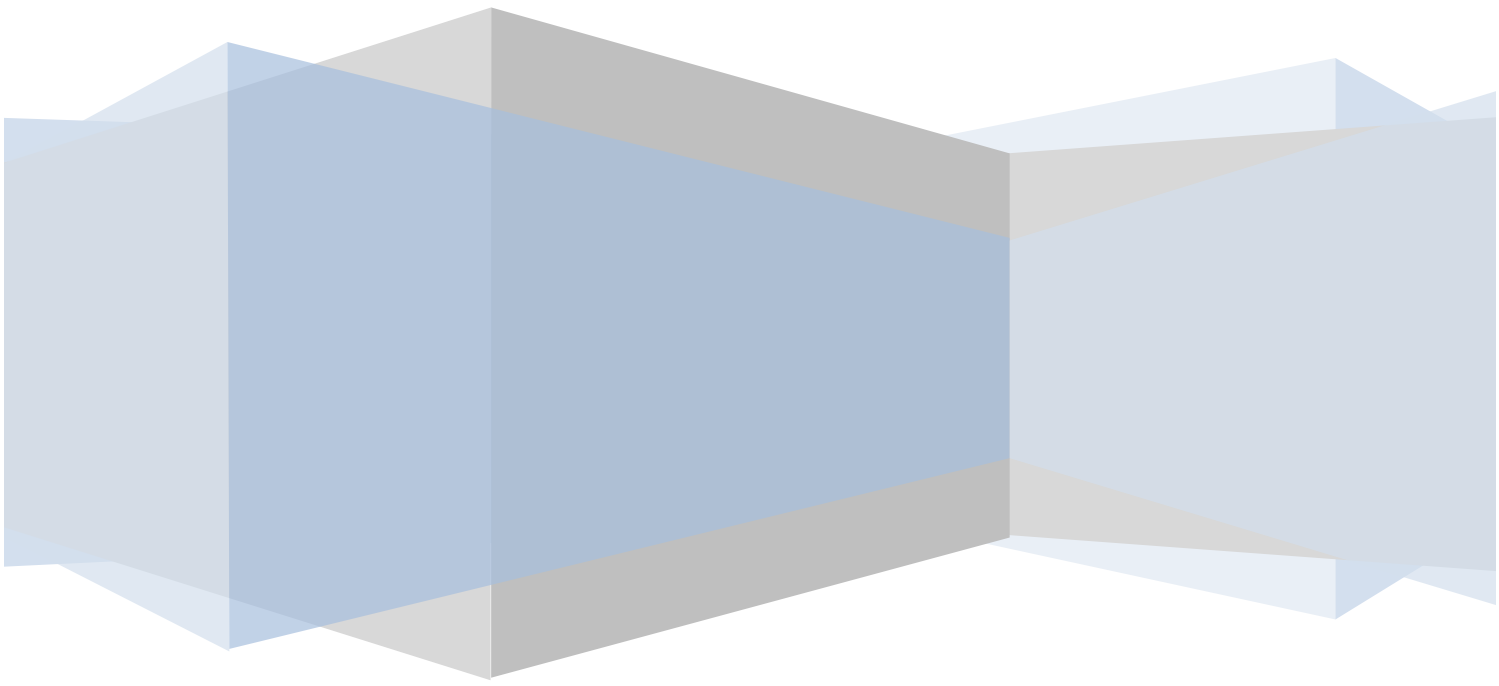
JavaBeans

Übersicht, Eigenschaften, heutiger Stand

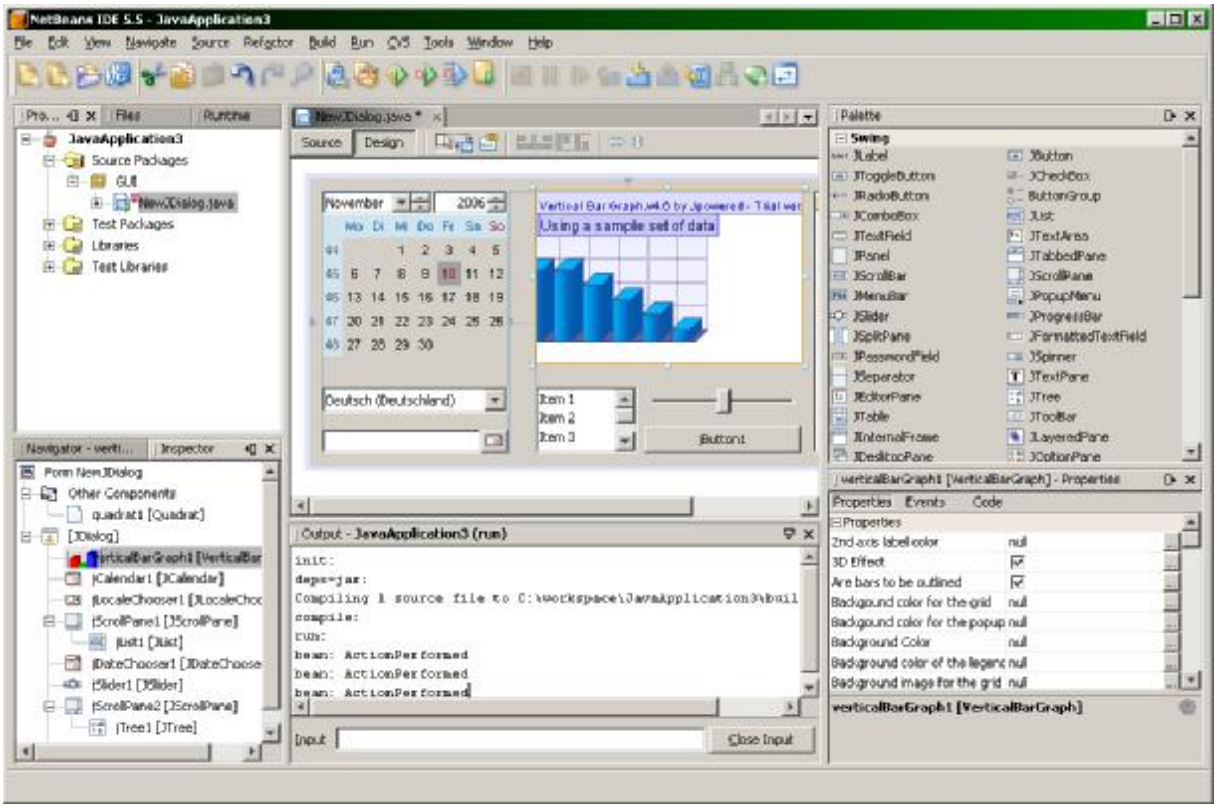


Inhaltsverzeichnis

Grundlagen

Als Beans werden in Java eigenständige, wieder verwendbare Softwarekomponenten zum Aufbau von Applets und Applikationen bezeichnet.

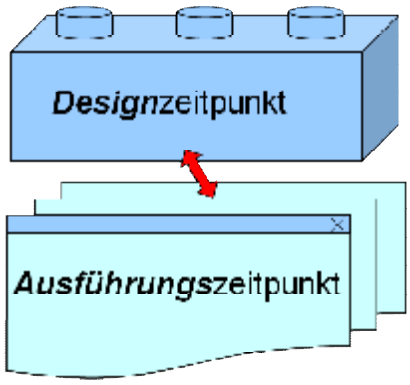
Sun definiert Komponenten so, dass sie sich visuell durch so genannte Application-Builder manipulieren lassen. Die Bean kann auf dem Bildschirm dargestellt und angepasst werden.



Vielen von uns dürfte dies von GUI-Buildern bekannt vorkommen. Hier gibt es auch eine Verwandtschaft, denn in Java sind die AWT- und Swing-Komponenten ebenfalls nichts anderes als Beans. Etwa eine Schaltfläche oder eine Liste. Die Schaltfläche lässt sich auf einem Arbeitsblatt positionieren, und über einen speziellen Dialog des Builders kann der Text oder die Schriftfarbe angepasst werden.

Eine Komponente muss aber nicht zwingend grafisch sein. Genauso gut kann es eine Datenstruktur oder eine FTP-Klasse sein. Eine häufig eingesetzte Komponente ist etwa ein Timer, der in festen Abständen ein Ereignis meldet. Die Beans-Komponenten können wiederum andere Beans aufnehmen und als Container arbeiten. Auf diese Weise wird eine Hierarchie von Komponenten geschaffen.

Entwicklung mit Beans



Bei der Entwicklung einer Anwendung mit grafischen Komponenten wird zweckmäßigerweise zwischen Designzeitpunkt und Ausführungszeitpunkt unterschieden.

Beide bezeichnen eigentlich Zeiträume und stehen für die beiden unterschiedlichen Phasen der Entwicklung einer Anwendung und ihrer späteren Ausführung. Zum Designzeitpunkt werden die Komponenten in einem grafischen Editor des Entwicklungswerkzeugs ausgewählt, in einem Formular platziert und an die Erfordernisse der Anwendung angepasst.

Ein großer Teil der bei den Beans realisierten Konzepte spielt nur für den Designzeitraum eine Rolle. Spielt dieser keine Rolle (beispielsweise weil die Komponenten nicht in einem GUI-Designer, sondern per Source-Code-Anweisungen eingebunden und konfiguriert werden), verlieren Beans einiges von ihrer Nützlichkeit, und herkömmliche Techniken der Wiederverwendung in objektorientierten Programmen können an ihre Stelle treten.

Eigenschaften

Sehen wir uns mal die wichtigsten Eigenschaften von Beans:

- Eine Bean wird im Programm durch ein Objekt repräsentiert. Das gilt sowohl für den Designzeitpunkt, bei dem es vom GUI-Designer instanziert wird, als auch für den Ausführungszeitpunkt, bei dem die Anwendung dafür verantwortlich ist, das Objekt zu konstruieren. Eine Bean benötigt dazu notwendigerweise einen parameterlosen Konstruktor.
- Sie besitzt konfigurierbare Eigenschaften, die ihr Verhalten festlegen. Diese Eigenschaften werden ausschließlich über Methodenaufrufe modifiziert, deren Namen und Parameterstrukturen genau definierten Designkonventionen folgen. Damit können sie vom GUI-Designer automatisch erkannt und dem Entwickler in einem Editor zur Auswahl angeboten werden. Die Eigenschaften werden dabei üblicherweise in einem Property Sheet angeboten und erlauben es den Zustand der Bean interaktiv zu verändern.
- Eine Bean ist serialisierbar und deserialisierbar. So können die zum Designzeitpunkt vom Entwickler vorgenommenen Einstellungen gespeichert und zum Ausführungszeitpunkt rekonstruiert werden. Ein spezieller Externalisierungsmechanismus erlaubt dem Entwickler die Definition eines eigenen Speicherformats, zum Beispiel als XML-Datei.
- Eine Bean sendet Events, um registrierte Listener über Änderungen ihres internen Zustands oder über sonstige wichtige Ereignisse zu unterrichten.
- Einer Bean kann optional ein Informations-Objekt zugeordnet werden, mit dessen Hilfe dem GUI-Designer weitere Informationen zur Konfiguration übergeben und zusätzliche Hilfsmittel, wie spezialisierte Eigenschafteneditoren oder Konfigurationsassistenten, zur Verfügung gestellt werden können.

Bean-Klasse

Als erstes: Es gibt keine Bean-Klasse, die man zum Erstellen von Beans erweitern könnte! Zwar erweitern sichtbare Beans direkt oder indirekt die Klasse Component, nicht sichtbare Beans müssen aber überhaupt keine bestimmte Superklasse erweitern.

Namensmuster für Eigenschaften

Ein wesentliches Designmerkmal von Beans sind Eigenschaften. Eine Eigenschaft ist eigentlich nur eine Membervariable, die über öffentliche Methoden gelesen und geschrieben werden kann. Eine Bean kann beliebig viele Eigenschaften haben, jede von ihnen besitzt einen Namen und einen Datentyp. Die Bean-Designkonventionen schreiben vor, dass auf eine Eigenschaft mit dem Namen „eigenschaftsName“ und dem Datentyp „X“ über folgende Methoden zugegriffen werden soll.

```
public X getEigenschaftsName()

public void setEigenschaftsName(X x)
```

Beim Namensmuster für Eigenschaften entspricht dieses Methodenpaar einer Lese-/Schreibeigenschaft vom Typ X. Und wenn es zwar eine get-Methode, nicht jedoch zugehörige set-Methode gibt, definiert man eine schreibgeschützte (Nur-Lese-) Eigenschaft.

Falls eine get-Methode nur einen Wahrheitswert zurückgibt, ermöglicht eine isXXX()-Methode eine Vereinfachung. Der allgemeine Typ ist:

```
public boolean isEigenschaftsName()

public void setEigenschaftsName(boolean wert)
```

Mit diesen Grundkenntnissen kann man schon eine einfache Bean erschaffen.

Beispiel: Eine sehr einfache Bean, die eine Quadratzahl berechnet.

```
package p;

public class Quadrat implements java.io.Serializable {
	private int zahl;

	public int getZahl() {
		return zahl;
	}

	public void setZahl(int zahl) {
		this.zahl = zahl;
	}

	public int getQuadrat() {
		return this.zahl * this.zahl;
	}
}
```

Hinweis: alle Beans sollten in Paketen liegen, da einige IDEs mit Klassen im Standardpaket Schwierigkeiten haben.

Beginnen wir nun unsere tolle Bean in einen Application-Builder einzubauen. Dazu muss die Softwarekomponente aber in einem speziellen Format vorliegen: in Jar-Dateien.

Nach dem Compilieren der Klasse wird die Datei mit einem zusätzlichen Manifest in eine Jar-Datei gepackt. Damit der Application-Builder erkennen kann, dass es sich dabei um eine Bean handelt, erwartet er eine bestimmte Information in einer Manifest-Datei.

```
Listing: Quadrat.mf
Manifest-Version: 1.0
Name: p/Qudrat.class
Java-Bean: True
```

Im nächsten Schritt packen wir mit jar die Klassen, sowie die Manifest-Datei zu einem Archiv zusammen.

```
$ jar cfm Quadrat.jar Quadrat.mf p/Quadrat.class
```

Namensmuster für Ereignisse

Bei Ereignissen ist das Namensmuster noch einfacher gestrickt. Eine Programmierumgebung für Beans geht davon aus, dass eine Bean Ereignisse generiert, wenn man Methoden, die Ereignisempfänger hinzufügen und entfernen, bereitstellen. Nehmen wir an, dass eine Bean Ereignisse von Typ ClickEvent generiert.

Ereignisse von Typ *Click*Event
Empfängerschnittstelle: *Click*Listener

Alle Ereignisnamen müssen mit Event enden. Dann muss die Empfängerschnittstelle ClickListener heißen und die Methoden zum Hinzufügen und entfernen eines Empfängers sind folgendermaßen zu benennen.

```
      public void addClickListener(ClickListener e)
      public void removeClickListener(ClickListener e)
```

Eigenschaftstypen von Beans

Eine komplexe Bean hat verschiedene Arten von Eigenschaften, die man in einem Programmierwerkzeug freilegen sollte, damit sie ein Benutzer zur Entwurfzeit setzen oder zur Laufzeit abrufen kann.

- Einfache Eigenschaften
    - Boolesche Eigenschaften
- Indizierte Eigenschaften
- *Gebundene Eigenschaften*
- *Eingeschränkte Eigenschaften*

Indizierte Eigenschaften

Anstelle eines Einzelwertes kann eine Eigenschaft auch durch ein Array von Werten repräsentiert werden. Es werden aber keine Arrays übergeben, sondern die Methoden erwarten jeweils den Index der gewünschten Eigenschaft als zusätzliches Argument. Für das Einhalten der Arraygrenzen ist der Aufrufer selbst verantwortlich! Eine Lösung ist, den falschen Index mit ArrayIndexOutOfBounds zu quittieren.

```
      public X getName(int index) { … }_
      public void setName(int index, X newValue) { … }
```

Neben primitiven Typen ist auch die Übergabe von Objekttypen erlaubt. Bei Objekteigenschaften kann allerdings nicht unbedingt davon ausgegangen werden, dass ein geeigneter Editor zur Verfügung steht. Bei einem selbstdefinierten Objekttyp muss der Entwickler selbst einen Editor entwickeln und dem Designer zur Verfügung stellen.

Gebundene Eigenschaften

Die gebundenen Eigenschaften einer Bean erlauben es, andere Komponenten über eine Zustandsänderung zu informieren. Damit werden Eigenschaften und Ereignisse direkt miteinander verbunden.

- Die interessierten Beans empfangen vom Auslöser ein PropertyChange-Ereignis, das sie auswerten können. Die Beans lassen sich mit addPropertyChangeListener() und removePropertyChangeListener()-Methoden als Zuhörer einfügen und abhängen.
- Bei einer Veränderung werden alle registrierten Zuhörer durch ein PropertyChangeEvent informiert. Die Interessierten implementieren dafür PropertyChangeListener.
- Über dieses Ereignis-Objekt erfahren wir den alten und neuen Wert, gleichzeitig erfahren wir etwas über den Typ und den Namen der Eigenschaft.
- Die Zuhörer werden erst nach der Änderung des internen Zustands informiert.

Beispiel: Diese Komponente ändert den internen String über setString(). Nach der Änderung werden alle Listener informiert. Sie bewirkt sonst nichts Großartiges.

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

public class PropertyChangeStringBean extends Canvas {

      private String string = "";
      private PropertyChangeSupport changes =
                  new PropertyChangeSupport(this);

      public void setString(String newString) {
            String oldString = string;
            string = newString;
            changes.firePropertyChange( "string", oldString, newString );
      }

      public String getString() {
            return string;
      }

      @Override
      public void addPropertyChangeListener( PropertyChangeListener l ) {
            changes.addPropertyChangeListener( l );
      }

      @Override
      public void removePropertyChangeListener( PropertyChangeListener l ){
            changes.removePropertyChangeListener( l );
      }
}
```

Der Methode setString() kommt zentrale Bedeutung zu. Der erste Parameter von firePropertyChange() ist der Name der Eigenschaft. Er ist nur für die Änderung von Belang und hat

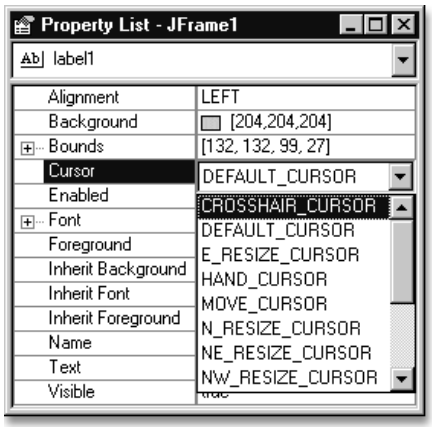
nichts mit dem Namen der Bean-Eigenschaft gemeinsam. Die Methode informiert alle angemeldeten Zuhörer über die Änderung mit einem PropertyChangeEvent. Im Ereignis stehen der alte und der neue Stand des Werts. Angemeldete Listener können dann darauf reagieren – etwa indem abhängige Klassen ihr Layout ändern.

Eingeschränkte Eigenschaften

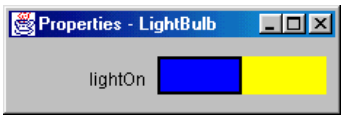
Bei gebundenen Eigenschaften informieren Komponenten andere Komponenten, wenn sich ein Zustand ändert. Möglicherweise haben diese Komponenten jedoch etwas dagegen. In diesem Fall kann die hörende Bean ihr Veto mit einer PropertyVetoException einlegen und so eine Werteänderung verhindern. Bevor wir die Änderung durchführen, holen wir also die Zustimmung dafür ein. Eine Veto-Eigenschaft ist insofern mit der gebundenen Eigenschaft verwandt, nur dass diese nicht in der Lage ist zu meckern. Programmieren wir eine setXXX()-Methode mit Veto, kann der Aufrufer nun durch eine PropertyVetoException nach außen anzeigen, dass er dagegen war.

Bean-Eigenschaften anpassen

Haben wir zum Beispiel als Eigenschaft eine Zeichenkette (wie der angezeigte Text eines Label-Objekts), können wir diese einfach in das Feld eintragen.



Mit einem eigenen Property-Editor sind wir jedoch nicht auf die einfachen Datentypen beschränkt. Was ist, wenn etwa eine Komponente die Auswahl zwischen $ und Euro anbieten will? Wir können dem Benutzer nicht zumuten, dies in Zahlen einzugeben. Auch bei einem Label gibt es zum Beispiel die Möglichkeit, einen Cursor aus einer Liste auszuwählen.



Reicht auch der Editor nicht aus, zum Beispiel bei einer Farbe, die wir gerne aus einem Farbkreis auswählen wollen, lässt sich zudem ein Customizer definieren, der noch einen Schritt weiter geht, denn für einen Customizer ist ein eigenes Fenster vorgesehen.

BeanInfo-Klasse

Durch Introspection/Reflection existiert ein leistungsfähiger Mechanismus, um die Eigenschaften und Ereignisse zur Laufzeit auszulesen. In der Regel nimmt die Entwicklungsumgebung dafür Methoden, die sich an die Namenskonvention halten. Es gibt aber noch eine zweite Möglichkeit, und die lässt sich über eine Bean-Information-Klasse nutzen. Sie bietet folgende Funktionalität:

- explizites Auflisten der freigegebenen Leistungen, die nach außen sichtbar sein sollen
- Zuordnung eines mit der Bean verwendeten Icons
- Anmeldung einer Customizer-Klasse

Sind die freigegebenen Leistungen aufgelistet, wird damit eine Untersuchung der Bean-Klassen auf die Namensgebung verhindert. Es gibt daher zur Freigabe der Eigenschaften und Ereignisse spezielle Methoden, die von uns gefüllt werden, indem wir jede Eigenschaft auflisten. Jede Methode nutzt zur Beschreibung der Leistungen so genannte Deskriptoren. Gibt es keinen Deskriptor, wird die jeweilige

Eigenschaft, Methode oder das Ereignis nicht veröffentlicht. Gibt die Anfrage-Methode aus der Bean-Information-Klasse null zurück, wird für die jeweilige Eigenschaft/Event/Methode Reflection genutzt.

Literaturhinweise

- Java ist auch eine Insel
  von Christian Ullenboom, Galileo Computing
- Handbuch der Java-Programmierung
  von Guido Krüger, Addison-Wesley
- Core Java 2 Bd.2 Expertenwissen
  C.S.Horstmann, G.Cornell, Addison-Wesley

- Sun Developer Network - JavaBeans
  http://java.sun.com/products/javabeans/index.jsp
- Sun Developer Network - API Specifications / JavaBeans Spec
  http://java.sun.com/products/javabeans/docs/spec.htmls
- Wikipedia.de - JavaBeans
  http://de.wikipedia.org/wiki/JavaBeans
- bean-builder: Home of The Bean Builder
  http://bean-builder.dev.java.net
- ActiveX Bridge
  http://java.sun.com/j2se/1.5.0/docs/guide/beans/axbridge/developerguide
- Artikel Using XMLEncoder zum Persistenzmodell
  http://java.sun.com/products/jfc/tsc/articles/persistence4