Common-Controls
Guided Tour
TreeControl

Version 1.6 - Stand: 14. Januar 2006

Common Controls

Herausgeber:
SCC Informationssysteme GmbH
64367 Mühltal

Tel: +49 (0) 6151 / 13 6 31 12
Internet http://www.scc-gmbh.com

Product Site:
http://www.common-controls.com

Inhaltsverzeichnis

1 Guided Tour TreeControl

1.1 Gegenstand

Diese Übung demonstriert den Einsatz des TreeControls. Dieses Kontrollelement erzeugt einen Baum, dessen Knoten sich auf- und zugeklappen lassen. Der Programmierer stellt hierfür lediglich die Anzeigedaten (das Datenmodell) durch Implementierung eines einfachen Interfaces bereit.

Das TreeControl bietet die folgenden Features:

- Die Linien auf der obersten Ebene können ein oder ausgeblendet werden. Zu den Knoten und den Blättern lassen sich unterschiedliche Bilder in einer Immag Map hinterlegen. Dabei wird die Zuordnung der Bilder zu dem jeweiligen Baumknoten mit Hilfe von Regulären Ausdrücken vorgenommen.
- Das Kontrollelement verwaltet selbstständig alle notwendigen Zustandsdaten über mehrere Server Roundtrips hinweg. Dazu zählt beispielsweise der auf- oder zugeklappte Status der Baumknoten.
- Vor den Baumeinträgen können Checkboxen ein oder ausgeblendet werden. Bei der Auswahl eines Knotens auf einer unteren Ebene, werden automatisch alle übergeordneten Knoten markiert.

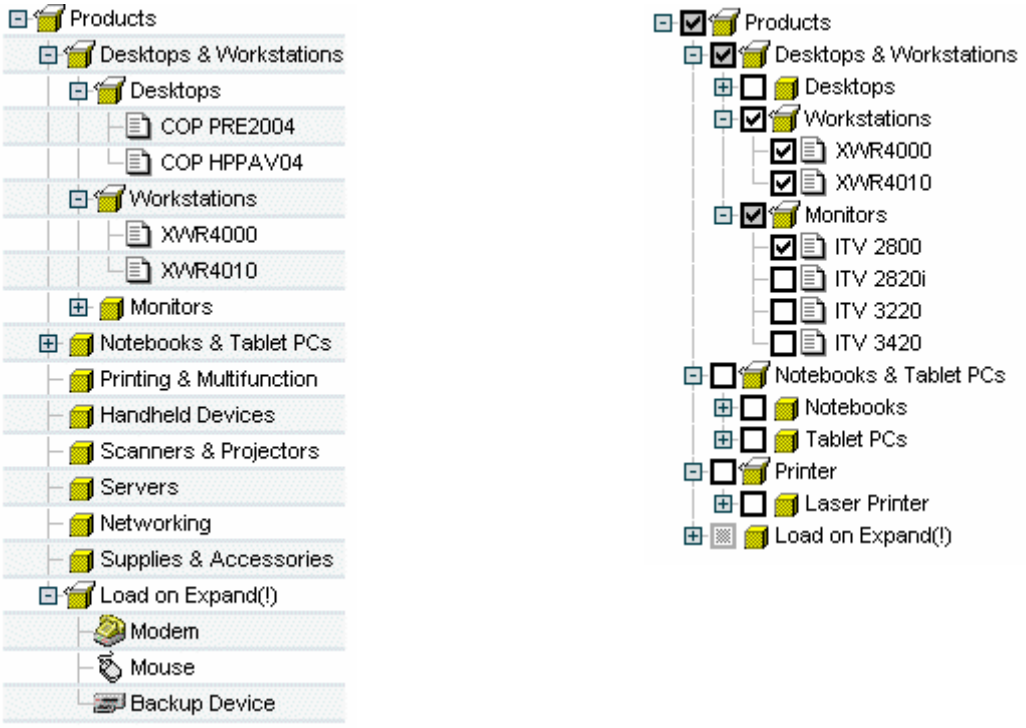


Abbildung 1: Beispieldarstellung TreeControl

Zum Einsatz des TreeControls lediglich folgende Schritte notwendig:

1. Auswahl des Designs der Benutzeroberfläche
2. Erstellung einer Actionklasse
3. Instanziierung eines TreeControls
4. Bereitstellung der Anzeigedaten
5. Konfiguration des Baumes innerhalb der JSP-Seite

1.2 Registrierung der Painterfactory

Zuerst erfolgt die Registrierung der Painterfactory. Sie legt fest, welches Design die Benutzeroberfläche erhält. Dies kann anwendungsweit in der init()-Methode des Frontcontroler-Servlets geschehen.[1] Wir wählen hier das Standarddesign, welches uns der DefaultPainter bereitstellt[2].

```
import javax.servlet.ServletException;

import org.apache.struts.action.ActionServlet;
import com.cc.framework.ui.painter.PainterFactory;
import com.cc.framework.ui.painter.def.DefPainterFactory;
import com.cc.framework.ui.painter.html.HtmlPainterFactory;

public class MyFrontController extends ActionServlet {

    public void init() throws ServletException {

       super.init();

       // Register all Painter Factories with the preferred GUI-Design
       // In this case we use the Default-Design.
       PainterFactory.registerApplicationPainter (
              getServletContext (), DefPainterFactory.instance());
       PainterFactory.registerApplicationPainter (
              getServletContext (), HtmlPainterFactory.instance());
    }
}
```

1.3 Ableitung der Action Klasse

In unserem Baum wollen wir Produktgruppen und Produkte anzeigen. Daher soll die Action-Klasse, welche das Laden und Befüllen des TreeControls übernimmt, die Bezeichnung „ProductTreeBrowseAction“ tragen. Die Actionklasse wird von der Klasse FWAction abgeleitet, welche die Struts-Action Klasse kapselt und um Funktionalitäten des Präsentationsframeworks erweitert. Dabei wird anstelle der execute()-Methode die doExecute()-Methode aufgerufen. [FWAction ist von org.apache.struts.Action abgeleitet] Sie erhält beim Aufruf den ActionContext, über den der Zugriff auf weitere Objekte, wie das Request- und Response-Objekt gekapselt ist.

```
import java.lang.Exception;

import com.cc.framework.adapter.struts.FWAction;
import com.cc.framework.adapter.struts.ActionContext;

public class ProductTreeBrowseAction extends FWAction {

    /**
     * @see com.cc.framework.adapter.struts.FWAction#doExecute(ActionContext)
     */
    public void doExecute(ActionContext ctx) throws Exception {
        // In the next chapter, we will instantiate
        // our TreeControls with the DisplayData
    }
}
```

[1] Wenn der einzelne Benutzer zwischen verschiedenen Oberflächendesigns wählen können soll, dann werden zusätzliche PainterFactorys in der Benutzersession registriert. Dies erfolgt meist in der LoginAction mit PainterFactory.registerSessionPainter() im Session Scope.

[2] Weitere Designs (PainterFactories) sind im Lieferumfang der Professional Edition enthalten, oder können selbst entwickelt werden.

1.4 Instanziierung des TreeControls

Nun wird das TreeControl innerhalb unserer Action instanziiert und mit den Anzeigedaten gefüllt. Das Datenmodell wird dem Kontrollelement über die setDataModel()-Methode zugeordnet. Die Methode nimmt als Argument ein Objekt vom Typ **TreeGroupDataModel** entgegen. Dabei handelt es sich um ein Interface welches den Zugriff auf die Anzeigedaten des Baumes bereitstellt. Es ist die Aufgabe des Anwendungsentwicklers eine entsprechende Implementierung zur Verfügung zu stellen.

```
import java.lang.Exception;

import com.cc.framework.adapter.struts.ActionContext;
import com.cc.framework.adapter.struts.FWAction;
import com.cc.framework.ui.control.TreeControl;

public class ProductTreeBrowseAction extends FWAction {

	/**
	 * @see com.cc.framework.adapter.struts.FWAction#doExecute(ActionContext)
	 */
	public void doExecute(ActionContext ctx) throws Exception {

		try {
			// first we get the Data for our Tree
			ProductGroupDsp data = DBProduct.fetch();

			// secondly create the TreeControl and populate it
			// with the Data to display
			TreeControl products = new TreeControl();
			products.setDataModel(data);

			// third put the TreeControl into the Session-Object.
			// Our Control is a statefull Object.
			ctx.session().setAttribute("products", products);
		}

		catch (Throwable t) {
			ctx.addGlobalError("Error: ", t);
		}

		// Display the Page with the Tree
		ctx.forwardToInput();
	}
}
```

1.5 Bereitstellen der Anzeigedaten

Der Baum besteht aus Gruppen- und Blattknoten. Gruppenknoten können wieder weitere Knoten enthalten (Composite Pattern). Entsprechend stehen für die beiden Knotentypen die Interfaces **TreeGroupDataModel** und **TreeNodeDataModel** bereit (TreeGroupDataModel erweitert dabei TreeNodeDatamodel). Mit ihrer Hilfe lässt sich die Baumstruktur einfach erzeugen.

Dabei wird zuerst der Wurzelknoten erzeugt unter dem dann weitere Gruppen oder Blätter eingehängt werden. Der Wurzelknoten wir dem TreeControl als Datenmodell übergeben.

```
// Root
ProductGroupDsp root = new ProductGroupDsp("0", "Products", "Root");

ProductGroupDsp group = null;
ProductGroupDsp subgroup = null;

// First Group under the Root-Element
group = new ProductGroupDsp("1201", "Workstations & Monitors");

subgroup = new ProductGroupDsp("2102", "Workstations");
subgroup.addChild( new ProductDsp("3005", "XWR4000", "product description"));
subgroup.addChild( new ProductDsp("3005", "XWR4010", "product description"));
group.addChild(subgroup);

subgroup = new ProductGroupDsp("2103", "Monitors");
subgroup.addChild( new ProductDsp("3101", "ITV 2800") );
subgroup.addChild( new ProductDsp("3102", "ITV 2820i") );
subgroup.addChild( new ProductDsp("3103", "ITV 3220") );
group.addChild(subgroup);

root.addChild(group);

// Secound Group under the Root-Element
group = new ProductGroupDsp("1204", "Printing & Multifunction");
root.addChild(group);
```

Die Klasse ProductGroupDsp

```
public class ProductGroupDsp extends ProductBaseDsp implements TreeGroupDataModel {

     /**
      * ParentNode
      */
     private TreeGroupDataModel parent = null;

     /**
      * ChildNodes
      */
     private Vector children = new Vector();

     // ------------------------------------------------
     //                Methods
     // ------------------------------------------------

     /**
      * Constructor
      * @param  key              Unique Key for the Group
      * @param  name             Name of the Group
      * @param  unknownChildren  true if the Childs should be loaded later
      *                          false if the Childnodes exists
      */
     public ProductGroupDsp(String key, String name) {
          super();

          this.key = key;
          this.name = name;
          this.type = "prodgroup";
     }

     /**
      * Constructor
      * @param  key              Unique Key for the Group
      * @param  name             Name of the Group
      * @param  description      description for the Group
      * @param  unknownChildren  true if the Childs should be loaded later
      *                          false if the Childnodes exists
      */
     public ProductGroupDsp(String key, String name, String description) {
          super();

          this.key = key;
          this.name = name;
          this.description = description;
          this.type = "prodgroup";
     }

     /**
      * @see TreeGroupDataModel#getChild(int)
      */
     public TreeNodeDataModel getChild(int index) {
          return (TreeNodeDataModel) children.elementAt(index);
     }
```

```
    /**
     * @see TreeGroupDataModel#addChild(TreeNodeDataModel)
     */
    public void addChild(TreeNodeDataModel child) {

        children.add(child);

        child.setParent(this);
    }

    /**
     * Returns the Number of ChildNodes
     * -1 = The Number of ChildNodes is unknown.
     *      When the Node opens an onExpandEx Event is generated
     *      and the Childs can be loaded at runtime
     * 0  = This Node has no ChildNodes
     * >0 = This Node has ChildNodes
     *
     * @see TreeGroupDataModel#size()
     */
    public int size() {
        return children.size();
    }

    /**
     * @see TreeNodeDataModel#getParent()
     */
    public TreeGroupDataModel getParent() {
        return parent;
    }

    /**
     * @see TreeNodeDataModel#setParent(TreeGroupDataModel)
     */
    public void setParent(TreeGroupDataModel parent) {
        this.parent = parent;
    }

    /**
     * @see TreeNodeDataModel#getParentKey()
     */
    public String getParentKey() {
        return parent.getUniqueKey();
    }

    /**
     * @see TreeNodeDataModel#getUniqueKey()
     */
    public String getUniqueKey() {
        return this.key;
    }
}
```

Die Klasse ProductDsp

```
public class ProductDsp extends ProductBaseDsp implements TreeNodeDataModel {

	/**
	 * ParentNode
	 */
	private TreeGroupDataModel parent = null;

	/**
	 * Constructor
	 * @param	key	Unique Productkey
	 * @param	name	Productname
	 */
	public ProductDsp(String key, String name) {
		super();

		this.key = key;
		this.name = name;
		this.type = "product";
	}

	/**
	 * Constructor
	 * @param	key		Unique Productkey
	 * @param	name		Productname
	 * @param	description	Product description
	 */
	public ProductDsp(String key, String name, String description) {
		super();

		this.key = key;
		this.name = name;
		this.description = description;
		this.type = "product";
	}

	public void setParent(TreeGroupDataModel parent) {
		this.parent = parent;
	}

	public TreeGroupDataModel getParent() { return parent; }
	public String getParentKey() { return parent.getUniqueKey(); }
	public String getUniqueKey() { return this.key; }

}
```

Die Klasse ProductBaseDsp

```
public class ProductBaseDsp {

	/**
	 * ProductKey
	 */
	protected String key = "";

	/**
	 * Name of the Product
	 */
	protected String name = "";

	/**
	 * Description for the Product
	 */
	protected String description = "";

	/**
	 * Type for the Node
	 */
	protected String type = "";


	/**
	 * Constructor
	 */
	public ProductBaseDsp() {
		super();
	}

	public String getName() { return name; }
	public String getDescription() { return description; }
	public String getType() { return type; }

}
```

1.6 Konfiguration des TreeControls innerhalb der JSP-Seite

Um das TreeControl-Tag auf einer JSP Seite einzusetzen, muss am Anfang der Seite die entsprechende Tag Library deklariert werden. Anschließend können die Common-Controls mit dem Präfix <ctrl:*tagname* /> verwendet werden. [Zudem muss die Aufnahme der Tag Bibliothek im Deployment-Deskriptor, der WEB-INF/web.xml Datei, erfolgen]

```
<%@ taglib uri="/WEB-INF/tlds/cc-controls.tld" prefix="ctrl" %>

<ctrl:tree
       name="products"
       action="sample201/productBrowse"
       root="true"
       linesAtRoot="true"
       labelProperty="name"
       imageProperty="type"
       expandMode="multiple"
       groupselect="true"
       checkboxes="false"/>
```

Damit sind alle notwendigen Schritte zur Nutzung des TreeControls abgeschlossen.
Das Auf- und Zuklappverhalten muß nicht selbst implementiert werden. Es wird durch das Kontrollelement selbst verwaltet. Dies gilt auch für die Selektionszustände von Checkboxen, die über das Attribut checkboxes=“true“ aktiviert werden können. Der Programmierer kann sich also auf die fachlichen Abläufe und die Bereitsstellung der Anzeigedaten konzentrieren.

Professional Edition: Mit dem Attribut runat=“client“ wird der Baum in einer JavaScript Version generiert und kann somit ohne Server Roundtrips auf- und zugeklappt werden. Diese Erhöhung des Benutzerkomforts bedarf jedoch keinerlei Änderungen im Anwendungsprogramm.

1.7 Tour Ende

Das TreeControl lässt sich schnell und einfach integrieren. Sein Standardverhalten lässt sich bei Bedarf auch überschreiben. So lassen sich auch spezielle Tree-Objektes erstellen, die bereits den Zugriff auf bestimmte fachliche Daten kapseln und wiederkehrend innerhalb eines Anwendungsprojektes verwendet werden können.

Durch die Konfigurationsmöglichkeiten in der JSP-Seite kann das Verhalten des TreeControls schnell geändert werden. Alternative Designs lassen sich durch eine Anpassung der bestehenden Painter einfach integrieren. Dabei werden unterschiedliche Designs auch parallel unterstützt.

Features des TreeControls:

- Implementiert ein automatisches auf- und zuklappen von Knoten
- Verwaltet den Zustand von optionalen Checkboxen.
- Verschiedene Konfigurationsmöglichkeiten (Ein- oder Ausblenden des Wurzelknotens, Änderung des Auf- und Zuklappverhalten einstellbar, Verbindungslinien auf oberster Ebene ausblendbar)
- Daten unterhalb eines Gruppenknotens können auch erst bei Öffnen der Gruppe geladen werden. Der Baum muss also nicht von Anfang an vollständig bekannt sein. Dies ist beispielsweise in Verbindung mit Datenbanken hilfreich. Beim ersten Aufklappen eines Knoten mit unbekannter Kinderzahl wird der Anwendung ein onExpandEx Ereignis gesendet.
- Design des TreeControls in der JSP oder auch serverseitig definierbar!
- Bildet Aktion, die auf dem Baum ausgeführt werden auf CallBack-Methoden in der Action-Klasse ab (Beispiele: onCheck, onExpand, onCollapse, onExpandEx).
- Bilder vor den Knoten/Blättern über Reguläre Ausdrücke zuordbar.
- Berechtigungsprüfung auf Knotenebene. Knoten können damit automatisch für unberechtigte Anwender ausgeblendet werden. (siehe Scurity Dokumentation)
- Layout durch Painterfactory an eigenen StyleGuide (Corporate Identity) anpassbar.
- Optimierter HTML-Code.
- Gleiches Look and Feel in Microsoft InternetExplorer > 5.x und Netscape Navigator > 7.x

1.8 Exkurs: Implementierung von Callback-Methoden

Das TreeControl generiert bei einem Klick auf ein **Label** automatisch ein onDrilldown-Event auf das der Programmierer innerhalb der Action-Klasse reagieren kann.

Um in unserem Beispiel auf dieses Ereignis zu reagieren, nehmen wir eine entsprechende Callback-Methode in der ProductTreeBrowseAction auf. Da wir die BusinessLogik nicht an dieser Stelle implementieren möchten, leiten wir das Ereignis an eine andere Action – ProductDisplayAction – weiter.

```
import java.lang.Exception;

import com.cc.framework.adapter.struts.ActionContext;
import com.cc.framework.adapter.struts.FWAction;
import com.cc.framework.ui.control.ControlActionContext;
import com.cc.framework.ui.control.TreeControl;

import com.cc.sampleapp.common.Forwards;

public class ProductTreeBrowseAction extends FWAction {

     /**
      * @see com.cc.framework.adapter.struts.FWAction#doExecute(ActionContext)
      */
     public void doExecute(ActionContext ctx) throws Exception {

          try {
               ProductGroupDsp data = DBProduct.fetch();
               TreeControl products = new TreeControl();
               products.setDataModel(data);
               ctx.session().setAttribute("products", products);
          }

          catch (Throwable t) {
               ctx.addGlobalError("Error: ", t);
          }

          // Display the Page with the Tree
          ctx.forwardToInput();
     }

      // ------------------------------------------------
      //          Tree-Control Event Handler
      // ------------------------------------------------


      /**
       * This Method is called when the TreeLabel is clicked
       * In our Example we switch to the DetailView, which shows
       * more Information about the node.
       * @param    ctx    ControlActionContext
       * @param    key    UniqueKey, as created in the Datamodel
       */
      public void products_onDrilldown(ControlActionContext ctx, String key)
            throws Exception {
            ctx.forwardByName(Forwards.DRILLDOWN, key);
      }

}
```

Der Name der CallBack-Methode setzt sich dabei aus dem Property-Namen des TreeControls – dem Namen der Bean - und dem eingetretenen Event zusammen. Da das TreeControl unter dem Namen „products“ in der Session abgelegt wurde, lautet der Name der CallBack-Methode **products**_onDrilldown.

1.9 Exkurs: Verwendung einer ImageMap

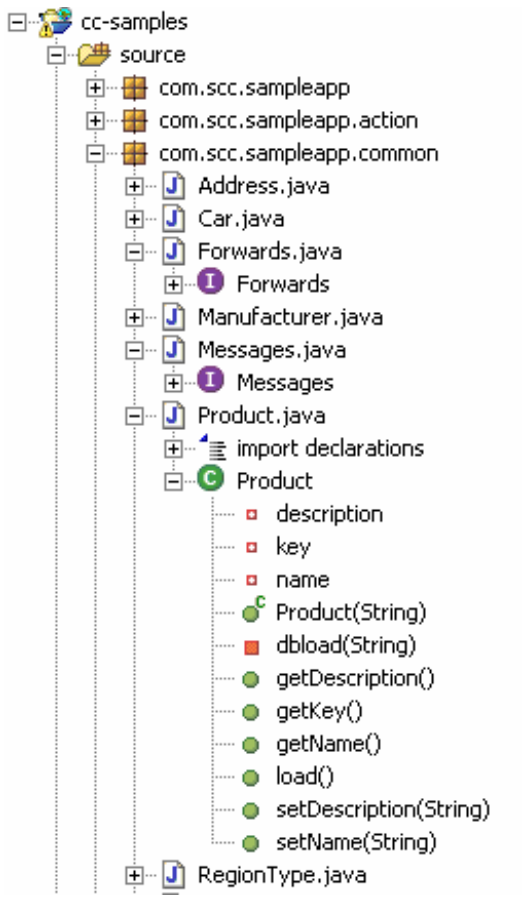
Das TreeControl verwendent bei der Darstellung für geöffnete und geschlossene Knoten vordefinierte Bilder, die sich bei Bedarf einfach austauschen lassen. Jedem Eintrag innerhalb des Baumes kann ein eigenes Bild zugeordnet werden.
Eine Möglichkeit hierzu stellt die Verwendung einer ImagMap dar. Diese wird außerhalb des Baumes in der JSP-Seite deklariert und dem TreeControl über das Attribut „**imagemap**“ zugeordnet.

```
<%@ taglib uri="/WEB-INF/tlds/cc-controls.tld" prefix="ctrl" %>
<%@ taglib uri="/WEB-INF/tlds/cc-utility.tld"  prefix="util" %>

<util:imagemap name="imap_products">
    <util:imagemapping
        rule="prodgroup.open"
        src="images/imgBoxOpen.gif"
        width="16" height="16"/>
    <util:imagemapping
        rule="prodgroup.closed"
        src="images/imgBoxClosed.gif"
        width="16" height="16"/>
    <util:imagemapping
        rule="product"
        src=" images/imgItem.gif"
        width="16" height="16"/>
</util:imagemap>

<ctrl:tree
    id="prodtree1"
    name="products"
    action="sample201/productBrowse"
    root="true"
    linesAtRoot="true"
    labelProperty="name"
    imageProperty="type"
    imagemap="imap_products"
    expandMode="multiple"
    groupselect="true"
    checkboxes="true"/>
```

Arbeitsweise:
Bei der Darstellung des Baumes liefert das Datenmodel über die Methode getType() jeweils einen Ausdruck zurück, der mit der ImageMap verglichen wird. Bei einer Übereinstimmung mit einer Regel wird das entsprechende Bild gezeichnet. Für geschlossene und geöffnete Knoten, werden diese Ausdrücke automatisch um das Suffix „.open“ bzw. „.closed“ erweitert, so dass sich unterschiedliche Bilder für die unterschiedlichen Zustände verwenden lassen. Die Angabe der Regeln erfolgt mit regulären Ausdrücken.

Die Methode, welche den Ausdruck für das zuzeichnende Bild liefert, wird über das Attribut „**imageProperty**“ bestimmt. In unserem Beispiel wird das type-Property verwendet, welches für Gruppen stets „prodgroup“ und für einzelne Blätter stets „product“ liefert.

1.10 Anhang: Alternative Designs



2 Glossar

C

CC Common-Controls