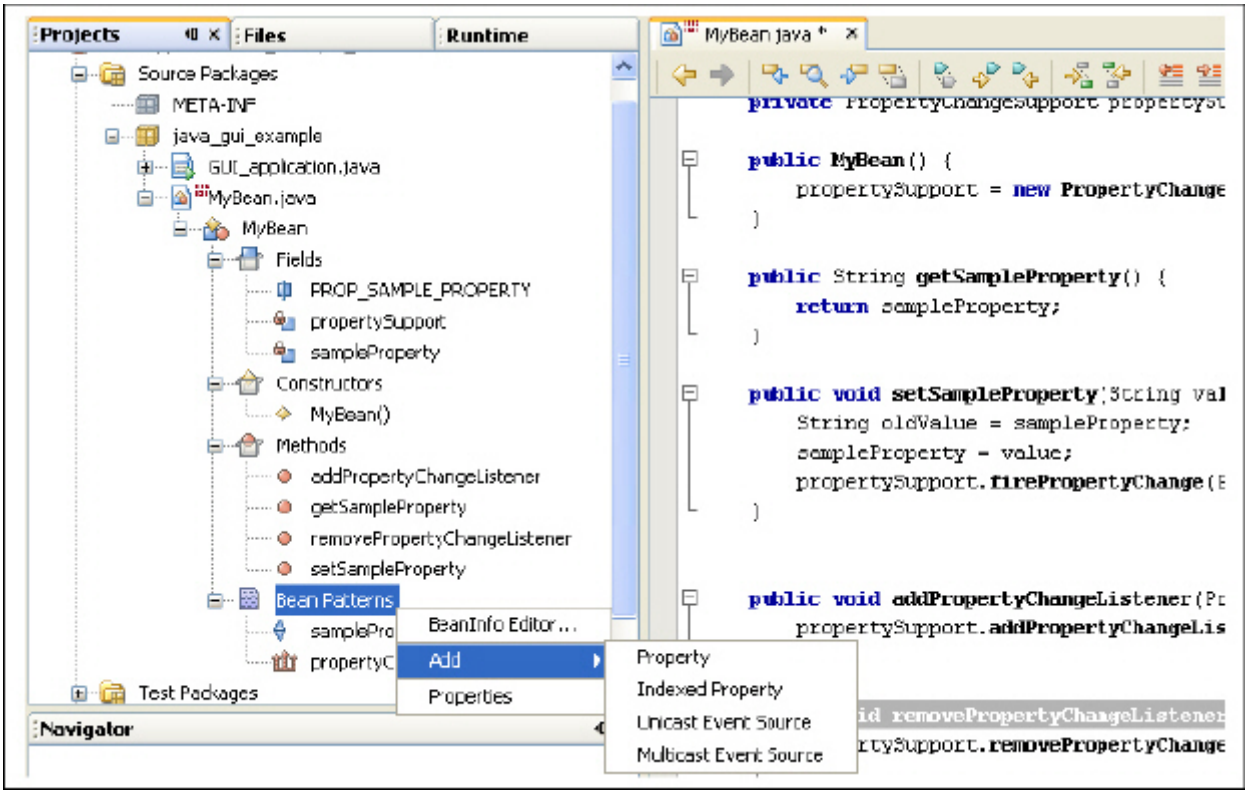
```
public void removePropertyChangeListener(PropertyChangeListener listener)
```

обеспечивают для компонента возможность добавления и удаления объекта слушателя – обработчика события `Property Change`.

Если требуется создать другие свойства или обеспечить добавление и удаление обработчиков других событий, можно воспользоваться соответствующим мастером. В узле **Bean Patterns** ("*Pattern*" означает "образец") следует правой кнопкой мыши вызвать всплывающее *меню*, и выбрать **Add**. А затем в зависимости от того, что необходимо, выбрать один из видов свойств (**Property**) или событий (**Event**). Об этом более подробно будет говориться далее.

Таким же образом удаляются свойства и события компонента.

12.3. Пример создания компонента в NetBeans – панель с заголовком



увеличить изображение

Рис. 12.3. Задание новых свойств и событий

В качестве простейшего примера визуального компонента создаем панель, у которой имеется заголовок (title). Унаследуем наш *компонент* от класса javax.*swing*.JPanel – для этого в импорте запишем

```
import javax.swing.*;
```

а в качестве *родительского класса* вместо Object напишем JPanel.

С помощью рефакторинга заменим имя myBean на JTitledPanel , в узле полей Fields (а не в *Bean Patterns* !) *поле* sampleProperty на title , а константу PROP_SAMPLE_PROPERTY уберем, написав в явном виде имя свойства "title" в методе firePropertyChange .

После чего в области **Bean Patterns** правой клавишей мыши вызовем всплывающее *меню*, и там вызовем *пункт* **Rename…** ("Переименовать") для свойства sampleProperty – заменим имя на title . Это приведет к тому, что методы getSampleProperty и setSampleProperty будут переименованы в getTitle и setTitle .

Обязательно следует присвоить начальное *значение* полю title – в заготовке, полученной из *Bean Pattern* , это не делается. Мы установим

```
private String title="Заголовок";
```

Для показа заголовка необходимо импортировать классы java.*awt*.Graphics , java.*awt*.geom.Rectangle2D и переопределить в JTitledPanel.java метод paint :

```
public void paint(Graphics g){
    super.paint(g);
    FontMetrics fontMetrics=g.getFontMetrics();
    Rectangle2D rect = fontMetrics.getStringBounds(title, g);
    g.drawString(title,(int)Math.round((this.getWidth()-rect.getWidth())/2),
                10);
    }
```

Для того, чтобы можно было пользоваться классами Graphics , FontMetrics и Rectangle2D , нам следует добавить импорт

```
import java.awt.*;
import java.awt.geom.Rectangle2D;
```

Отметим, что можно было бы не вводить переменные fontMetrics и rect , а сразу писать в методе drawString соответствующие функции в следующем виде:

```
g.drawString(title,
      (int)Math.round( ( this.getWidth() -
      g.getFontMetrics().getStringBounds(title,g).getWidth()
      )/2 ),
      10);
```

Но от этого текст программы стал бы гораздо менее читаемым. Даже несмотря на попытки отформатировать текст так, чтобы было хоть что-то понятно.

Еще одно необходимое изменение – добавление repaint() в операторе setTitle . Если этого не сделать, после изменения свойства *компонент* не перерисуется с вновь установленным заголовком.

В результате получим следующий код компонента:

```
/*
 * JTitledPanel.java
 *
 * Created on 30 Октябрь 2006 г., 23:16
 */

package java_gui_example;

import java.beans.*;
```

```
import java.io.Serializable;
import javax.swing.*; //добавлено вручную
import java.awt.*; //добавлено вручную
import java.awt.geom.Rectangle2D; //добавлено вручную

/**
 * @author В.Монахов
 */
public class JTitledPanel extends JPanel implements Serializable {

   private String title="Заголовок"; //добавлено вручную

   private PropertyChangeSupport propertySupport;

   public JTitledPanel() {
       super();
       propertySupport = new PropertyChangeSupport(this);
   }

   public String getTitle() {
       return title;
   }

   public void setTitle(String value) {
       String oldValue = title;
       title = value;
       propertySupport.firePropertyChange("title", oldValue, title);
       repaint(); //добавлено вручную
   }

   public void addPropertyChangeListener(PropertyChangeListener listener) {
       propertySupport.addPropertyChangeListener(listener);
   }

   public void removePropertyChangeListener(PropertyChangeListener listener){
       propertySupport.removePropertyChangeListener(listener);
   }

   public void paint(Graphics g){ //метод добавлен вручную
     super.paint(g);
     FontMetrics fontMetrics=g.getFontMetrics();
     Rectangle2D rect = fontMetrics.getStringBounds(title, g);
     g.drawString(title,(int)Math.round((this.getWidth() -
                                          rect.getWidth())/2), 10);
   }
}
```

Для того, чтобы добавить наш *компонент* в палитру, следует открыть *файл* **JTitledPanel.java** в окне редактора исходного кода, и в *меню* **Tools** выбрать *пункт* **Add to Palette.** После чего в появившемся диалоге выбрать палитру, на которую будет добавлен *компонент*.

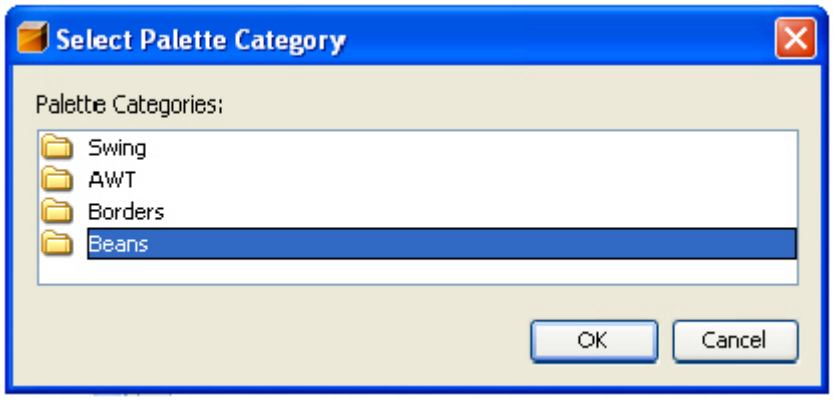


Рис. 12.4. Выбор палитры, на которую будет добавлен компонент

Желательно выбрать **Beans** (чтобы не путать наши компоненты со стандартными) и нажать **OK**. Теперь *компонент* можно использовать наравне с другими.

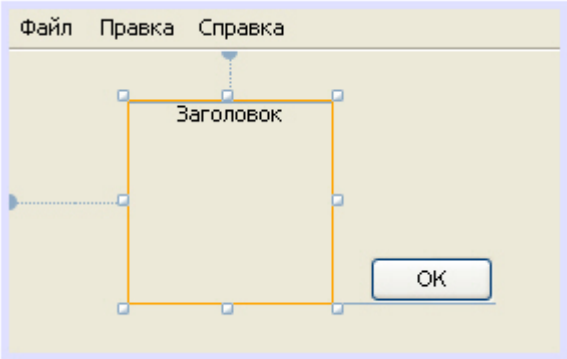


Рис. 12.5. Использование созданного компонента

Теперь мы можем менять текст заголовка как в редакторе свойств на этапе визуального проектирования, так и программно во время работы приложения. Мы также можем рисовать по нашей панели, и заголовок при этом будет виден, как и отрисовываемые примитивы. Например, мы можем вывести по нажатию на какую-нибудь кнопку строку **"Тест"**:

```
Graphics g=jTitledPanel1.getGraphics();
FontMetrics fontMetrics=g.getFontMetrics();
Rectangle2D rect = fontMetrics.getStringBounds("Тест", g);
g.drawString("Тест",10,30 );
```

Если мы будем усовершенствовать код нашего компонента, нет необходимости каждый раз удалять его из *палитры компонентов* и заново устанавливать – достаточно после внесения изменений заново скомпилировать проект (**Build main project – F11**).

12.4. Добавление в компонент новых свойств

В *компонент* можно добавить новое свойство. Пусть мы хотим задать свойство `titleShift` типа `int` – оно будет задавать высоту нашего заголовка., выбираем в окне **Projects…** для соответствующего компонента узел **Bean Patterns** и щелкаем по ним правой клавишей мыши. В появившемся всплывающем *меню* выбираем **Add/Property**, после чего в появившемся диалоге вводим имя и тип свойства.

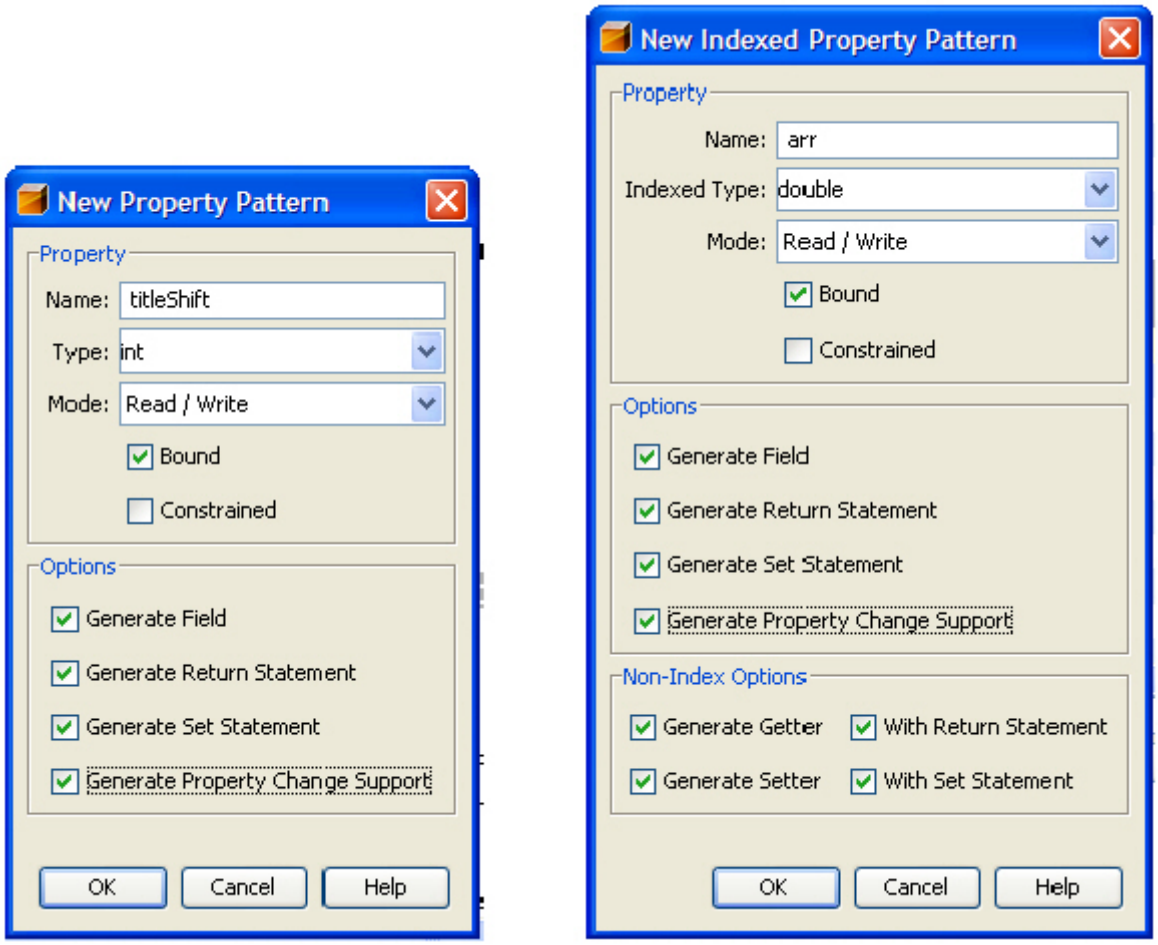


Рис. 12.6. Добавление в компонент свойства, слева – обычного, справа – массива

Пункты **Bound** ("связанное свойство") и **Constrained** ("стесненное свойство") позволяют использовать опцию **"Generate Property Change Support"** – без ее выбора они ни на что не влияют.

Свойства вида **Bound** – обычные свойства. При отмеченных опциях **"Bound"** и **"Generate Property Change Support"** автоматически добавляется код, генерирующий в компоненте событие `PropertyChange` при изменении свойства компонента. Именно таким образом была ранее создан средой NetBeans код для работы с событием `PropertyChange` .

Например, если мы добавим целочисленное свойство `titleShift` ("*shift*"- сдвиг) вида *Bound* , задающее сдвиг заголовка по вертикали, в исходный код компонента добавится следующий текст:

```
/**
 * Holds value of property titleShift.
 */
private int titleShift;

/**
 * Getter for property titleShift.
 * @return Value of property titleShift.
 */
public int getTitleShift() {
    return this.titleShift;
}

/**
 * Setter for property titleShift.
 * @param titleShift New value of property titleShift.
 */
public void setTitleShift(int titleShift) {
```

```
    int oldTitleShift = this.titleShift;
    this.titleShift = titleShift;
    propertySupport.firePropertyChange ("titleShift",
                    new Integer (oldTitleShift),
                    new Integer (titleShift));
    repaint();//добавлено вручную
}
```

Правда, как и в предыдущем случае, в автоматически сгенерированный код сеттера пришлось добавить оператор `repaint()`.

Свойства вида *Constrained* требуют проверки задаваемого значения свойства на принадлежность к области допустимых значений. Если *значение* не удовлетворяет этому условию, возбуждается *исключительная ситуация*. При изменении таких свойств порождается событие `VetoableChangeEvent`. *Слово* Vetoable происходит от "Veto *able*" – способный на накладывание ограничения, накладывание вето.

При задании свойств – массивов во всплывающем *меню*, вызываемом правой кнопкой мыши в узле **Bean Patterns**, следует пользоваться опцией **Add/Indexed Property**. Например, если мы выбрали параметры так, как указано на правом рисунке, приведенном выше, будет добавлен следующий код:

```
/**
 * Holds value of property arr.
 */
private double[] arr;

/**
 * Indexed getter for property arr.
 * @param index Index of the property.
 * @return Value of the property at <CODE>index</CODE>.
 */
public double getArr(int index) {
    return this.arr[index];
}

/**
 * Getter for property arr.
 * @return Value of property arr.
 */
public double[] getArr() {
    return this.arr;
}

/**
 * Indexed setter for property arr.
 * @param index Index of the property.
 * @param arr New value of the property at <CODE>index</CODE>.
 */
public void setArr(int index, double arr) {
    this.arr[index] = arr;
    propertySupport.firePropertyChange ("arr", null, null );
}

/**
 * Setter for property arr.
 * @param arr New value of property arr.
```

```
     */
    public void setArr(double[] arr) {
        double[] oldArr = this.arr;
        this.arr = arr;
        propertySupport.firePropertyChange ("arr", oldArr, arr);
    }
```

После добавления нового свойства следует заново скомпилировать проект (**Build main project - F11**). При этом, если при визуальном проектировании (**Design**) выделить *компонент* **jTitledPanel1**, его новые свойства появятся в окне **jTitledPanel1[JTitledPanel]-Properties/Properties** сразу после компиляции проекта.

12.5. Добавление в компонент новых событий

Поскольку мы наследуем *компонент* от класса `JPanel` , большинство необходимых событий он уже умеет генерировать. Но в ряде случаев может потребоваться другой *тип событий*. В ряде случаев имеется возможность использовать готовые интерфейсы слушателей. Например, мы хотим, чтобы возникло событие `java.awt.event.TextEvent` , связанное с изменением текста заголовка. "Обычная" панель **JPanel** не имела свойств, связанных с текстом, и это событие в ней не поддерживалось. *Интерфейс* `java.awt.event.TextListener` имеет всего один метод `textValueChanged(TextEvent e)` , так что в адаптере нет необходимости.

Для создания такого события для нашего компонента требуется использовать добавление поддержки события через *Bean* Patterns.

В *Java* имеется два типа *источников событий*:

- `Unicast Event Source` - источники порождают целевые объекты событий, которые передаются одному слушателю-приемнику. `"Cast"` - список исполнителей, `"Unit"` - единичный, `"Unicast"` - от Unit и *Cast* - один обработчик, `"Source"` - источник. В этом случае список слушателей не создается, а резервируется место только для одного.
- `Multicast Event Source` - источник порождают целевые объекты событий, которые передаются нескольким слушателям-приемникам. `"Multi"` - много, `"Multicast"` - много обработчиков. В этом случае для событий данного типа создается список слушателей.

Очевидно, что для некоторых типов событий обязательно создавать *список* слушателей. Хотя в случае `Unicast Event Source` реализация оказывается проще. В нашем случае в списке нет необходимости, поэтому выберем первый вариант. В выпадающем списке диалога имеется возможность выбрать некоторые интерфейсы слушателей из пакетов `java.awt.event` и `javax.swing.event` . Однако нам нужен *интерфейс*, поддерживающий событие `java.awt.event.TextEvent` , который в нем отсутствует. Поэтому мы укажем имя интерфейса `java.awt.event.TextListener` вручную.

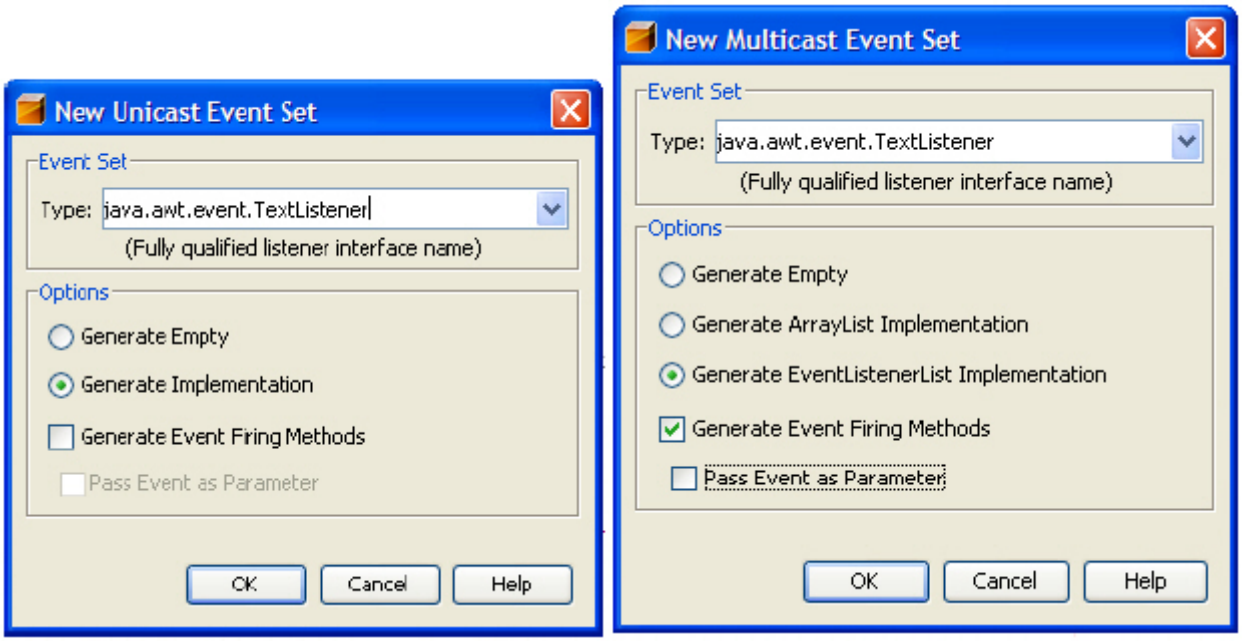


Рис. 12.7. Задание в компоненте нового типа событий

При выборе варианта `Generate Empty` ("Генерировать Пустое") в коде компонента появятся пустые реализации методов добавления и удаления слушателей. Это достаточно экзотический случай, поэтому мы

выберем вариант Generate Implementation ("Генерировать Реализацию").

Если выбрать опцию Generate Event Firing Methods ("Генерировать методы "выстреливания событиями" "), происходит автоматическая генерация заготовок fire-методов fire *ИмяСобытия*, предназначенных для оповещения зарегистрированных слушателей. В случае *Unicast*-источников обход списка слушателй не требуется, поэтому нам нет необходимости отмечать данный *пункт*. А вот в случае *Multicast*-источника это наиболее часто требующееся решение. При этом обычно бывает желательно передавать в методы событие как *параметр* – и для этого надо выбрать опцию Pass Event as Parameter ("Передавать событие как *параметр*").

Если *пункт* **Generate Event Firing Methods** отмечен, а *опция* **Pass Event as Parameter** не выбрана, событие не будет передаваться в fire-методы, а будет создано в самом fire-методе. Именно так происходит в примере для свойства sampleProperty , где вызов

```
propertySupport.firePropertyChange(PROP_SAMPLE_PROPERTY,
                                   oldValue, sampleProperty)
```

приводит к порождению внутри метода firePropertyChange события PropertyChange .

Генерация кода, поддерживающего *интерфейс* java.*awt*.event.TextListener , приведет для *Unicast*-источника без генерации fire-методов к появлению следующего кода:

```
/**
 * Utility field holding the TextListener.
 */
private transient java.awt.event.TextListener textListener = null;

/**
 * Registers TextListener to receive events.
 * @param listener The listener to register.
 */
public synchronized void addTextListener(java.awt.event.TextListener
                    listener) throws java.util.TooManyListenersException {
    if (textListener != null) {
        throw new java.util.TooManyListenersException ();
    }
    textListener = listener;
}

/**
 * Removes TextListener from the list of listeners.
 * @param listener The listener to remove.
 */
public synchronized void removeTextListener(java.awt.event.TextListener
listener) {
    textListener = null;
}
```

Но в этом случае добавлять код, обеспечивающий порождение события, должен программист.

Если выбрана *опция* генерации fire-методов без передачи события как параметра, появится следующий *дополнительный код* по сравнению с предыдущим вариантом:

```
/**
 * Notifies the registered listener about the event.
 *
 * @param object Parameter #1 of the <CODE>TextEvent<CODE> constructor.
 * @param i Parameter #2 of the <CODE>TextEvent<CODE> constructor.
 */
```

```
private void fireTextListenerTextValueChanged(java.lang.Object object,int i){
    if (textListener == null) return;
    java.awt.event.TextEvent e = new java.awt.event.TextEvent (object, i);
    textListener.textValueChanged (e);
}
```

Этот код также не обеспечивает автоматической генерации события в нашем компоненте, но дает возможность сделать это путем добавления одной строчки в код метода `setTitle` – перед вызовом метода `repaint()` мы напишем

```
fireTextListenerTextValueChanged(this,
                                 java.awt.event.TextEvent.TEXT_VALUE_CHANGED);
```

В качестве первого параметра fire-метода идет *ссылка* на *объект*-источник события, в качестве второго – *идентификатор типа события*. Найти, где задается *идентификатор*, просто

достаточно перейти мышкой по гиперссылке `http://java.awt.event.TextEvent`, появляющейся в среде разработке при нажатии клавиши **<CTRL>**, и посмотреть исходный код конструктора.

данную гиперссылку можно получить в строке

```
java.awt.event.TextEvent e = new java.awt.event.TextEvent (object, i);
```

в *теле метода* `fireTextListenerTextValueChanged` , в которой, собственно, и используется этот не очень понятный с первого взгляда параметр.

Теперь после компиляции проекта мы можем назначать обработчики событий типа `TextValueChanged` нашему компоненту. К сожалению, для того, чтобы событие `textValueChanged` появилось в списке событий компонента в окне **jTitledPanel1[JTitledPanel]-Properties/Events**, требуется закрыть среду NetBeans и зайти в нее вновь. Для свойств этот баг отсутствует – они появляются в окне **jTitledPanel1[JTitledPanel]- Properties/ Properties** сразу после компиляции проекта.

Теперь для нашего компонента можно назначать и удалять обработчик события `textValueChanged` как непосредственно на этапе визуального проектирования, так и программным путем.

Покажем, каким образом это делается на этапе визуального проектирования. Выделим *компонент* **jTitledPanel1** и выберем в окне **jTitledPanel1[JTitledPanel]-Properties/Events** событие `textValueChanged` . Нажмем кнопку с тремя точками, находящуюся рядом с полем – вызовется диалог добавления и удаления обработчиков событий. Вообще, имеется правило – если название на кнопке или пункте *меню* кончается на три точки, это означает, что при нажатии на кнопку или выборе пункта *меню* появится какой-нибудь диалог.

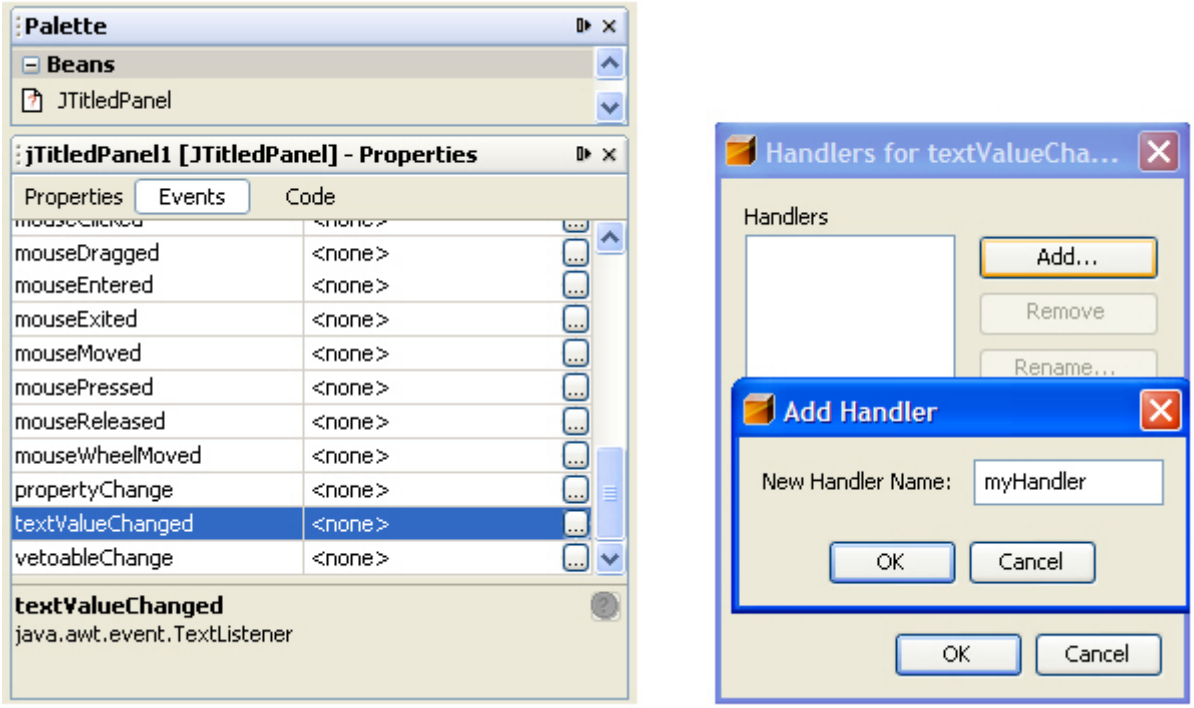


Рис. 12.8. Создание обработчика события на этапе визуального проектирования

Введем в качестве имени обработчика события (*event handler*) в качестве примера **"myHandler"** и нажмем **"OK"**. В списке обработчиков **Handlers** формы **"Handlers for textValueChanged"** появится имя **myHandler**. При закрытии этой формы по нажатию **"OK"** в исходном коде приложения (а не компонента!) появится код

```
private void myHandler(java.awt.event.TextEvent evt) {
// TODO add your handling code here:
    }
```

Вместо комментария `"// TODO add your handling code here:"` , как обычно, следует написать свой код обработчика. Например, такой:

```
javax.swing.JOptionPane.showMessageDialog( null,"Text="+
                                          jTitledPanel1.getTitle() );
```

Краткие итоги

Компонент – это:

- автономный элемент программного обеспечения, предназначенный для многократного использования, который может распространяться для использования в других программах в виде скомпилированного кода класса;
- подключение к этим программам осуществляется с помощью интерфейсов;
- взаимодействие с программной средой осуществляется по событиям, причем в программе, использующей компонент, можно назначать обработчики событий, на которые умеет реагировать компонент.

Компонент *JavaBeans* является классом Java и имеет три типа атрибутов:

- *Методы* компонента *JavaBeans* не отличаются от других методов объектов в Java. Они описывают поведение компонента. Общедоступные методы компонента могут вызываться из других компонентов или из обработчиков событий.
- *Свойства* (Properties) компонента *JavaBeans* характеризуют его внешний вид и поведение и могут быть изменены в процессе визуального проектирования. Это можно сделать с помощью редактора свойств (Property Editor), а некоторые из свойств – вручную (положение компонента, его размер, текст). Свойство задается комбинацией геттера и сеттера (метода по чтению и метода по записи).
- *События* (Events) используются для связи между компонентами. При помещении компонента на *экранную форму* среда разработки исследует компоненты и определяет, какие программные события данный компонент может порождать (рассылать) и какие – получать (обрабатывать).

Наиболее простым способом создания компонента является использование мастера среды NetBeans.

Методы с названием `fire` *ИмяСобытия* ("fire" – "стрелять", в данном случае – "выстрелить событием") осуществляют поочередный вызов зарегистрированных слушателей из списка для данного события, передавая им событие на обработку.

Методы `add` *ИмяСобытия* `Listener` и `remove` *ИмяСобытия* `Listener` обеспечивают для компонента возможность добавления и удаления объекта слушателя – обработчика события.

Если требуется задать в компоненте новые свойства или обеспечить генерацию компонентом новых типов событий, следует воспользоваться мастером, вызываемым через узел *Bean* Patterns. Таким же образом удаляются свойства и события компонента.

Для того, чтобы добавить компонент в палитру, следует открыть файл компонента в окне редактора исходного кода, и в меню **Tools** выбрать пункт **Add to Palette**. После чего в появившемся диалоге выбрать палитру, на которую будет добавлен компонент. Желательно выбирать **Beans**, чтобы не путать наши компоненты со стандартными.

Свойства вида **Bound** – обычные свойства. При изменении таких свойств порождается событие `PropertyChange` . Свойства вида *Constrained* требуют проверки задаваемого значения свойства на принадлежность к области допустимых значений. Если значение не удовлетворяет этому условию, возбуждается исключительная ситуация. При изменении таких свойств порождается событие `VetoableChangeEvent` .

В Java имеется два типа *источников событий*:

- *Unicast* Event Source – источники порождают целевые объекты событий, которые передаются одному слушателю-приемнику. В этом случае список слушателей не создается, а резервируется место только для одного обработчика.

Multicast Event Source – источник порождают целевые объекты событий, которые передаются нескольким слушателям-приемникам. В этом случае для событий данного типа создается список слушателей.

Генерация события в компоненте обеспечивается вручную вызовом fire-метода или другим способом.

Задания

Создать собственный компонент JTitledPane , описанный в данной лекции.

Усовершенствовать компонент, обеспечив добавление в него свойства titleColor . Подсказка: установка красного цвета рисования в качестве текущего цвета вывода графических примитивов для объекта Graphics g осуществляется вызовом метода g.setColor(Color.red) .

Усовершенствовать компонент, обеспечив генерацию в нем событий типа TitleShiftEvent , предварительно создав соответствующий интерфейс. По желанию можно добавить и событие изменения цвета заголовка.

*По желанию учащегося: Усовершенствовать компонент, обеспечив добавление в него свойства titleFont и методов, обеспечивающих установку нужного размера и типа фонта.