

**Integration eines Prozessmodells
in das WebPoint-Framework**

Studienarbeit von

Daniel Woithe 1004568

UniBwM-IS 51/2006

Aufgabenstellung und Betreuung:
Dr. Lothar Schmitz

Universität der Bundeswehr München
Fakultät für Informatik
Neubiberg, 31.10.2006

Inhaltsverzeichnis

| | | |
|----------|--|------------|
| 1 | Einleitung | 7 |
| 1.1 | Motivation | 7 |
| 1.2 | Aufgabenstellung | 11 |
| 1.3 | Gliederung der Arbeit | 11 |
| 2 | WebPoint | 13 |
| 2.1 | Entstehungsgeschichte | 13 |
| 2.2 | Konzepte | 13 |
| 2.2.1 | Shopkonzept..... | 14 |
| 2.2.2 | Prozesskonzept | 14 |
| 3 | Technischer Überblick | 15 |
| 3.1 | HTTP | 15 |
| 3.2 | Servlet | 16 |
| 3.3 | Struts | 17 |
| 3.3.1 | Ablauf einer Abfrage | 17 |
| 3.4 | JDOM | 18 |
| 4 | Umsetzung | 19 |
| 4.1 | Anpassung des RequestProcessor | 19 |
| 4.1.1 | Garbage-Collection | 19 |
| 4.2 | Darstellung des DEA | 20 |
| 4.3 | Validierung der Prozessübergänge | 21 |
| 5 | Integration in WebPoint | 25 |
| 5.1 | Darstellung des veränderten Arbeitsablaufs | 25 |
| 5.2 | Generierung der struts-config.xml | 25 |
| 6 | Ausblick | 29 |
| A | Tutorial | 31 |
| B | Quellcode | 115 |
| | Literaturverzeichnis | 125 |

Bestätigung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken und Zitate sind als solche kenntlich gemacht. Es wurden keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit wurde weder einer anderen Prüfungsbehörde vorgelegt noch veröffentlicht.

Neubiberg, den 31.10.2006

1 Einleitung

In dieser Arbeit geht es um die Erweiterung des bestehenden *WebPoint*-Frameworks, in das ein Prozessmodell integriert werden soll.

1.1 Motivation

Zwischen Java- und Web-Anwendungen gibt es große Unterschiede, nicht nur im Bereich der Oberfläche sondern vor allem unter sicherheitstechnischem Aspekt. In einem Java-Programm mit GUI-Oberfläche sind nur die vorgegebenen Schaltflächen und Eingabemasken verfügbar, der Nutzer kann somit nur vorhersehbare und nachvollziehbare Aktionen ausführen. Bei einer Web-Anwendung in einem beliebigen Browser hingegen kann der Benutzer beliebige Links in der Adressleiste eingeben oder den Web-Browser schließen, ohne dass die Anwendung dies registriert.

Für beide Anwendungsgebiete gibt es zahlreiche Frameworks, wobei wir an dieser Stelle die im Programmierpraktikum verwendeten Frameworks *SalesPoint* als Java- und *WebPoint* als webbasierte Umsetzung betrachten wollen. *WebPoint* wurde auf *SalesPoint* aufbauend entwickelt, somit bestehen die Unterschiede nur im webtechnischen Bereich.

Dass beim Schließen des Browsers keine Daten verloren gehen oder Inkonsistenzen auftreten, wird durch die Invalidierung der Session nach einer festgelegten Zeit geregelt. Dadurch werden alle getätigten und nicht vollständig abgeschlossenen Aktionen rückgängig gemacht und Bestände aus Warenkörben in den Ursprungsbestand verschoben. Jedoch gibt es für die manuelle Linkeingabe noch keine Lösung. Bisher prüft *WebPoint* nur, ob der Nutzer die korrekten Rechte besitzt, um eine Seite aufzurufen oder eine Aktion durchzuführen. Hat er aber die geforderten Rechte, kann er durch die Eingabe eines bestimmten Links an einen anderen Punkt der Anwendung springen, obwohl dies vom Geschäftsablauf nicht vorgesehen war und dadurch zu Datenverlusten oder Inkonsistenzen führen kann.

Dieses Problem soll durch ein konkretes Beispiel an dem bisherigen Videoautomaten (*WebPoint*-Tutorial) verdeutlicht werden. *Customer1* will zwei Videos ausleihen, wählt diese aus und bestätigt die Eingabe mit „Rent“ (Abb. 1).

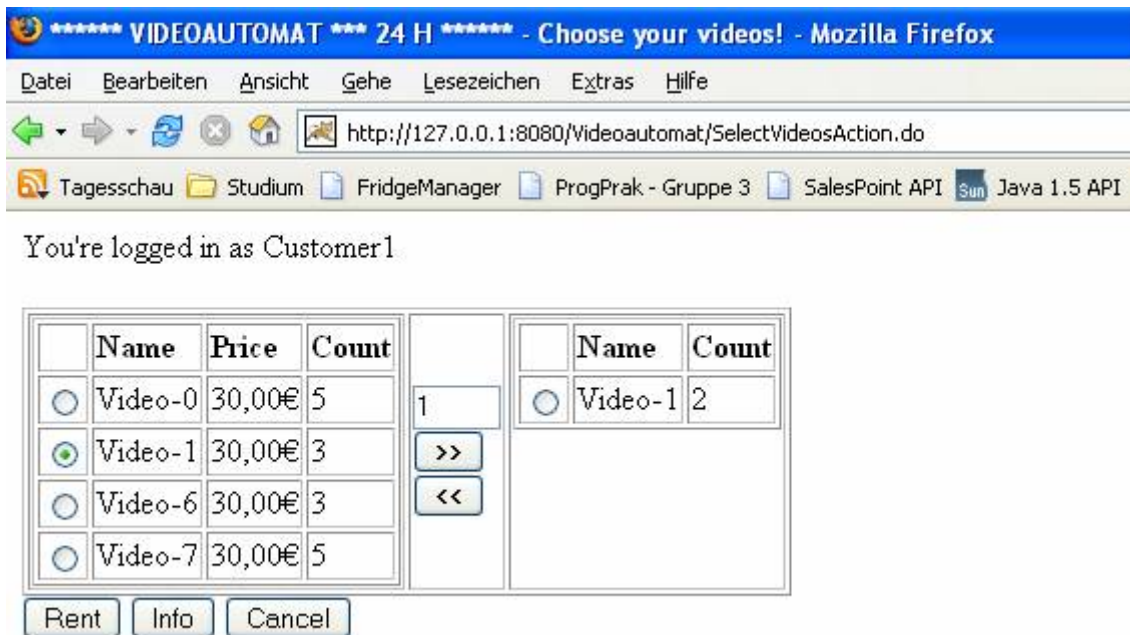


Abbildung 1: Auswahl von zwei Videos im Ausleihvorgang

Die Videos liegen nun in seinem persönlichen Warenkorb. Beim Bezahlvorgang wählt er den 200-Euro-Schein und legt ihn in den Warenkorb. Nun schließt er den Ausleihvorgang jedoch nicht wie vom Entwickler vorgesehen mit dem „Pay“-Button ab oder geht mit dem „Cancel“-Button zurück, sondern tippt, wie in der Adressleiste markiert, manuell den Link zur Startseite ein (Abb. 2).

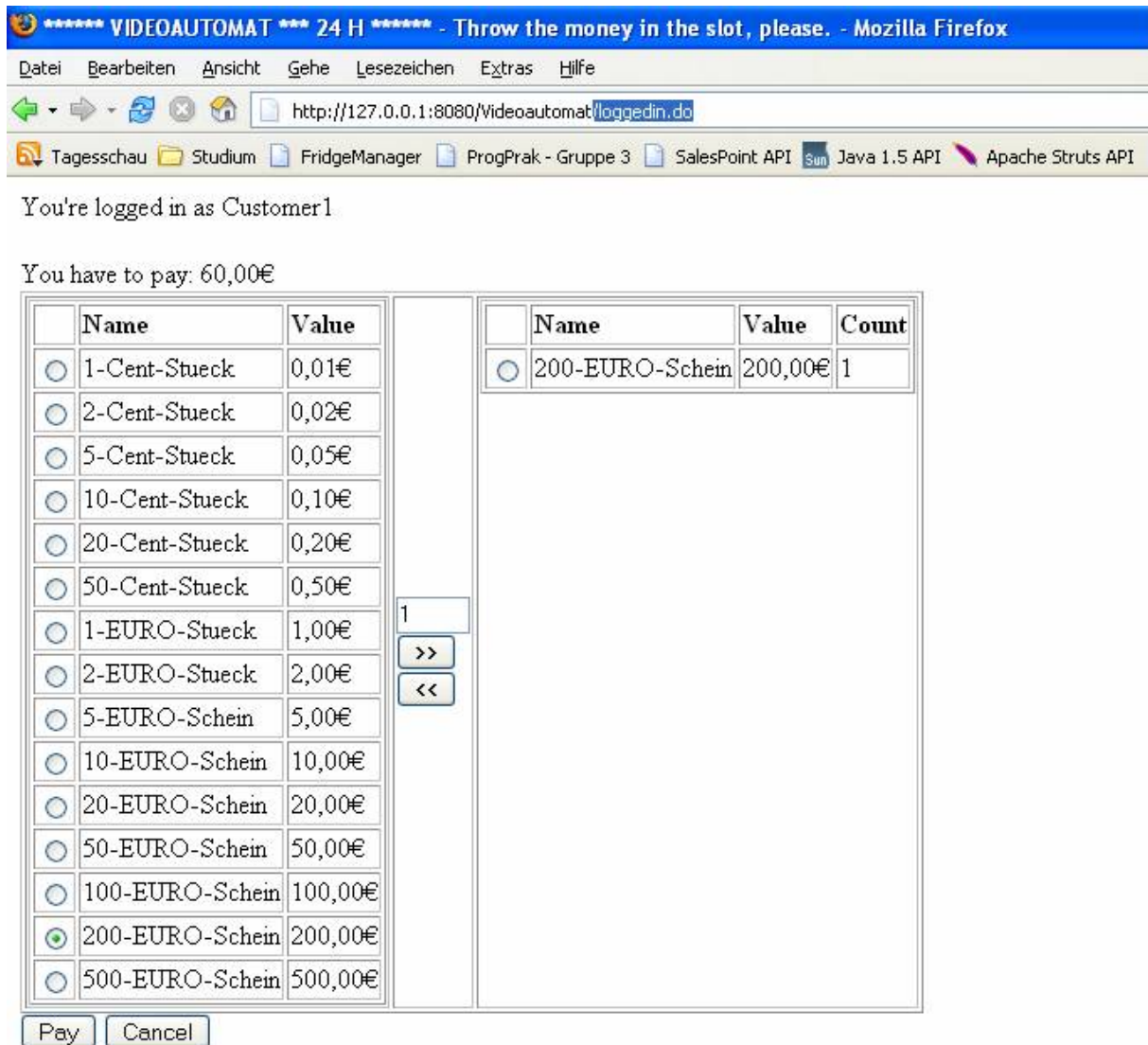


Abbildung 2: Bezahlvorgang mit manueller Linkeingabe

Geht er nun abermals in den Ausleihvorgang hinein, ergibt sich folgendes Bild: In seinem Warenkorb befinden sich zwei unterschiedliche Datentypen, weil keine der beiden eigentlich vorgesehenen Actions ausgeführt wurde und somit die Daten unbearbeitet im Speicher blieben (Abb. 3). Als Ergebnis beim Fortfahren des Vorgangs wird eine Exception geworfen, die daraus entsteht, dass der Entwickler nicht mit den hier aufgetretenen Daten gerechnet hat (Abb. 4). Bei einem Ablauf des Programms nach dem gedachten Prozessmodell wäre es zu diesem Problem nie gekommen.

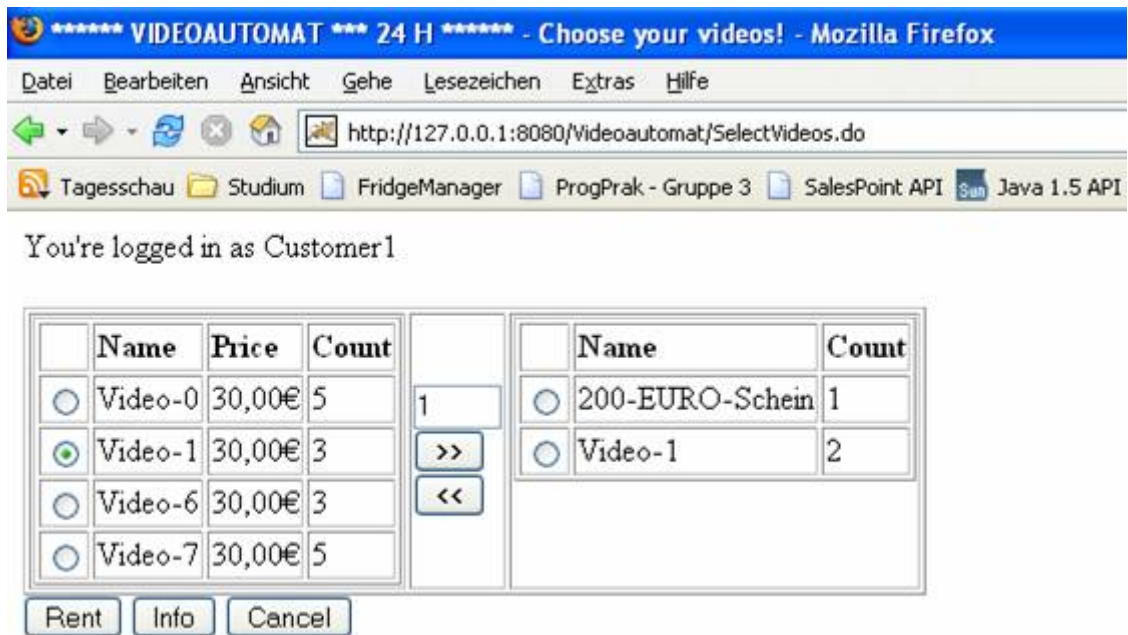


Abbildung 3: Ausleihvorgang mit ungültigen Daten im Warenkorb

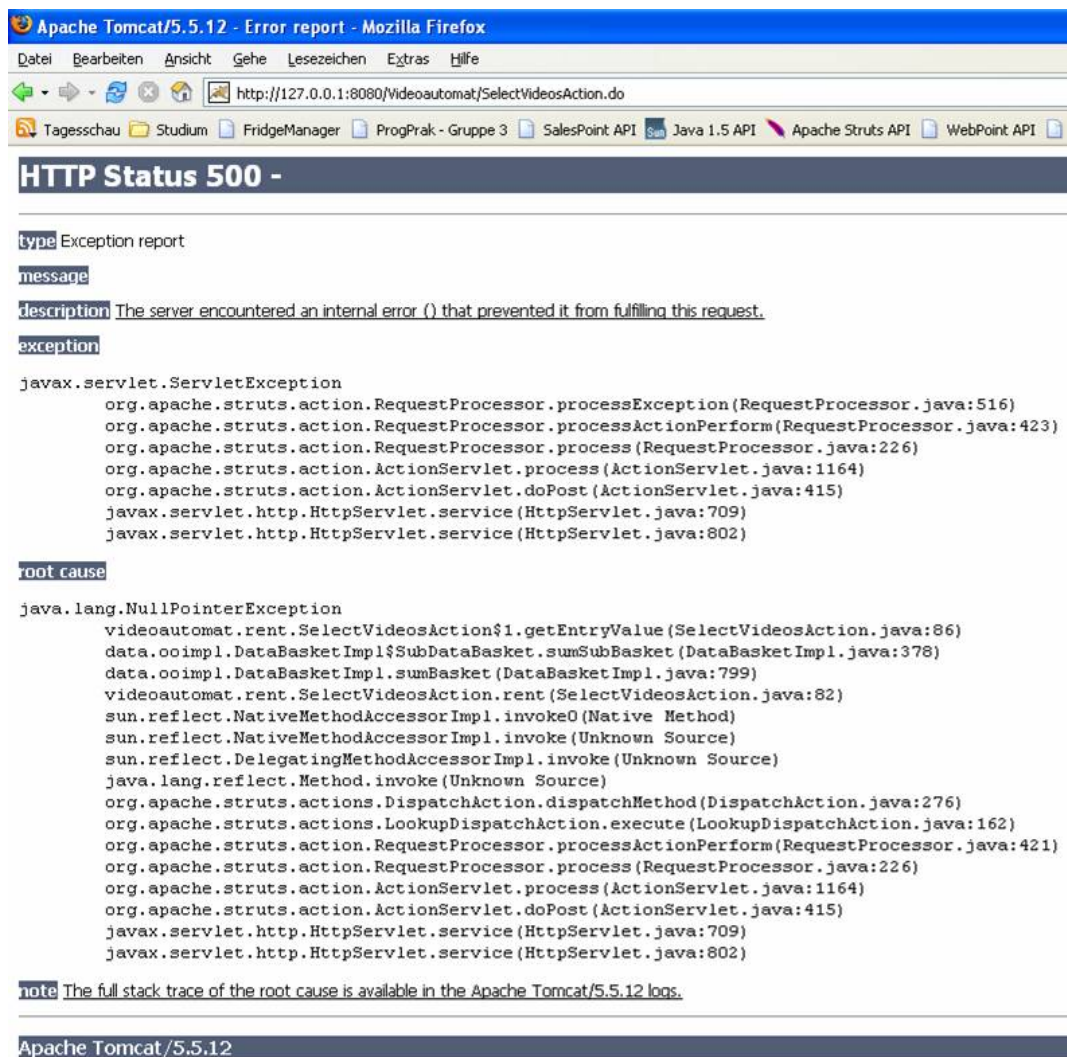


Abbildung 4: Fehlermeldung beim Aufruf der SelectVideosAction

1.2 Aufgabenstellung

Um diesen Fehler zu beheben, ist es die Aufgabe ein Prozessmodell in *WebPoint* zu integrieren. Der Entwickler soll dadurch gezwungen sein, alle möglichen Übergänge zwischen Zuständen vorher in Form eines deterministischen endlichen Automaten (DEA) zu definieren. Damit ist es für den Benutzer unmöglich, die festgelegten Transitionen durch manuelle Eingaben zu verlassen oder zu umgehen.

Die Aufgabe besteht aus drei Teilen:

- Eingabemöglichkeit zur Definition der Zustandsübergänge verbunden mit einer Arbeitserleichterung für den Entwickler
- Überprüfung der Zustandsübergänge durch das Framework
- Anpassung des Tutorials für das Programmierpraktikum an den veränderten Entwicklungsablauf

1.3 Gliederung der Arbeit

Die Arbeit ist folgendermaßen gegliedert: Der nächste Abschnitt befasst sich mit einer Einführung in das *WebPoint*-Framework und behandelt vor allem die Grundlagen, die für die Weiterentwicklung gemäß der obigen Aufgabenstellung notwendig sind. Im Folgenden wird auf die verwendeten Technologien, mit Schwerpunkt auf den direkt verwendeten Komponenten, eingegangen, wie z.B. das Struts-Framework. Danach folgt die Umsetzung der Aufgabenstellung. Dabei wird die Kombination der verschiedenen Ideen mit den vorhandenen Technologien erläutert, also wie die Definition des DEA und die Validierung der Zustandsübergänge realisiert wird. Im nächsten Kapitel wird die Integration in das *WebPoint*-Framework und die veränderten Arbeitsabläufe erklärt. Am Beispiel des Videoautomaten wird abschließend ein kompletter Entwicklungsablauf an einer einfachen Verkaufsanwendung erklärt. Dieses Tutorial baut auf die *SalesPoint*- und *WebPoint*-Tutorials auf und dient dem Entwickler als Einarbeitung in das Framework.

2 WebPoint

Dieses Kapitel beschäftigt sich mit einer kurzen Einführung in das *WebPoint*-Framework. Dabei werden vor allem die Konzepte vorgestellt, welche für die in dieser Arbeit dargestellte Weiterentwicklung des Frameworks eine bedeutende Rolle spielen.

2.1 Entstehungsgeschichte

Das *WebPoint*-Framework wurde 2005 von Conrad Reisch und Torsten Walter im Rahmen einer Studienarbeit an der Universität der Bundeswehr München entwickelt [RW05] und im Herbsttrimester 2005 zum ersten Mal bei einem Programmierpraktikum eingesetzt. Es ist eine Erweiterung des im Jahre 1997 an der TU Dresden entwickelten *Salespoint*-Framework. [Zsc00] Ziel dessen war es, den Studenten ein einfaches aber auch mächtiges Werkzeug zur Verfügung zu stellen, um Verkaufsanwendungen in einem relativ kurzen Zeitraum selbstständig programmieren zu können. Da in *Salespoint* geschriebene Anwendungen jedoch auf nur einem Rechner laufen konnten und auch die Datenverwaltung nicht über eine Datenbankbindung realisiert wurde, konnten diese Programme nicht in der Praxis verwendet werden.

Aus diesem Grund kam es zu der Entstehung des *WebPoint*-Frameworks. Dieses entstand durch Integration einer Webschnittstelle, wodurch die Anwendung nun gleichzeitig auf mehreren Rechnern ausgeführt werden kann. Die Datenverwaltung wurde vollständig aus *Salespoint* übernommen, da die Studenten zu diesem Studienzeitpunkt noch nicht über die nötigen Datenbankkenntnisse verfügen und somit die Komplexität steigen würde. Zusammenfassend bedeutet dies, dass mit dem *WebPoint*-Framework zwar eine vollständig laufende Webshop-Anwendung entwickelt werden kann, sie jedoch aufgrund der Datenhaltung nicht für den kommerziellen Betrieb mit mehreren Tausend Zugriffen am Tag ausgelegt ist.

2.2 Konzepte

Da die *WebPoint*-Konzepte für das Gesamtverständnis des Frameworks eine große Rolle spielen, jedoch zu [RW05] identisch sind, werden sie aus dieser Studienarbeit unverändert übernommen. Auf eine Kenntlichmachung der Zitate wird aus Gründen der Lesbarkeit bewusst verzichtet.

2.2.1 Shopkonzept

Jede Anwendung, die mit Hilfe von *WebPoint* erstellt wird, stellt einen Laden (*Shop*) dar. Dieser *Shop* ist nach dem Singleton-Muster implementiert und innerhalb der Anwendung einzigartig. Es gibt einen globalen *Shop*, der beim Start der Anwendung initialisiert wird, in dem alle Daten der Anwendung gespeichert werden. Wenn ein Kunde diesen *Shop* betritt, bekommt er einen Einkaufskorb (*DataBasket*), mit dem er sich um Laden umschauchen kann. Innerhalb des Ladens kann es einen oder mehrere Verkaufsstände geben. Wenn der Kunde an einem der Verkaufsstände etwas kaufen will, beginnt ein Verkaufsprozess. An einem Verkaufsstand können aber mehrere Kunden gleichzeitig einkaufen. Daher kann es prinzipiell vorkommen, dass zwei Kunden um den gleichen Artikel konkurrieren. Dabei gilt wie so oft: „Wer zuerst kommt, mahlt zuerst.“ Der Kunde, der als zweiter einen Artikel kaufen will, der dann schon vergriffen ist, hat in diesem Fall das Nachsehen.

2.2.2 Prozesskonzept

Jeder Kunde, der an einem Verkaufsstand einkaufen will, startet damit einen Verkaufsprozess. Ein solcher Prozess stellt im Prinzip einen endlichen Automaten mit Zuständen und Zustandsübergängen dar. Die Zustände des Automaten werden in Form von *JavaServerPages* dargestellt. Nur an diesen definierten Zuständen kann eine Kommunikation mit dem Kunden stattfinden. Die Zustandsübergänge werden durch Links dargestellt. Veränderungen an den Daten der Anwendung können nur innerhalb eines Zustandsüberganges stattfinden. Nur an einem solchen Übergang wird mit dem Webserver Verbindung aufgenommen, weshalb auch nur in diesem Moment eine Änderung der serverseitig gespeicherten Daten stattfinden kann. Wird ein Speichervorgang der Anwendung ausgelöst, werden automatisch alle Einkaufskörbe, die in den Sitzungsdaten der Kunden abgespeichert sind, geleert. Erst danach wird ein Abbild der Anwendung erstellt. Ansonsten könnte es zu Inkonsistenzen kommen, da Waren, die ein Kunde in seinen Einkaufskorb getan hat, nicht mehr im Laden vorhanden sind und daher verloren gehen würden. Zumindest kann man anhand eines solchen Automaten seine Anwendung entwerfen. Denn im Widerspruch zu einem richtigen endlichen Automaten kann der Kunde seinen Browser jederzeit schließen. Damit wird der Verkaufsprozess auf ungewollte Weise verlassen. Wenn ein Kunde seinen Browser schließen sollte, während er sich in einem Verkaufsprozess befindet, läuft diese Sitzung (*Session*) nach einer gewissen Zeit aus. Nach dem Ablauf einer Sitzung wird der dort gespeicherte Einkaufskorb ebenfalls geleert.

3 Technischer Überblick

Dieses Kapitel dient der Einführung in die wichtigsten verwendeten Technologien, die vor allem für die Erweiterung des *WebPoint*-Frameworks von Bedeutung sind. Beginnend bei den Grundlagen wird Schritt für Schritt bis zum Ablauf einer Anfrage (siehe 3.3.1) hingearbeitet, worauf in den folgenden Kapiteln zurückgegriffen wird. Für die allgemeineren Grundlagen des *WebPoint*-Frameworks wird auf [RW05] verwiesen.

3.1 HTTP

Das *Hypertext Transfer Protocol* (HTTP) ist ein Protokoll der Anwendungsschicht zur Übertragung von Daten zwischen Computern. Es benötigt eine zuverlässige Übertragung der Daten und verwendet deshalb nahezu in allen Fällen TCP/IP. HTTP ist ein zustandsloses Protokoll, das heißt, die Verbindung zwischen zwei Teilnehmern wird nicht über die komplette Kommunikation aufrechterhalten, sondern in Anfragen und Antworten unterteilt. Im Internet findet dieses Vorgehen fast ausschließlich bei Client-Server-Architekturen Anwendung. Der Client sendet eine Anfrage (*Request*) an den Server, worauf dieser mit einem *Response* antwortet. Die Abarbeitung der Anfragen beim Server erfolgt nach dem FCFS-Prinzip.

Für das Senden von statischen Inhalten ist HTTP sehr effizient und deshalb auch so weit verbreitet. Allerdings besteht ein entscheidender Nachteil in der Erstellung dynamischer Inhalte, wenn die Antwort zum Beispiel vom anfragenden Benutzer abhängig sein soll. Durch zusätzliche Informationen im HTTP-Header kann diesem Problem begegnet werden. So können zum Beispiel Angaben über den Browser oder zur gewünschten Sprache mit übergeben werden. Außerdem können in Cookies Informationen gespeichert werden, für obiges Problem zum Beispiel der Name des anfragenden Benutzers. Zusätzlich enthält der HTTP-Header die Methode der Anfrage, wobei GET und POST die gebräuchlichsten sind. Die Antwort des Servers erfolgt mit Hilfe von Seitenbeschreibungssprache wie (X)HTML und ihren Ergänzungen, die vom Webbrowser interpretiert und dargestellt werden. Die (*Extensible*) *HyperText Markup Language* ist eine textbasierte Auszeichnungssprache, die die Darstellung von Texten, Bildern und Hyperlinks im Browser regelt. Da diese Seiten statisch sind, braucht man eine Möglichkeit HTML dynamisch zu erzeugen. Bei der Verwendung von Java gibt es dafür *Servlets*.

3.2 Servlet

Servlets sind Java-Klassen, deren Instanzen innerhalb eines *Servlet-Containers* laufen und Anfragen von Clients entgegennehmen und beantworten. Dabei kann der Inhalt der Antworten dynamisch erstellt werden, da bei der Verarbeitung der Anfrage beliebiger Java-Code ausgeführt werden kann. Somit kann zum Beispiel die Anbindung an eine Datenbank oder die Anzeige personalisierter Seiten gewährleistet werden. Neben den Vorteilen von Java, die hierauf vererbt werden, haben *Servlets* gegenüber der konventionellen CGI-Programmierung (Common Gateway Interface) weitere Vorteile: Ein *Servlet* startet nicht für jeden *Request* einen extra Prozess, was den Overhead deutlich reduziert. Außerdem existiert nur eine einzige Instanz, die alle *Requests* beantwortet. Dies spart Speicherplatz und erleichtert die persistente Haltung der Daten. Auch die nebenläufige Verarbeitung von Anfragen ist möglich, jedoch muss deshalb unbedingt auf die threadsichere Programmierung der *Servlets* geachtet werden.

Alle *Servlets* müssen das Interface `javax.servlet.Servlet` oder davon abgeleitete Klassen implementieren. Die abstrakten Klassen `javax.servlet.GenericServlet` und `javax.servlet.HttpServlet` implementieren bereits die wichtigsten Methoden des Interfaces und stellen zusätzlich die Methoden `doGet()` und `doPost()` zur Verarbeitung von HTTP-Requests zur Verfügung.

Folgendes Beispiel zeigt wie damit ein einfaches *Servlet* erstellt werden kann:

```
public class HelloUserServlet extends HttpServlet {  
    protected void doGet(HttpServletRequest request,  
        HttpServletResponse response)  
        throws ServletException, IOException {  
  
        String user = request.getUsername(); //Methode vorausgesetzt  
        response.setContentType("text/html");  
        PrintWriter out = response.getWriter();  
        out.println("<HTML><HEAD><TITLE>Welcome</TITLE></HEAD>");  
        out.println("<BODY>Hello " + user + "!</BODY></HTML>");  
        out.close();  
    }  
  
    protected void doPost(HttpServletRequest request,  
        HttpServletResponse response)  
        throws ServletException, IOException {  
  
        doGet(request,response); //Arbeit an doGet weiterreichen  
    }  
}
```

In diesem Beispiel wird anschaulich gezeigt, dass der Seiteninhalt von dem aufrufenden Benutzer abhängig ist. Dessen Name wird bei jeder Ausführung ausgelesen und dann mittels `out.println()` in HTML-Formatierung ausgegeben. Ebenfalls ist erkennbar, dass es ausreicht, `doGet()` und `doPost()` zu implementieren, alle anderen Methoden werden vom `HttpServletRequest` geerbt. Da es keine Rolle spielt, welche von beiden Methoden den *Request* bearbeitet, wird die Arbeit von `doPost()` an `doGet()` weitergereicht. In welcher Richtung dies geschieht ist irrelevant.

3.3 Struts

Struts ist ein Open-Source-Projekt für die Entwicklung von Java-Webanwendungen, welches das MVC2-Muster umsetzt. Die Model-2-Architektur sorgt für eine strikte Trennung von Model, View und Controller zur besseren Übersichtlichkeit und Wartbarkeit. Die drei Hauptkomponenten zur Umsetzung dieser Trennung sind:

- JSP als *View*: zur Laufzeit Umwandlung in ein Servlet und Kompilieren.
- Action als *Controller*: verantwortlich für Umsetzung der Geschäftslogik, Schnittstelle zum View
- FormBean als *Model*: Speicherung und Validierung der Daten

Das Zusammenspiel dieser Komponenten wird im folgenden Kapitel erklärt.

3.3.1 Ablauf einer Abfrage

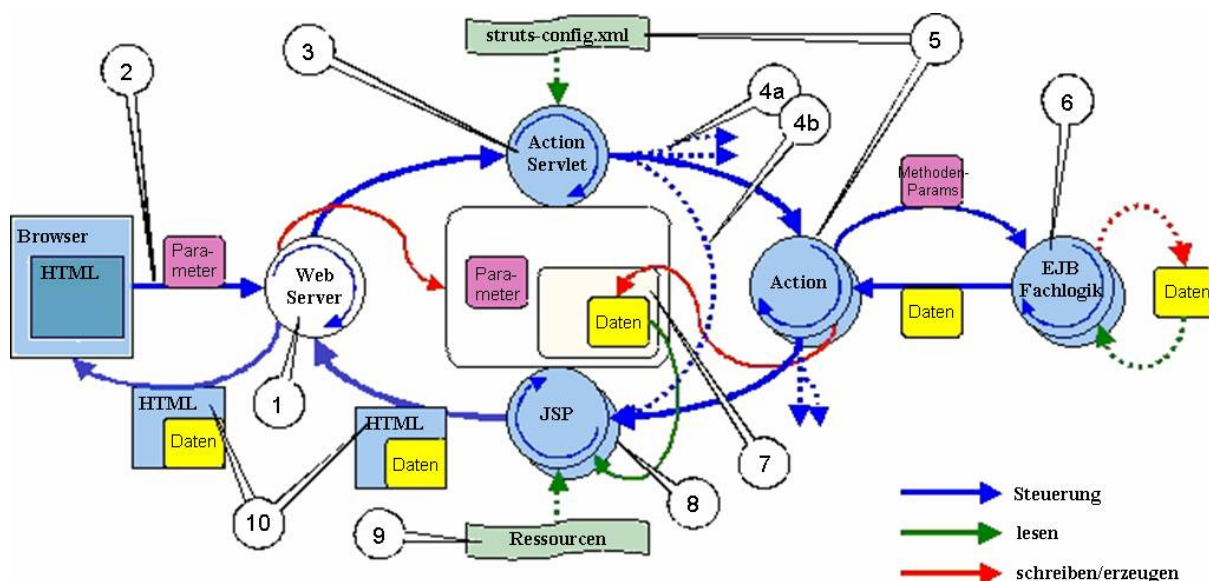


Abbildung 5: Programmablauf innerhalb einer Struts-Anwendung [Sch]

In Abbildung 5 ist der Gesamtablauf einer Struts-Anwendung für das Aufrufen einer Seite gezeigt. Nach der Initialisierung der Struts-Komponenten mit Hilfe der `web.xml` (1) kann der Benutzer eine Anfrage (HTTP-Request) an den Web-Server stellen (2). Diese Anfrage wird vom Controller entgegengenommen (3). Er entscheidet, je nachdem ob die Geschäftslogik ausgeführt werden muss oder direkt eine Ausgabe erfolgen kann, ob er die Anfrage an die Klasse `ActionMapping` übergibt (4a), oder ob sie gleich an die JSP weitergeleitet werden kann (4b). Wenn durch das Mapping in der `struts-config.xml` eine entsprechende *Action* gefunden wird (5), leitet diese Klasse die Anfrage an die zugehörige *JavaBean* weiter (6). Nach der Ausführung der Geschäftslogik wird das Ergebnis als *JavaBean* (7) an die JSP übergeben (8). Diese liest aus externen Ressourcen (wie z.B. dem Properties-File der entsprechenden Sprache oder Tag-Libraries) die verlinkten Daten aus (9), generiert daraus die Seite mit den Ergebnisdaten und sendet diese an den Web-Browser zurück (10).

3.4 JDOM

JDOM steht für *Java Document Object Model* und ist eine speziell für Java optimierte API. Es vereint die Vorteile und APIs von SAX und DOM in einer Java-Klasse, welches ihrerseits eigenständige Standards zur Bearbeitung von XML-Dokumenten sind. Jedoch haben beide Nachteile, zum einen den nicht-objektorientierten Ansatz und zum anderen die teils große Prozessor- und Speicherbelastung, welche vor allem für Web-Anwendungen kontraproduktiv ist.

JDOM bietet die Möglichkeit, ein XML-Dokument mittels des `SAXBuilder` in ein `JDOM Document` umzuwandeln. Ein `Document` ist das Wurzelement des JDOM-Baumes, der beliebig viele `Elements` enthält. Diese Struktur spiegelt 1:1 die der ursprünglichen XML-Datei wider. `Document` und `Element` sind die wichtigsten Klassen bei der Arbeit mit JDOM. Dieses erzeugte `JDOM Document` kann nun beliebig bearbeitet und dann wieder als XML-Datei ausgegeben werden. Dafür wird der `XMLOutputter` genutzt. Aufgrund der Vereinigung von SAX und DOM gibt es auch die Möglichkeit, das `Document` mittels des `SAXOutputter` oder des `DOMOutputter` auszugeben.

Zur Validierung der Prozessübergänge (siehe 4.3) und Erzeugung der `struts-config.xml` (siehe 5.2) wurde der hier beschriebene Standardweg benutzt: Einlesen einer XML-Datei mit dem `SAXBuilder`, Bearbeiten des erzeugten `Documents` und Ausgabe mit dem `XMLOutputter` wiederum in einer XML-Datei.

4 Umsetzung

4.1 Anpassung des RequestProcessor

Der `RequestProcessor` stellt die einzige Instanz dar, in den Ablauf von einem gesendeten `Request` bis zu einem empfangenen `Response` einzugreifen. Diese Klasse ist daher die einzige Möglichkeit zu entscheiden, ob die Anfrage an den Server auch mit dem definierten Prozessmodell übereinstimmt. Der für solche Anpassungen von Struts definierte Hook ist die Methode `processPreprocess`, die bei jedem Aufruf ausgeführt wird und standardmäßig nur `true` zurückliefert.

Diese Methode wurde nun so überschrieben, dass sie die Gültigkeit der Transition folgendermaßen überprüft: Aus der `transitions.xml` werden alle möglichen Folgezustände herausgelesen, wonach überprüft wird, ob die angeforderte Seite mit einem dieser Zustände übereinstimmt. (im Detail, siehe Kap. 4.3.) Das einzige Problem stellt jedoch das Herausfinden der Seite dar, von welcher aus der `Request` gesendet wurde. Dies ist sowohl über die `Session` als auch über andere bei der Anfrage verwendete Klassen nicht möglich. Deswegen wird in einem privaten Klassenattribut (hier: `HistoryMap`) jeweils die letzte gültige besuchte Seite pro `SessionID` gespeichert. So kann durch die `SessionID` eindeutig die Seite zugeordnet werden, von der an sich die Anfrage gestellt wurde.

Die einzige Möglichkeit, diesen Vorgang zu umgehen, ist es, wenn ein angemeldeter Benutzer während eines Bestellprozesses durch Eintippen eines externen Links die Anwendung verlässt und wiederum durch Eingabe einer gültigen URL innerhalb der Gültigkeit der `Session` zur Anwendung zurückkehren will. Dann geht der `RequestProcessor` davon aus, dass die Anfrage von der zuletzt innerhalb der Anwendung besuchten Seite kommt. Das ist zwar an sich falsch, jedoch kann dies keine Fehler verursachen, da nur die gültigen Zustände entsprechend dem Prozessmodell zugelassen werden. Somit kann es im Gegensatz zu vorher zu keinen Dateninkonsistenzen innerhalb der Anwendung kommen.

4.1.1 Garbage-Collection

Der Begriff wurde aufgrund der Ähnlichkeit zu dem Java-internen Garbage-Collector übernommen. Da zu jeder `SessionID` in dem privaten Klassenattribut die zuletzt besuchte Seite gespeichert wird, muss man dafür sorgen, dass die dabei verwendete `Map` nicht nach längerer Verwendungszeit der Anwendung irgendwann „überläuft“. Beim Invalidieren einer

Session innerhalb der Anwendung wird dies direkt vom `RequestProcessor` bemerkt und somit der Eintrag direkt gelöscht. Wird eine Session jedoch aus anderen Gründen ungültig, z.B. ein Session-Time-Out beim Verlassen der Website, kann dies nicht nachvollzogen werden. Aus diesem Grund wird zusätzlich die `LastAccessedTime` zu jeder `SessionID` gespeichert. Der Garbage-Collector überprüft in vordefinierten Intervallen mit Hilfe dieser Zeit, ob eine Session aus der Map abgelaufen ist und entfernt sie dann sofort.

4.2 Darstellung des DEA

Um zu gewährleisten, dass nur die Transitionen ausgeführt werden, die gemäß den entworfenen Zustandsübergangsdigrammen definiert sind, müssen diese grafischen Definitionen durch den Entwickler 1:1 in die `transitions.xml` umgesetzt werden. Diese darf beliebig viele `<transition>` Tags enthalten, das heißt die Strukturierung der Datei ist vollkommen dem Entwickler überlassen. Es bietet sich allerdings an, diese genau anhand der Aufteilung zwischen den Zustandsübergangsdigrammen zu übernehmen.

Anhand eines Beispiels wird verdeutlicht, was mit einer 1:1 Umsetzung gemeint ist:

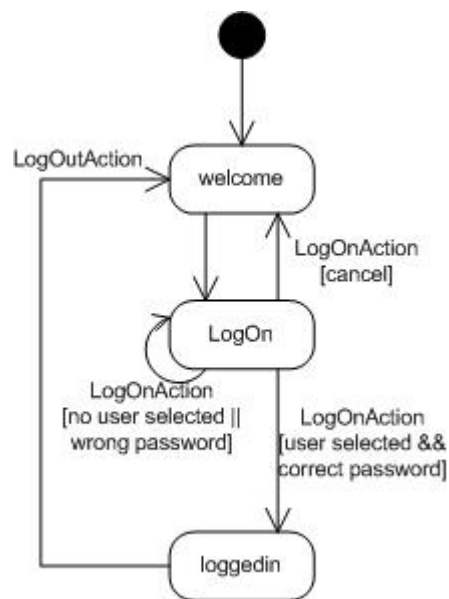


Abbildung 6: Zustandsübergangsdigramm eines Anmeldeprozesses

Dieses UML Zustandsübergangsdigramm wird in folgenden XML-Code umgesetzt:

```
<transition>
  <page>/welcome</page>
  <page>/LogOn</page>
  <page action="/LogOnAction" input="/LogOn">
    <forward>/welcome</forward>
  </page>
  <page>/loggedin</page>
  <page action="/Logout"/>
  <page>/welcome</page>
</transition>
```

Jeder Zustand wird als `<page>` dargestellt, jeder Übergang als eine *Action* interpretiert und deshalb als `<page action="***"/>` übersetzt. Da es von einem Zustand mehrere Übergänge geben kann, gibt es die folgenden drei Möglichkeiten, den nächsten erlaubten Zustand anzugeben:

- im `input`-Attribut, was die Weiterleitung bei einem `getInputForward()`-Aufruf in der zugehörigen Action darstellt,
- mit beliebig vielen `<forward>` Tags innerhalb eines `<page action=...>` Tags,
- mit dem `<page>` Tag, der nach dem `<page action=...>` Tag folgt. (Dies sollte die Standardfortsetzung des Prozesses sein, wenn eine solche existiert.)

Bei Anwendung der dritten Möglichkeit sollte darauf geachtet werden, dass dieser Zustand nicht schon in den vorhergehenden `<forward>` Tags mit auftaucht, da sie diesen gleichwertig ersetzt und sonst zu einer doppelten Aufführung bei der Generierung der `struts-config.xml` führen würde.

4.3 Validierung der Prozessübergänge

Wie schon in Kap. 4.1 beschrieben, besteht der Ansatz aus dem Überschreiben der Hook-Methode `processPreprocess` der Klasse `web.sale.RequestProcessor`. Diese ist ihrerseits eine Erweiterung der Klasse `org.apache.struts.action.RequestProcessor` um die Nutzer-Authentifizierung und wurde schon durch die erste *WebPoint*-Version bereitgestellt. [RW05]

Einen wesentlichen Bestandteil der neuen Subklasse `CustomRequestProcessor` bildet die private Methode

```
private ArrayList<String> nextTransition(String prev, String path).
```

Sie liest mit Hilfe des SAX-Parsers aus der `transitions.xml` (path) alle zu `prev` definierten Folgetransitionen aus und gibt sie in einer `ArrayList` zurück.

Nun gibt es bei der Überprüfung einer Transition mehrere mögliche Ausgangspositionen: der Benutzer

1. besucht die Website zum ersten Mal (mit seiner zugewiesenen `SessionID`),
2. führt einen gültigen Link aus,
3. verlässt den vorgesehenen DEA (z.B. Abb. 6) durch erneutes Aufrufen der Startseite,
4. ruft durch Eintippen in der Adresszeile eine ungültige Transition innerhalb der Anwendung auf.

Bei der ersten Möglichkeit muss in der `HistoryMap` seine `SessionID` und die Startseite eingetragen und der Zugriff auf diese Seite zugelassen werden.

Tritt der zweite Fall ein, so wird überprüft, ob die angeforderte Seite in der `ArrayList` enthalten ist, die durch `nextTransition()` erzeugt wird. Ist dies der Fall, so wird der Eintrag in der `HistoryMap` aktualisiert und der Zugriff gestattet. Wenn nicht, kann es sich nur um Punkt 3 oder 4 handeln.

Beim erneuten Aufrufen der Startseite wird der Zugriff natürlich gestattet. Dabei wird die Session des Nutzers aber im gleichen Zuge invalidiert, d.h., es wird ein `rollback` auf allen Datenkörben ausgeführt und der User damit ausgeloggt.

Als letzter Fall bleibt nur noch der Aufruf eines ungültigen Links übrig. Hierbei wird die `HistoryMap` nicht aktualisiert und dem Benutzer eine Fehlermeldung (Abb. 7) angezeigt.

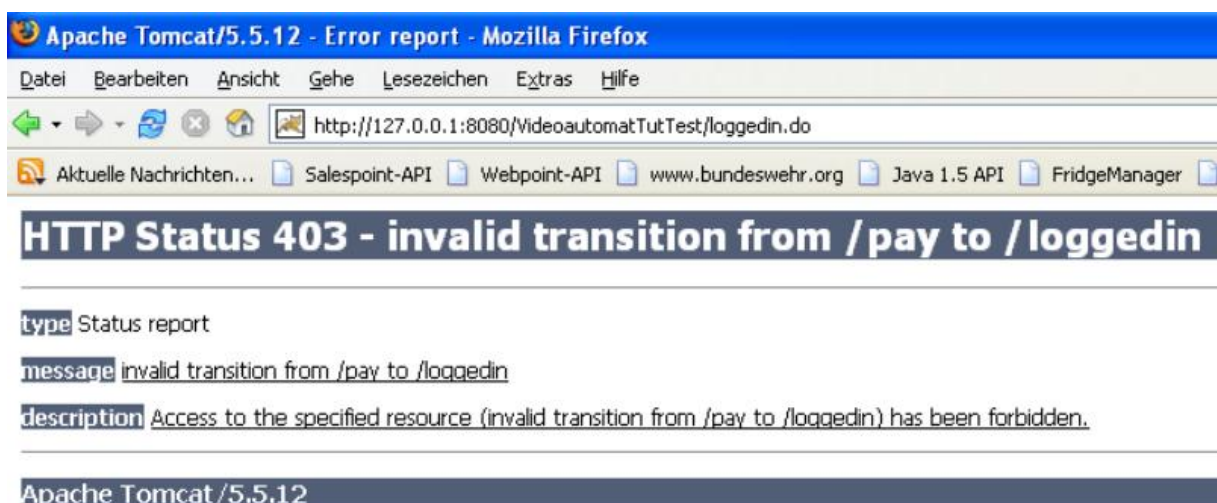


Abbildung 7: Fehlermeldung bei ungültiger Transition

Da diese Fehlermeldung durch das Framework fest vorgegeben ist, wäre hier ein Ansatzpunkt für eine Erweiterung. Dabei könnte dem Entwickler die Möglichkeit gegeben werden, zwischen verschiedenen Varianten der Reaktion bei ungültigen Links zu wählen, zum Beispiel Rückkehr zur Startseite mit gleichzeitiger Invalidierung der Session oder eine POST-Meldung wie auf vielen Internetseiten (Abb. 8).

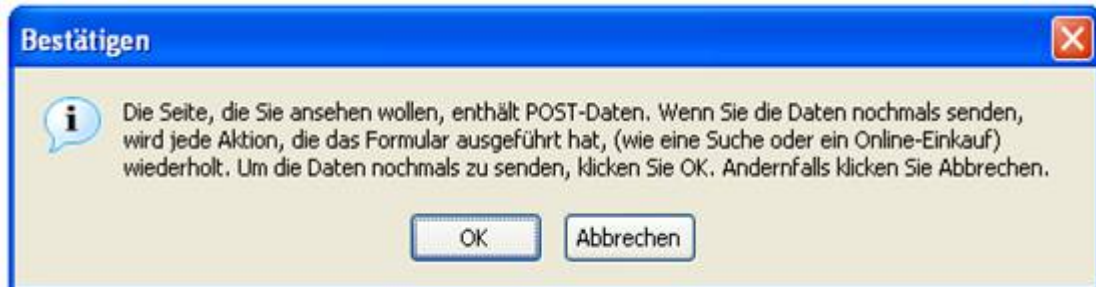


Abbildung 8: alternative Fehlermeldung für ungültige Transition

Integration in WebPoint

5.1 Darstellung des veränderten Arbeitsablaufs

Um die Veränderungen bei der Erstellung eines *WebPoint*-Projekts durch das Ergebnis dieser Studienarbeit darzustellen, wird zuerst kurz die alte Vorgehensweise beschrieben. Darauf folgend wird der neue Arbeitsablauf dargestellt mit Hinweis auf die schwerpunktmäßigen Veränderungen.

Nach dem alten Konzept waren drei Komponenten zur Erstellung einer Transition nötig:

- eine JSP, von der durch die Interaktion mit dem Benutzer die Aktion ausgeht,
- eine *Action*, in der alle auszuführenden Aktionen implementiert sind,
- die Konfiguration der *Action* (evtl. auch der zugehörigen *Form*) in der `struts-config.xml`.

Die Reihenfolge der Implementierung ist weitestgehend egal, jedoch müssen alle Verweise, mit denen diese Komponenten verbunden sind, genau übereinstimmen. Deshalb ist es empfehlenswert, die Konfiguration und damit Verbindung aller Komponenten in der `struts-config.xml` zum Abschluss eines Arbeitsabschnittes durchzuführen.

Nach der Integration des Prozessmodells in WebPoint kommt zu den oben genannten ein weiterer Arbeitsschritt hinzu: die Definition der Zustandsübergänge in der `transitions.xml` (siehe 4.2). Da sich diese direkt aus den Zustandsübergangsdiagrammen ergeben, sollte dieser Schritt als erstes durchgeführt werden. Danach kommt – unverändert zum alten Vorgehen – die Implementierung der *Action*, evtl. der *Form* und der JSP in beliebiger Reihenfolge, aber gleichzeitig unter strikter Beachtung der Bezeichner in der `transitions.xml`. Abschließend wird die `struts-config.xml` automatisch generiert und die markierten Attribute müssen durch den Entwickler entsprechend belegt werden. Mit diesem Punkt beschäftigt sich das nächste Kapitel ausführlicher.

5.2 Generierung der `struts-config.xml`

Die automatische Erzeugung der `struts-config.xml` aus der `transitions.xml` wird durch einen Ant-Task über die `build.xml` aufgerufen, der in der Klasse `web.util.ant.tasks.UpdateStrutsConfig` implementiert ist. Diese liest beide Dateien mit dem JDOM Parser ein (siehe 3.4). In der `transitions.xml` sind für diesen Vorgang ausschließlich alle Einträge interessant, die als *Action* gekennzeichnet sind:

```
<page>/LogOn</page>
<page action="/LogOnAction" input="/LogOn"/>
```

Die erste Zeile bleibt somit unberücksichtigt, dagegen wird der zweite Eintrag mit den aktuellen Einträgen in der `struts-config.xml` verglichen. Wenn dieser schon vorhanden ist, wird zum nächsten *Action*-Eintrag in der `transitions.xml` gesprungen und mit diesem fortgefahren. Gibt es allerdings Veränderungen, so wird der alte Eintrag in der `struts-config.xml` gelöscht und ein neuer mit den korrekten Daten erzeugt. Existiert noch gar keine Definition für eine solche *Action*, wird ebenfalls ein neuer Eintrag erstellt. Alle anderen Daten in der `struts-config.xml` bleiben unberührt bestehen, das heißt, sie werden in die neue Datei unverändert übernommen.

Ein aus obiger *Action*-Definition erzeugter Eintrag für die `struts-config.xml` sieht folgendermaßen aus:

```
<action path="/LogOnAction"
        type="***"
        name="***"
        scope="***"
        validate="***"
        input="/LogOn.do"
        parameter="***"
/>
```

Wie man sieht, können nur der Pfad und der `input-Forward` aus der `transitions.xml` übernommen werden, da sie nicht mehr Informationen enthält. Alle anderen Attribute, die mit `***` als Platzhalter gekennzeichnet sind, müssen durch den Entwickler manuell nach dieser Generierung befüllt oder entfernt werden.

Die Daten für die `struts-config.xml` werden während des gesamten Generierungsvorganges in einem `JDOM`-Baum zusammengesetzt und erst nach dem kompletten Durchlaufen durch den `XMLOutputter` in eine neue `struts-config.xml` geschrieben. Die alte wird davor als `struts-config.xml.old` ebenfalls im `WEB-INF` Ordner gesichert.

Alle durch den Entwickler vorgenommenen Formatierungen gehen bei der Erzeugung der neuen Datei verloren. Eine Möglichkeit, diese zu übernehmen und somit eine bessere Übersichtlichkeit in der `struts-config.xml` zu schaffen, gibt es leider nicht. Bei sehr großen Projekten können dafür jedoch auch Tools zur Bearbeitung und Konfiguration der Datei, wie z.B. *Struts Console*, verwendet werden. [SC]

6 Ausblick

Die mit dieser Studienarbeit erweiterte Version des WebPoint-Frameworks wird zum ersten Mal im Herbst 2006 eingesetzt und damit dem ersten richtigen Test unterzogen. Alle dabei erkannten Fehler werden im Nachhinein beseitigt. Vor allem das Feedback der Nutzer ist sehr wichtig, um Erkenntnisse über die Zweckmäßigkeit der Änderungen zu gewinnen, z.B., ob die automatische Generierung der `struts-config.xml` dem Entwickler wirklich Zeit spart und somit die Arbeit erleichtert.

Es existieren noch mehrere Ansatzpunkte, an denen sich eine Erweiterung des Frameworks anbietet: Zum Beispiel könnte geprüft werden, ob eine vollständige Generierung der `struts-config.xml` aus `JSP`, `Action`, `Form` und `transitions.xml` und im gleichen Zuge eine Konsistenzprüfung der Einträge zwischen diesen Dateien möglich ist. Außerdem wäre ein Eclipse-PlugIn denkbar, mit dem man Zustandsübergangsdiagramme zeichnen und daraus automatisch die `transitions.xml` erzeugen kann. Desweiteren könnte man die sehr strenge Validierung der Transitionen auf wirklich daten- und sicherheitsrelevante Übergänge lockern und eine Fehlermeldung wie in Abb. 8 einführen

Anlage A

WebPoint Tutorial

Inhaltsverzeichnis

| | | |
|----------|--------------------------------------|-----------|
| 1 | Tutorial | 35 |
| 1.1 | Aufgabenstellung | 35 |
| 1.2 | Überblick | 35 |
| 1.3 | Hinweise | 36 |
| 1.4 | Vorbereitungen für das erste Projekt | 36 |
| 1.4.1 | Tomcat installieren | 36 |
| 1.4.2 | Eclipse installieren | 37 |
| 1.5 | WebPoint Projekt in Eclipse anlegen | 38 |
| 1.6 | Grundlegende Funktion | 41 |
| 1.7 | Der Shop | 44 |
| 1.7.1 | Der Videokatalog | 45 |
| 1.7.2 | Der Videobestand | 48 |
| 1.8 | Den Videobestand anzeigen | 49 |
| 1.9 | Die Nutzerverwaltung | 52 |
| 1.9.1 | Der Usermanager | 53 |
| 1.9.2 | Der Automatenutzer | 53 |
| 1.10 | Die Anmeldung | 55 |
| 1.10.1 | logon.jsp | 57 |
| 1.10.2 | LogOnForm | 59 |
| 1.10.3 | LogOnAction | 62 |
| 1.10.4 | LogOut | 69 |
| 1.10.5 | Ein „kleiner“ Schönheitsfehler | 70 |
| 1.11 | Das Geld | 71 |
| 1.11.1 | Die Währung | 71 |
| 1.11.2 | Der Geldbeutel | 72 |
| 1.11.3 | Formatierung der Preise | 74 |
| 1.12 | Die Zeit | 75 |
| 1.12.1 | Definition eines Timers | 75 |
| 1.12.2 | Die Zeit weiterschalten | 76 |
| 1.13 | Die Leih-Kassette | 78 |
| 1.14 | Der Ausleihprozess | 80 |
| 1.14.1 | Select Videos | 82 |
| 1.14.2 | Pay | 91 |
| 1.14.3 | Confirm | 100 |
| 1.15 | Das Protokollieren | 103 |

| | |
|---|-----|
| 1.15.1 Log-Einträge | 103 |
| 1.15.2 Ein Zuhörer schreibt mit | 106 |
| 1.16 Das Protokoll ansehen | 108 |
| 1.17 Die fertige Anwendung | 113 |

1 Tutorial

Das Tutorial basiert auf dem SalesPoint-Tutorial [Tut] und dem WebPoint-Tutorial [RW05]. Große Teile sind aus diesen direkt übernommen bzw. wurden nur geringfügig angepasst. Zugunsten einer besseren Lesbarkeit wurde auf eine Kenntlichmachung der Zitate bewusst verzichtet.

Frameworks erleichtern die Programmierarbeit in vielerlei Hinsicht. Sie können Datenstrukturen und Prozesse eines bestimmten Anwendungsgebietes vordefinieren und darüber hinaus einen sauberen Entwurf erzwingen. Dennoch bedeutet ihre Verwendung zunächst einen erhöhten Einarbeitungsaufwand für den Programmierer. Um diesen zu minimieren wurde die folgende Abhandlung geschrieben. Grundlage für das Verständnis dieses Tutorials ist der Einführungsvortrag zum WebPoint Framework.

Auf Basis von WebPoint wird exemplarisch ein Videoautomat Schritt für Schritt zusammengesetzt. Dabei wird dem Leser empfohlen, die einzelnen Schritte per copy & paste selbst zu vollziehen. Darüber hinaus wird an geeigneten Stellen dazu aufgefordert, das Programm zu kompilieren und auszuführen, um den Zusammenhang von WebPoint-Konstrukten und Anzeige zu verdeutlichen. Zunächst wird der zu entwickelnde Automat anhand der vom Automatenbetreiber formulierten Anforderung kurz umrissen.

1.1 Aufgabenstellung

Die Videothek HOMECINEMA bietet mit Hilfe eines Videoverleihautomaten einen vereinfachten 24-Stunden-Service an: Am Videoverleihautomaten erhalten registrierte, erwachsene Kunden Videobänder gegen Bezahlung des Verkaufspreises. Bei Rückgabe von Bändern werden je angefangene 24 Stunden 2,00 € von diesem Einsatz abgezogen und der Rest ausgezahlt. Bleibt dabei kein positiver Rest, erhält der Kunde das Band; es gilt als gekauft. Der Automat hat ein Sortiment von 10 Filmen, die in je 5 Exemplaren vorhanden sind. Aus gesetzlichen Gründen werden sämtliche Leihvorgänge mitprotokolliert. Der Automatenbetreiber nutzt diese Informationen außerdem, um Ladenhüter auszusondern und durch neue Filme zu ersetzen.

1.2 Überblick

Im weiteren Verlauf werden einige Annahmen gemacht, die kurz erläutert werden müssen. Es wird davon ausgegangen, dass die Videoautomaten von HOMECINEMA untereinander nicht vernetzt sind. Somit können die Automaten als eigenständige, voll funktionstüchtige

Verkaufs- bzw. Verleihstellen betrachtet werden und das zu entwerfende Programm repräsentiert genau einen Automaten. Ein Automat besitzt genau ein Display, vorstellbar wäre ein Touch-Screen als Interaktionsfläche für die Kunden und den Betreiber. Eine Person meldet sich autorisiert durch ein Kennwort an und kann entsprechend ihrer Berechtigungen Verleih-, Rückgabe- oder administrative Vorgänge vollführen.

1.3 Hinweise

Auf den folgenden Seiten wird nahezu jeder einzelne Programmierschritt erläutert, wobei auf import-Anweisungen verzichtet wird, um die Übersichtlichkeit zu erhöhen. Es sei an dieser Stelle jedoch noch einmal ausdrücklich darauf hingewiesen, dass sofern eine Klasse oder Methode vom Compiler als unbekannt zurückgewiesen wird, möglicherweise lediglich ein import vergessen wurde. In Eclipse kann dies durch die Funktion „Organize Imports“ (STRG + SHIFT + O) gemacht werden.

In den Codebeispielen sind Abschnitte, die bereits erläutert und daher weggelassen wurden, durch Punkte angedeutet.

1.4 Vorbereitungen für das erste Projekt

Das fertige Programm muss auf einem Server laufen, der Servlets und JSPs unterstützt. Tomcat ist ein solcher Server und gilt als Referenzimplementierung. Um die Java-Programmierung zu erleichtern, bietet es sich an, eine integrierte Entwicklungsumgebung einzusetzen. Eclipse ist eine solche IDE. Sie wurde auch für die Implementierung von WebPoint eingesetzt und hat sich dabei bestens bewährt. Aus diesen Gründen werden wir Tomcat und Eclipse im Verlauf dieses Tutorials einsetzen. Dazu müssen wir beide Programme installieren.

1.4.1 Tomcat installieren

Zur Installation von Tomcat die Datei „jakarta-tomcat-5.5.9.exe“ starten und den Anweisungen auf dem Bildschirm folgen. Wichtig ist, dass wir uns den Benutzernamen und das Passwort für den Administrator und den eingestellten Port merken (Abb. 1). Wenn wir Tomcat gestartet haben, sollten wir, wenn wir im Browser die Adresse `http://127.0.0.1:8080/`¹ aufrufen, die Standardseite von Tomcat angezeigt bekommen.

¹Wobei die 8080 durch den bei der Tomcat Installation angegebenen Port zu ersetzen ist.

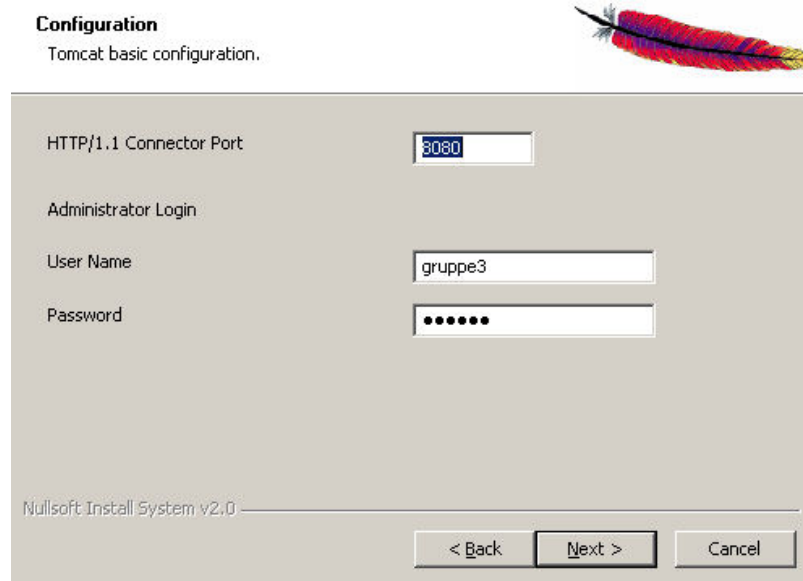


Abbildung 1: Tomcat Installation - Eingabe von Benutzernamen, Passwort und Port

1.4.2 Eclipse installieren

Um Eclipse zu installieren, müssen wir einfach die Datei **eclipse-SDK-3.1-win32.zip** in einen Ordner unserer Wahl entpacken. Nach dem Entpacken können wir Eclipse starten, indem wir die Datei **eclipse.exe** aus dem Installationsverzeichnis ausführen. Nach dem ersten Start wird man gefragt, welchen Ordner Eclipse als Workspace benutzen soll (Abb. 2). Es ist praktisch, das Häkchen bei „Use this as the default and do not ask again“ zu setzen. Dann einfach auf ok klicken.

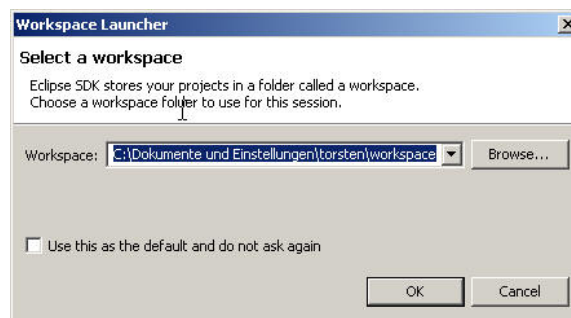


Abbildung 2: Eclipse - erster Start

Um später das Projekt per ssh auf den Praktikumsserver zu übertragen, muss noch die Datei **jsch-0.1.20.jar** in das Verzeichnis **%ECLIPSE_HOME%\plugins\org.apache.ant_1.6.5\lib**² kopiert werden. Jetzt müssen wir Eclipse noch mit-

²%ECLIPSE_HOME% steht für das Installationsverzeichnis von Eclipse und muss entsprechend angepasst werden.

teilen, dass wir eine neue Bibliothek in den Ordner kopiert haben. Dazu im Menü **Window Preferences** wählen und dann unter **Ant -> Runtime** auf **Ant Home...** klicken (Abb. 3) und das Verzeichnis `%ECLIPSE_HOME%\plugins\org.apache.ant_1.6.5` auswählen und mit **OK** bestätigen.

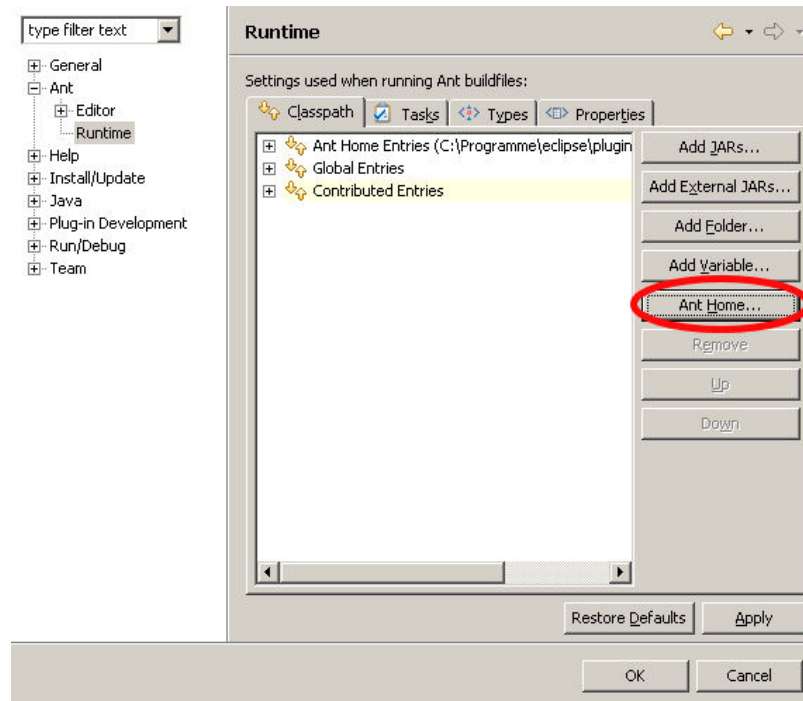


Abbildung 3: Eclipse - Ant Home einstellen

1.5 WebPoint Projekt in Eclipse anlegen

Nachdem die Vorbereitungen abgeschlossen sind, können wir unser erstes WebPoint Projekt starten. Dazu in Eclipse unter **File -> New -> Project...** und dann **Java Project** wählen und auf **weiter** klicken. Im folgenden Dialog geben wir dann „Videoautomat“ als Namen für unser Projekt an und klicken auf **finish**. Jetzt haben wir ein leeres Java-Projekt in Eclipse und müssen nun als nächstes die „Vorlage“ für unser WebPoint-Projekt importieren. Dieses geht am einfachsten, indem wir mit der rechten Maustaste auf unser **Videoautomat**-Projekt klicken und dann **Import...** auswählen. Es erscheint der Dialog wie in Abb. 4 zu sehen.

Da wir ein war-Archiv³ importieren wollen, wählen wir **Archive file** und klicken dann auf **Next**. Im nächsten Dialog unter **From archive file:** die Datei

³Eine war-Datei ist eine spezielles jar-Archiv, dass eine bestimmte Struktur aufweisen muss. In diesem Fall enthält die Datei eine „leere“ WebPoint-Anwendung.

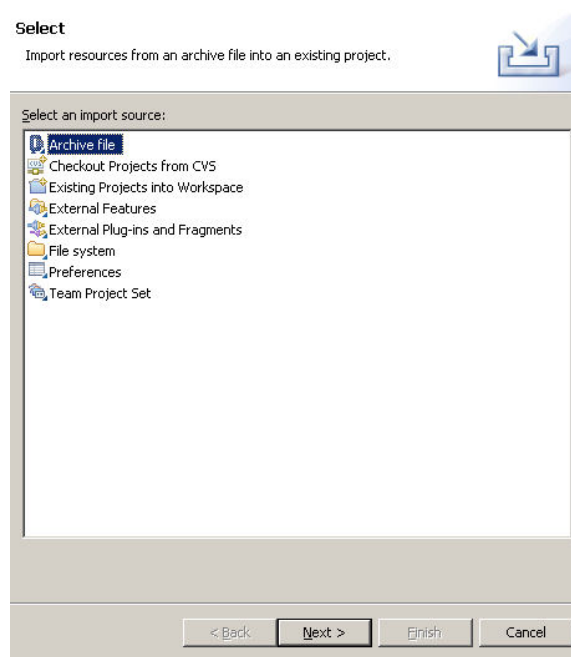


Abbildung 4: Eclipse - „Vorlage“ für WebPoint-Projekt importieren

blank.war⁴ auswählen und dann auf **Finish** klicken. Um Eclipse mitzuteilen, welcher Ordner für den Quelltext genutzt werden soll, klicken wir mit der rechten Maustaste auf den Ordner **src** und wählen unter **Build Path** -> **Use as Source Folder** aus. Jetzt meldet uns Eclipse, dass nicht mehr das ganze Projekt für Quelltexte genutzt wird. Die Meldung einfach mit **OK** bestätigen. Das Projekt sollte jetzt wie in Abb. 5 aussehen.

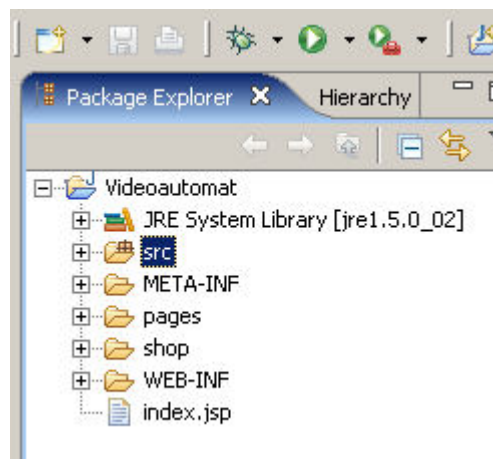


Abbildung 5: Eclipse - Source Folder eingestellt

Als nächstes müssen die benötigten Bibliotheken aus dem gerade entpackten Archiv

⁴Damit die war-Datei mit angezeigt wird, muss in dem Auswahldialog unter Dateityp *.* ausgewählt werden.

noch dem **Build Path** hinzugefügt werden. Dazu in den Ordner **WEB-INF\lib** gehen und alle Dateien in diesem Ordner markieren. Die rechte Maustaste betätigen und dann **Build Path -> Add to Build Path** auswählen (Abb. 6). Zusätzlich werden zum Kompilieren noch zwei Bibliotheken von Tomcat benötigt. Um diese zu importieren, mit der rechten Maustaste auf **Videoautomat** klicken, dann wieder **Build Path** und dann **Add External Archives** auswählen. Im Verzeichnis **%TOMCAT_HOME%\common\lib** die jar-Archive **jsp-api.jar** und **servlet-api.jar** öffnen.

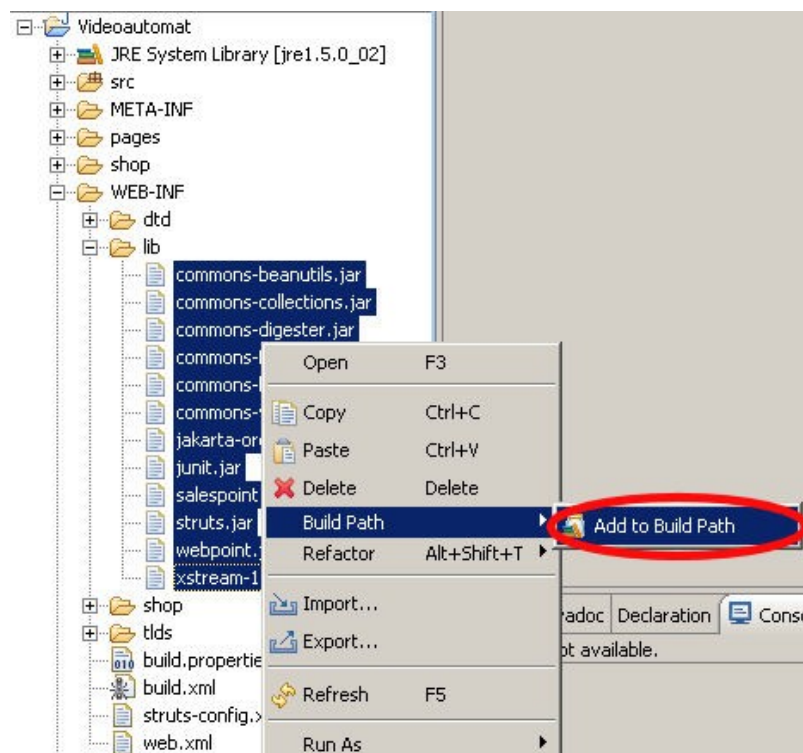


Abbildung 6: Eclipse - Bibliotheken zum Build Path hinzufügen

Damit wir das Projekt auf unseren Tomcat Server übertragen können, müssen wir noch die Einstellungen in der Datei **build.properties** im Ordner **WEB-INF** anpassen. Dazu ändern wir den Wert von **projectname** von **WebPoint** auf **Videoautomat**. Der Wert von **catalina.home** muss dem Installationsverzeichnis von Tomcat entsprechen. Die Variablen **remote.deploy.dir** und **remote.password** sind für das Übertragen des Projektes auf den Praktikumsserver via ssh gedacht. Die ersten beiden Zeilen der **build.properties** nach unserer Änderung:⁵

```
projectname=Videoautomat
catalina.home=C:/programme/tomcat 5.5/
```

⁵Vorrausgesetzt **C:/programme/tomcat 5.5/** ist unser **%TOMCAT_HOME%**-Verzeichnis.

Nachdem die Änderungen durchgeführt und die Datei gespeichert wurde, können wir das Projekt auf den Server übertragen. Das geht mit einem Rechtsklick auf die Datei **build.xml** -> **Run As** -> **2 Ant Build...**, dann die gewünschten Targets auswählen und auf **Run** klicken. Am wichtigsten sind die Targets **deploy** und **deploy.remote**. Beide erstellen zuerst die war-Datei für das Projekt und kopieren diese dann entweder in das lokale `%TOMCAT_HOME%\webapps` Verzeichnis (**deploy**) bzw. auf den Praktikumsrechner (**deploy.remote**). Zum Testen übertragen wir das Projekt auf unseren lokalen Tomcat Server (Target **deploy**) und schauen uns dann das Ergebnis im Browser an.

Auf die Homepage zu unserem Beispielprojekt kommt man über die URL `http://127.0.0.1:8080/Videoautomat/`⁶. Alternativ kann man auch auf die Startseite von Tomcat gehen (`http://127.0.0.1:8080`), sich dort als **Tomcat Manager** einloggen und in der Übersicht der Anwendungen einfach auf **Videoautomat** klicken. In jedem Fall sollte eine HTML-Seite mit dem Text „WebPoint“ angezeigt werden.

Hinweis: Falls das Anzeigen der Seite nicht funktioniert, kann dieses am eingestellten Proxy-Server liegen. Um das zu korrigieren, bitte im Browser einstellen, dass für den Rechner `127.0.0.1` kein Proxy benutzt werden soll.

1.6 Grundlegende Funktion

Wenn wir uns die URL im Browser anschauen, stellen wir fest, dass dort inzwischen etwas ähnliches wie `http://127.0.0.1:8080/Videoautomat/welcome.do;jsessionid=2C6F9BA009118CD996C4FBEDB7D741C5` steht. Im gesamten Java Projekt existiert aber keine **welcome.do**-Datei! Wie also funktioniert das Ganze? Der Browser ruft die URL `http://127.0.0.1:8080/Videoautomat/` auf. Jetzt muss der Webserver wissen, welche Datei angezeigt werden soll. Die Information, welche Datei aufgerufen werden soll, wenn nichts angegeben wurde, steht in der Datei **web.xml** im Verzeichnis **WEB-INF**. Die Datei **web.xml** ist eine XML-Datei zur Konfiguration des Webservers für das Projekt und enthält unter anderem einen Abschnitt

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

Wie man sieht, ist hier nur die Datei **index.jsp** als Willkommensdatei angegeben. Also schauen wir uns diese Datei genauer an. Sie enthält nur die beiden folgenden Zeilen:

⁶8080 ist der bei der Tomcat Installation eingestellte Port und „Videoautomat“ der Wert von **projectname** in der Datei **build.properties**.

```
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<logic:redirect forward="welcome"/>
```

Auch hier steht noch nichts, was nach dem Text „WebPoint“ aussieht. Die erste Zeile gibt an, dass die Tag-Library mit der uri **/tags/struts-logic** mit dem Präfix **logic** verwendet werden soll. Damit stehen in dieser JSP-Datei die Tags der Library zur Verfügung. Die zweite Zeile benutzt einen dieser Tags, nämlich den **redirect**-Tag.⁷ Dieser Tag sorgt dafür, dass das globale **forward** „**welcome**“ aufgerufen wird und sich die URL im Browser damit auf `http://127.0.0.1:8080/Videoautomat/welcome.do` ändert. Wir wissen jetzt aber immer noch nicht, woher dieses **welcome.do** weiß, was es anzuzeigen hat. Diese Information steht in der Datei **struts-config.xml**, die sich auch im Ordner **WEB-INF** befindet. Dort ist im Abschnitt **global-forwards** das **forward** mit dem Namen **welcome** definiert und verweist auf den Pfad **welcome.do**.

```
<global-forwards>
  <forward name="welcome" path="/welcome.do"/>
</global-forwards>
```

Der Pfad **/welcome** ist dann im Abschnitt **action-mappings** als „action forward“ definiert und leitet auf die Seite **/pages/welcome.jsp**:

```
<action-mappings>
  <action path="/welcome" forward="/pages/welcome.jsp"/>
</action-mappings>
```

Diese Datei enthält dann endlich ein **<h1> WebPoint </h1>**.

An dieser Stelle fragt man sich sicher: Wozu eigentlich der ganze Aufwand, nur um so eine simple Datei wie die **welcome.jsp** anzuzeigen? Sicherlich hätte man die Datei **welcome.jsp** auch als **index.jsp** speichern können. Dadurch, dass wir an der Stelle ein **forward** nutzen, haben wir aber die Möglichkeit, die Startseite sehr einfach zu ändern, ohne das Original zu löschen z. B. um Kunden zu informieren, dass die Seite gerade gewartet wird. Um die Seiten wirklich flexibel ersetzen zu können, wurde die Action mit dem Pfad **/welcome** definiert. Diesen Pfad **/welcome.do** kann man jetzt für jeden Verweis auf die Seite nutzen, also auch in „normalen“ HTML-Links. Wenn man später statt der Datei **welcome.jsp** z. B. eine Datei mit dem Namen **neu.jsp** verwenden möchte, braucht man nur die **action** in

⁷ Mehr Informationen zu den logic-Tags gibt es unter <http://struts.apache.org/1.2.4/userGuide/struts-logic.html>

```
<action path="/welcome" forward="/pages/neu.jsp"/>
```

zu ändern. Alle anderen Verweise auf die Seite funktionieren dann noch, da man ja nur indirekt über **welcome.do** darauf zugreift.

Jetzt wollen wir die Startseite etwas anpassen. Statt „WebPoint“ soll später „***** VIDEOAUTOMAT *** 24 H *****“ angezeigt werden. Alle Texte für die Homepage sollen „unformatiert“, das heißt ohne HTML-Formatierung in der Datei **MessageResources.properties**⁸ gespeichert werden. Dieses hat mehrere Vorteile: Zum einen können auf die Texte sowohl aus den JSP-Dateien als auch aus dem Java-Code zugegriffen werden, womit man gleiche Texte/Meldungen nicht mehrmals speichern und später auch warten muss. Zum anderen kann man so relativ einfach verschiedene Sprachversionen unterstützen.

Zurück zu unserem Problem: Als erstes öffnen wir die Datei **MessageResources.properties** und fügen folgende Zeile neu hinzu:

```
videoautomat.caption=***** VIDEOAUTOMAT *** 24 H *****
```

Die Datei **welcome.jsp** sieht nach zwei kleinen Änderungen so aus:

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<%@ taglib uri="/tags/webpoint" prefix="wp" %>

<html:html>
  <head>
    <title> <bean:message key="videoautomat.caption"/> </title>
  <html:base/>
</head>
  <body bgcolor="#FFFFFF">
    <html:errors/>
    <h1> <bean:message key="videoautomat.caption"/> </h1>
  </body>
</html:html>
```

Neu sind hier die beiden **<bean:message key="videoautomat.caption"/>** Zeilen. Damit wird der Text „***** VIDEOAUTOMAT *** 24 H *****“ ausgegeben -

⁸Die Datei ist im Source-Verzeichnis zu finden.

also der Wert, den wir unter dem Schlüssel „videoautomat.caption“ angegeben haben. Um `<bean:message . . . />` verwenden zu können, muss die Tag Library **struts-bean** mit dem Präfix **bean** eingebunden sein. Dies geschieht über die erste Zeile der Datei.⁹ Jetzt können wir das Projekt erneut auf den Server übertragen und uns das Ergebnis anschauen.

1.7 Der Shop

Begonnen wird mit der zentralen Klasse einer jeden WebPoint-Anwendung, dem Shop. Es wird eine neue Klasse **VideoShop**, als Unterklasse von **Shop**¹⁰ erzeugt.

Hinweis: In diesem Beispiel des Videoautomaten wird jede Klasse in einer separaten Datei gespeichert. Die Klassen des Automaten werden zu einem Paket **videoautomat** zusammengefasst. Klassen eines Pakets werden normalerweise in einem Verzeichnis gespeichert, das den Namen des Pakets trägt. Darüberhinaus muss am Anfang einer Klasse eine Zeile der Form: **package paketname;** stehen, die aussagt, welchem Paket die Klasse angehört.

Beim Starten der Webanwendung soll nun eine Instanz unserer Shop-Klasse erstellt werden. Dazu legen wir eine neue Klasse **MainPlugIn** an, die das Interface **PlugIn**¹¹ implementiert.

In der `init`-Methode wird eine Instanz von **VideoShop** erzeugt, welche an die statische Methode `Shop.setTheShop(Shop s)` übergeben wird. Dieser Aufruf bewirkt, dass die übergebene Instanz zur einzigen und global erreichbaren erhoben wird. Auf diese globale Instanz kann über die ebenfalls statische Methode `Shop.getTheShop()` von überall aus zugegriffen werden. Das hier angewandte Entwurfsmuster **Singleton**¹² ist insofern zweckmäßig, da über dieses einzelne Shopobjekt nahezu alle global benötigten Daten gekapselt werden können.

⁹In dieser Datei sind übrigens die vier für „WebPoint“ am häufigsten benutzten Tag Libraries eingebunden. Sie werden hier zwar nicht alle verwendet, aber dafür habt Ihr ein Beispiel, wie sie eingebunden werden.

¹⁰`web.sale.Shop`

¹¹`org.apache.struts.action.PlugIn`

¹²Mehr Informationen zu diesem oder anderen Mustern findet man in [GHJV94].

```
package videoautomat;
public class MainPlugIn implements PlugIn {
    public void init(ActionServlet servlet, ModuleConfig config) {
        VideoShop shop = new VideoShop();
        Shop.setTheShop(shop);
    }
    public void destroy() { }
}
```

Um dem Webserver die Existenz des **MainPlugIns** mitzuteilen, muss in der **struts-config.xml** unter **Plug Ins Configuration** folgendes ergänzt werden:

```
<plug-in className="videoautomat.MainPlugIn"/>
```

1.7.1 Der Videokatalog

Die Videos eines Automaten sind charakterisiert durch einen Titel und die jeweilige Anzahl sowie den Einkaufspreis für den Betreiber und den Verkaufspreis für den Kunden. Entsprechend bietet sich zu ihrer Datenhaltung ein **CountingStock** an. Ein solcher Bestand referenziert auf die Einträge des ihm zugeordneten Katalogs und speichert deren verfügbare Anzahl. Die Katalogeinträge wiederum besitzen die Attribute Bezeichnung und Preis.

Dementsprechend wird zunächst ein **Catalog** benötigt, der die Videonamen und -preise enthält. Da es sich dabei um ein Interface handelt, bedarf es einer Klasse, die dieses Schnittstellenverhalten implementiert. Im Framework existiert bereits eine vordefinierte Klasse namens **CatalogImpl**, die für die meisten Zwecke ausreichen dürfte. Der Konstruktor dieser Klasse verlangt einen **String**-Bezeichner, der den Katalog eindeutig von anderen unterscheidet. Es wird zunächst folgende Zeile der Klasse **VideoShop** hinzugefügt:

```
public class VideoShop extends Shop {
    public static final String C_VIDEOS = "VideoCatalog";
    .
    .
}
```

Diese Konstante soll der künftige Bezeichner für den Videokatalog sein. Der Katalog wird im Konstruktor von **VideoShop** wie folgt instantiiert:

```
public VideoShop() {
    super();
    addCatalog(new CatalogImpl(C_VIDEOS));
}
```

Der Videokatalog ist durch die Aufnahme in die Katalogsammlung des Ladens von jeder Klasse der Anwendung aus erreichbar, jedoch ist der Aufruf, um an den Katalog zu gelangen, unangenehm lang und wird vermutlich mehr als einmal verwendet. Zur Erleichterung wird eine statische Hilfsmethode in der Klasse **VideoShop** geschaffen, die den Videokatalog zurückgibt.

```
public class VideoShop extends Shop {
    .
    .
    public static CatalogImpl getVideoCatalog() {
        return (CatalogImpl) Shop.getTheShop()
            .getCatalog(C_VIDEOS);
    }
}
```

Nun existiert zwar ein Katalog, jedoch ohne Einträge. Damit im weiteren Verlauf des Programmierens und Testens einige Daten zur Verfügung stehen, wird die **MainPlugIn** um folgende Methode ergänzt:

```
public class MainPlugIn implements PlugIn {
    .
    .
    public static void initializeVideos() {
        for (int i = 0; i < 10; i++) {
            String s = "Video-" + i;
            VideoShop
                .getVideoCatalog()
                .add(new CatalogItemImpl(
                    s,
                    new QuoteValue(
                        new IntegerValue(1500),
                        new IntegerValue(3000))) {
                    protected CatalogItemImpl getShallowClone() {
```

```

        return null;
    }
},
null);
}
}
}

```

Die Methode bedarf ein wenig der Erklärung. Wie unschwer zu erkennen, werden in einer for-Schleife dem Videokatalog mehrere Katalogeinträge hinzugefügt. Es handelt sich bei **CatalogItemImpl** um eine abstrakte Implementation des Interface **CatalogItem**. Daher muss bei der direkten Instanzierung dieser Klasse die abstrakte Methode **getShallowClone()** implementiert werden. Diese Methode soll normalerweise einen echten Klon der Instanz zurückliefern. In diesem Fall wird die Methode nicht benötigt und liefert daher lediglich null zurück.

Dem Konstruktor von **CatalogItemImpl** muss ein **String** und ein **Value** übergeben werden. **Value** ist ein Interface und es existieren zwei Implementationen dieser Schnittstelle im Framework. Zum Einen **NumberValue**, welches einen numerischen Wert kapselt und **QuoteValue**, das ein Paar von Werten repräsentiert, z. B. einen Ein- und Verkaufswert. In diesem Fall wird dem Konstruktor von **CatalogItemImpl** der Titel des Videos und als zweiter Parameter eine Instanz von **QuoteValue** übergeben, welches wiederum mit zwei **IntegerValue** erzeugt wird. **IntegerValue** ist lediglich eine Spezialisierung von **NumberValue**, die einen int-Wert kapselt, hierbei der Wert ausgedrückt in Cent.

Abschließend wird in der **init**-Methode der **MainPlugIn** die neue Methode zur Ausführung gebracht, damit die Änderungen wirksam werden.

```

public class MainPlugIn implements PlugIn {
    public void init(ActionServlet servlet, ModuleConfig config) {
        .
        .
        initializeVideos();
    }
    .
    .
}

```

1.7.2 Der Videobestand

Nach der Fertigstellung des Katalogs kann im Folgenden der Bestand aufgebaut werden. Wie bereits im Abschnitt 1.7.1 erwähnt, sollen die Videos des Automaten in einem **CountingStock** gespeichert werden. Auch für dieses Interface existiert eine vordefinierte Klasse, wieder mit Impl am Ende des Namens. Ein jeder Bestand bezieht sich auf einen Katalog, so dass dieser konsequenterweise neben dem String-Bezeichner dem Konstruktor von **CountingStockImpl** übergeben werden muss. Es wird analog zum Videokatalog in den Konstruktor von **VideoShop** folgende Zeile eingefügt:

```
public class VideoShop extends Shop {
    .
    .
    public VideoShop() {
        .
        .
        addStock(
            new CountingStockImpl(
                CC_VIDEOS,
                (CatalogImpl) getCatalog(CC_VIDEOS)));
        }
        .
        .
    }
```

Am Anfang der Klasse muss außerdem die String-Konstante deklariert werden.

```
public static final String CC_VIDEOS = "VideoStock";
```

Auch beim Videobestand lohnt es sich, den Zugriff durch eine Hilfsmethode zu erleichtern.

```
public class VideoShop extends Shop {
    .
    .
    public static CountingStockImpl getVideoStock() {
        return (CountingStockImpl) Shop.getTheShop()
            .getStock(CC_VIDEOS);
    }
}
```

Der neue Bestand ist wiederum leer. Durch das Hinzufügen einer Zeile in die Initialisierungsmethode der Videos in `MainPlugIn` können dem Videobestand die benötigten Testdaten zugefügt werden.

```
public static void initializeVideos() {
    for (int i = 0; i < 10; i++) {
        String s = "Video-" + i;
        .
        .
        VideoShop.getVideoStock().add(s, 5, null);
    }
}
```

Der Aufruf `add(String id, int count, DataBasket db)` bewirkt, dass von dem Katalogeintrag mit der Bezeichnung `id` insgesamt `count`-Stück in den Bestand aufgenommen werden. Der `DataBasket`, der zum Schluss übergeben wird, hat etwas mit der Sichtbarkeit der vollführten Aktion zu tun. Was es mit dem Interface `DataBasket` auf sich hat, wird im Abschnitt 1.14.1 näher erläutert. Vorerst reicht es zu wissen, dass hier durch die Übergabe von `null` das Hinzufügen unmittelbar wirksam wird.

1.8 Den Videobestand anzeigen

In diesem Abschnitt soll der zuvor konstruierte Videobestand für den Kunden sichtbar gemacht werden. Wie im Einführungsvortrag erläutert, gibt es für die Darstellung von Katalogen, Beständen usw. in Java Server Pages bereits vorgefertigte Tags in der Tag-Library „wp“. Zur Darstellung gibt es verschiedene Tabellen-Tags:

- **DataBasketTable** - für Warenkörbe (**DataBasket**)
- **CatalogTable** - für Kataloge (**Catalog**)
- **CountingStockTable** - für aufzählende Bestände (**CountingStock**)
- **StoringStockTable** - für speichernde Bestände (**StoringStock**)
- **LogTable** - für Inhalte von Logdateien

Ein Tabellen-Tag bildet die einzelnen Elemente der jeweiligen Datenstruktur auf die Zeilen und deren Attribute auf die Spalten ab. Die Frage ist jedoch, woher die Information stammt, welche Attribute des jeweiligen Datums wie angezeigt werden sollen. Die Antwort liefert das Interface **TableEntryDescriptor** (TED). Eine Klasse mit dem

Verhalten eines TED kann unter anderem die Spaltenanzahl, die Überschriften, den Inhalt der Spalten sowie die Art der Formatierung über verschiedene Methoden zurückgeben. Die Klasse **AbstractTableEntryDescriptor** reduziert den Aufwand der Implementierung eines TEDs, da in ihr diverse Methoden vordefiniert sind. Eine neu erzeugte Klasse **TEDVideoStock** soll von **AbstractTableEntryDescriptor**¹³ erben:

```
package videoautomat;

public class TEDVideoStock extends AbstractTableEntryDescriptor {
    public int getColumnCount() {
        return 3;
    }
    public String getColumnName(int nIndex) {
        String[] cNames = { "Name", "Price", "Count" };
        return cNames[nIndex];
    }
    public Class getColumnClass(int nIndex) {
        Class[] cClasses = { String.class, NumberValue.class,
                                Integer.class };
        return cClasses[nIndex];
    }
    public Object getValueAt(Object oRecord, int nIndex) {
        CountingStockTableModel.Record r =
            (CountingStockTableModel.Record) oRecord;
        switch (nIndex) {
            case 0 :
                return r.getDescriptor().getName();
            case 1 :
                return ((QuoteValue)
                    r.getDescriptor().getValue()).getOffer();
            case 2 :
                return new Integer(r.getCount());
        }
        return null;
    }
    public String getID(Object oRecord) {
        return "" + getValueAt(oRecord,0) + getValueAt(oRecord,1);
    }
}
```

¹³`web.util.swing.AbstractTableEntryDescriptor`

```

    }
}

```

Die erste Methode liefert die Spaltenanzahl. Die jeweilige Überschrift wird durch die Methode `getColumnName(int idx)` ermittelt. Dazu wird der Spaltenindex übergeben. Beispielsweise erhält man die Überschrift der ersten Spalte durch den Aufruf `getColumnName(0)`. Informationen über den Typ der anzuzeigenden Daten und damit verbunden über die nötige Art der Formatierung liefert `getColumnClass(int idx)`. Die einzutragenden Daten gibt `getValueAt(Object oRecord, int nIndex)` zurück. Der zuletzt genannten Methode muss der betreffende Zeileneintrag übergeben werden. Der Typ dieses Zeileneintrags hängt vom verwendeten Tabellenmodell ab, dieses wiederum von der Datenstruktur, die angezeigt werden soll. Ein Blick in die WebPoint-API auf die Subklassen von `AbstractTableModel` genügt, um das jeweils verwendete Tabellenmodell und damit den Typ der Einträge zu bestimmen.

Die Methode `getID(Object oRecord)` bekommt ebenfalls den betreffenden Zeileneintrag übergeben. Sie gibt einen eindeutigen `String` zurück, der verwendet wird, um die Zeilen zu unterscheiden.¹⁴

In diesem Fall soll ein `CountingStock` angezeigt werden, entsprechend handelt es sich bei den Tabelleneinträgen um den Typ `CountingStockTableModel.Record`. Wie die Attribute, die angezeigt werden sollen, über diese Klasse zurückgegeben werden können, ist im obigen Beispiel des `TEDVideoStock` zu erkennen bzw. in der API nachzuschlagen.

Der aktuelle Bestand der Videos soll als Begrüßungsbildschirm des Automaten fungieren, so dass die Kunden sofort einen Blick auf die aktuelle Filmauswahl haben, ohne sich vorher anmelden zu müssen. Dazu müssen wir die Datei `welcome.jsp` wie folgt anpassen:

```

<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<%@ taglib uri="/tags/webpoint" prefix="wp" %>
.
.
<h1> <bean:message key="videoautomat.caption"/> </h1>
<wp:CountingStockTable
    cs="<%= (videoautomat.VideoShop)web.sale.Shop

```

¹⁴Dies wird später in Formularen benötigt, die Daten nicht nur anzeigen, sondern den Benutzer auch Datensätze auswählen lassen.

```

        .getTheShop().getVideoStock()%"
    ted="<%=new videoautomat.TEDVideoStock()%" />
</body>
</html:html>

```

Wir benutzen hier den **CountingStockTable**-Tag, dem wir als Parameter **cs** unseren Videobestand und als Parameter **ted** unseren gerade erstellten **TEDVideoStock** übergeben. Um die WebPoint Tags mit dem Präfix **wp** nutzen zu können, muss die Taglib **webpoint** eingebunden werden, was jedoch bei der **welcome.jsp** schon geschehen ist. Nach einem **deploy** sollte die Seite im Browser wie in Abb. 7 aussehen.

***** VIDEOAUTOMAT *** 24 H *****

| Name | Price | Count |
|---------|-------|-------|
| Video-0 | 3000 | 5 |
| Video-1 | 3000 | 5 |
| Video-2 | 3000 | 5 |
| Video-3 | 3000 | 5 |
| Video-4 | 3000 | 5 |
| Video-5 | 3000 | 5 |
| Video-6 | 3000 | 5 |
| Video-7 | 3000 | 5 |
| Video-8 | 3000 | 5 |
| Video-9 | 3000 | 5 |

Abbildung 7: Videoautomat - Anzeige des Videobestandes

Auffällig ist die Anzeige der Preise. Es wird zwar der korrekte Wert in Cent ausgegeben, jedoch wäre es wünschenswert, die allgemein übliche Währungsformatierung zu nutzen. Um diesen Schönheitsfehler zu korrigieren, ist zuvor die Definition einer Währung notwendig. Dieser Vorgang wird im Abschnitt 1.11 beschrieben.

1.9 Die Nutzerverwaltung

Bevor mit dem ersten Prozess der Anwendung, dem Anmeldevorgang, begonnen werden kann, muss die Nutzerverwaltung angelegt werden. Zur Erinnerung: Es können lediglich registrierte Kunden am Automaten Filme ausleihen. Entsprechend braucht die Anwendung eine Datenstruktur, anhand derer der Automat erkennen kann, wer Kunde ist und wer nicht.

WebPoint bietet dafür die Klassen **User** und **UserManager** an.

1.9.1 Der Usermanager

Der Nutzermanager ist eine Art Containerklasse, in der alle Nutzer gespeichert werden. Ebenso wie beim **Shop** wurde bei der Klasse **UserManager** auf das Entwurfsmuster Singleton zurückgegriffen, das heißt, es gibt genau eine Instanz des Nutzermanagers, die über **UserManager.getGlobalUM()** referenziert werden kann.

Im Konstruktor von **VideoShop** wird also eine neue Instanz von **UserManager** erzeugt und zur globalen Instanz erhoben.

```
public class VideoShop extends Shop {
    .
    .
    public VideoShop() {
        .
        .
        UserManager.setGlobalUM(new UserManager());
    }
    .
}
```

1.9.2 Der Automatenutzer

Anwender des Programms können durch die Klasse **User** dargestellt werden. Ein **User** besitzt einen Namen, über den er eindeutig identifiziert werden kann. Darüber hinaus besteht die Möglichkeit, ein Passwort zu setzen sowie Rechte auf mögliche Aktionen zu vergeben.

Damit die entliehenen Videos direkt beim Kunden gespeichert werden können, muss eine neue Klasse von **User** abgeleitet werden.

```
package videoautomat;
public class AutomatUser extends User {
    private StoringStockImpl ss_videos;
    public AutomatUser(String user_ID, char[] passWd) {
        super(user_ID);
        setPassWd(garblePassWD(passWd));
        ss_videos = new StoringStockImpl(user_ID,
                                         VideoShop.getVideoCatalog());
    }
    public StoringStockImpl getVideoStock() {
```

```
        return ss_videos;
    }
}
```

Abgesehen vom Kunden gibt es die Nutzergruppe der Betreiber bzw. Administratoren. Für diese legen wir eine Unterklasse von **AutomatUser** mit dem Namen **AutomatAdmin** an.

```
package videoautomat;
public class AutomatAdmin extends AutomatUser {
    public AutomatAdmin(String user_ID, char[] passWd) {
        super(user_ID, passWd);
    }
}
```

An dieser Stelle können wir für die entliehenen Videos keinen **CountingStock** verwenden, weil dieser nur die Anzahl gewisser Katalogeinträge speichert. Wir müssen später jedoch für jedes einzelne Video noch die Leihzeit wissen, die ja bei Videos mit dem gleichen Katalogeintrag verschieden sein kann. Aus diesem Grund verwenden wir hier eine Klasse, die das Interface **StoringStock** implementiert, welches uns die Unterscheidung ermöglicht. **CountingStockImpl** ist eine solche Klasse. Die Bestandseinträge der Videos werden über die Klasse **VideoCassette** definiert. Deren Implementation erfolgt im Abschnitt 1.13. Abgesehen von diesem Unterschied des **StoringStockImpl** zum **CountingStockImpl** ist der Konstruktoraufruf nahezu identisch.

Weiterhin ist zu erkennen, dass beim Setzen des Passworts nicht das Passwort selbst, sondern der Rückgabewert der Methode **garblePassWD (passWd)** übergeben wird. Dabei handelt es sich um eine Sicherheitsmaßnahme. Damit das Passwort nicht im Klartext abgespeichert wird und ausgelesen werden kann, wird es vorher verschlüsselt. Welcher Algorithmus dabei verwendet wird, kann ebenfalls über die Klasse **User** gesetzt werden. Der voreingestellte Algorithmus für die Umschlüsselung ist einfach umkehrbar und daher für sicherheitsrelevante Kontexte nicht zu empfehlen. Wird das Passwort mit der Methode **isPassWd (String s)** überprüft, ist darauf zu achten, dass der übergebene String vorher ebenfalls verschlüsselt wird.

Zum Schluss sollen einige Testkunden und ein Administrator der Nutzerverwaltung zugefügt werden. Dafür wird eine Methode in der Klasse **MainPlugIn** definiert und in der main-Methode aufgerufen. Als Passwort wird der vereinfachten Testbarkeit halber jeweils eine leere Zeichenkette übergeben.

```
public void init(ActionServlet servlet, ModuleConfig config)
```

```

    .
    .
    initializeUsers();
}
public static void initializeUsers() {
    UserManager.getGlobalUM().addUser(
        new AutomatAdmin("Administrator", new char[0]));
    for (int i = 0; i < 10; i++) {
        UserManager.getGlobalUM().addUser(
            new AutomatUser("Customer" + i, new char[0]));
    }
}
}

```

1.10 Die Anmeldung

Es sind nun alle Voraussetzungen erfüllt, um den Anmeldeprozess zu implementieren. Bei diesem initialen Prozess muss der Anwender sich zunächst durch Name und Passwort identifizieren. Nach erfolgreicher Anmeldung werden die weiteren möglichen Aktivitäten angezeigt. Abbildung 8 verdeutlicht dies in einem Zustandsdiagramm.

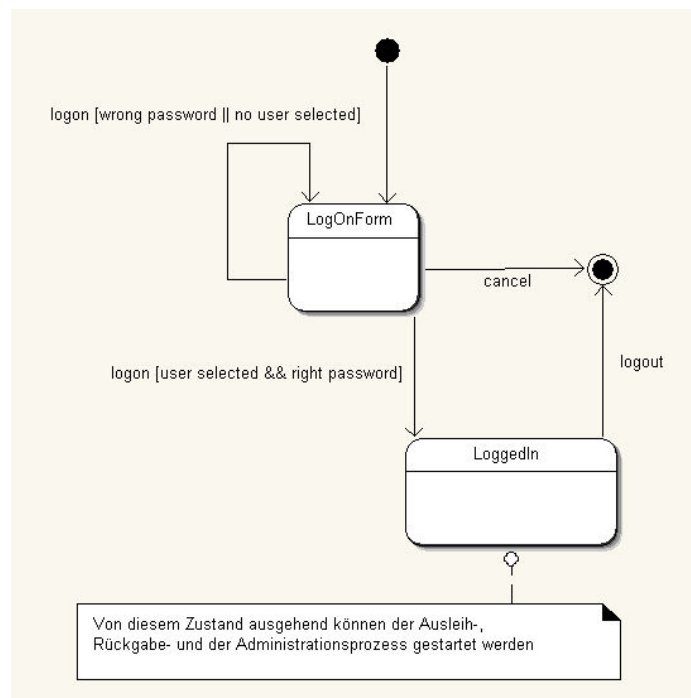


Abbildung 8: Videoautomat - Anmeldeprozess

Für die Zustände **LogOnForm** und **LoggedIn** werden wir JSPs erzeugen. Diese die-

nen der Interaktion mit dem Benutzer, hier kann der Benutzer von uns erstellte Links bzw. Buttons nutzen um Transitionen zu starten. Beim Zustand **LogOnForm** muss der Nutzer seinen Benutzernamen und das Passwort eingeben, dazu brauchen wir zusätzlich zur JSP noch ein **Form**-Objekt. Dieses Java-**Form**-Objekt ermöglicht uns einfach auf die Eingaben des Nutzers zuzugreifen.

Einfache Transitionen des Zustandsdiagramms kann man als gewöhnliche Links realisieren. So werden wir später den Übergang zum Startzustand realisieren. Transitionen, die Arbeit erledigen oder Entscheidungen treffen müssen, werden durch **Actions** realisiert. So muss zum Beispiel beim Übergang **logon** überprüft werden, ob das eingegebene Passwort korrekt ist. Innerhalb der **Action** kann man dann auf das **Form**-Objekt zugreifen und gibt entsprechend der Eingaben ein **Forward** zurück, welches andeutet, auf welche Seite der Benutzer weitergeleitet werden soll. Die endgültige Zuordnung von **Forward** zu JSP bzw. weiterer **Action** erfolgt mittels der Konfigurationsdatei **struts-config.xml**. Auch die Transitionen vom Startzustand bzw. zum Endzustand werden durch einen Link bzw. eine **Action** realisiert. An dieser Stelle müssen wir uns als Entwickler darüber klar werden, was der „umgebende Prozess“ ist bzw. wohin der Übergang zum Endzustand gehen soll. In diesem Fall ist Ausgangs- und Endpunkt des Prozesses die Seite **welcome.jsp**.

Zusammengefasst benötigen für den Zustand **Logon**

- eine **logon.jsp** für die Eingabe von Benutzername und Passwort
- ein **LogOnForm** in das diese Daten übertragen werden
- eine **LogOnAction** für die Transitionen ausgehend von diesem Zustand

Diese drei Komponenten: JSP, **Form** und **Action** gehören zusammen und werden später über die **transitions.xml**¹⁵ und die **struts-config.xml** miteinander verknüpft. Für die Realisierung des Zustandes **LoggedIn** benötigen wir:

- eine **loggedin.jsp** die dem Benutzer später die Transitionen zum starten der weiteren Prozesse und die Transition **logout** ermöglicht
- und eine **LogOutAction**, die den Benutzer wieder abmeldet.

¹⁵In der **transitions.xml** wird die Reihenfolge der Zustandsübergänge genau definiert, d.h., sie stellt ein Abbild des Zustandsdiagrammes für die gesamte Anwendung dar. Dies dient der Sicherheit, um ungewollte Sprünge des Users in der Anwendung durch manuelle Linkeingaben zu verhindern. Aus ihr können Teile der **struts-config.xml** automatisiert erzeugt werden.

1.10.1 logon.jsp

Die Datei könnte folgendermaßen aussehen:

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>

<html:html>
<head>
  <title><bean:message key="videoautomat.caption"/> -
    <bean:message key="logon.caption"/></title>
  <html:base/>
</head>

<body bgcolor="white">
  <html:errors/>
  <html:form action="/LogOnAction">
    <table>
      <tr>
        <td align="center" colspan="2">
          <h1><bean:message key="logon.caption"/></h1>
        </td>
      </tr>
      <tr>
        <td align="right">
          <bean:message key="logon.enter.username"/>
        </td>
        <td align="left">
          <html:text property="username" size="30"
            maxlength="30"/>
        </td>
      </tr>
      <tr>
        <td align="right">
          <bean:message key="logon.enter.password"/>
        </td>
        <td align="left">
          <html:password property="password" size="30"
            maxlength="30"/>
        </td>
      </tr>
    </table>
  </html:form>
</body>
</html>
```

```

                                maxlength="30"/>
        </td>
    </tr>
    <tr>
        <td align="right">
            <html:submit property="method">
                <bean:message key="videoautomat.ok"/>
            </html:submit>
        </td>
        <td align="left">
            <html:cancel>
                <bean:message key="videoautomat.cancel"/>
            </html:cancel>
        </td>
    </tr>
</table>
</html:form>
</body>
</html:html>

```

Die ersten beiden Zeilen machen, wie schon öfter erwähnt, die Tag-Libraries (hier **struts-html** und **struts-bean**) für diese JSP verfügbar. Auch auf dieser Seite werden keine Texte direkt ausgegeben, sondern sind in der Datei **MessageResources.properties** gespeichert. Dazu wurde diese wie folgt ergänzt:

```

.
.
videoautomat.ok=Ok
videoautomat.cancel=Cancel

logon.caption=Please Log On
logon.enter.username=Enter your user name:
logon.enter.password=Enter your passphrase:

```

Die Benutzereingaben finden innerhalb eines HTML-Formulars statt. Die Anfangs- und End-Tags des Formulars werden durch **<html:form action="/LogOnAction">** und **</html:form>** generiert, wobei das Formular mit der Aktion **/LogOnAction** ver-

knüpft wird, welche wir später noch erstellen. Das Formularfeld für den Benutzernamen wird mit der Anweisung

```
<html:text property="username" size="30" maxlength="30"/>
```

erzeugt. Dabei gibt **property** an, welcher Setter später zum Übertragen der Eingabe in die Java Klasse aufgerufen werden soll - hier **setUsername(...)**, **size** gibt die Größe des Eingabefeldes und **maxlength** die maximale Länge der Texteingabe an. Für das Passwort sieht die Anweisung sehr ähnlich aus:

```
<html:password property="password" size="30" maxlength="30"/>
```

Sie unterscheidet sich nur dadurch, dass die Eingabe nicht im Klartext zu sehen ist, sondern durch „*“ ersetzt wird.¹⁶

Abschließend werden noch zwei Buttons mit den Anweisungen

```
<html:submit property="method">
    <bean:message key="videoautomat.ok"/>
</html:submit>
<html:cancel>
    <bean:message key="videoautomat.cancel"/>
</html:cancel>
```

erstellt. Auch hier werden die Beschriftungen für die Buttons in der Datei **MessageResources.properties** gespeichert.

Der erste Button ist ein „submit“-Button und seine Beschriftung wird als Wert von „method“ weitergegeben. Der zweite Button ist ein „cancel“-Button. Bei diesem wird keine Validierung der Formulardaten vorgenommen, was auch so erwünscht ist.

Wichtig ist noch die Zeile **<html:errors/>**. Damit werden mögliche Fehler für den Benutzer ausgegeben. Im Abschnitt 1.10.2 wird zum Beispiel eine Fehlermeldung erzeugt, wenn der Benutzer kein Passwort eingibt. Diese Fehler würden dann an der Stelle ausgegeben, wo das **<html:errors/>** steht.

1.10.2 LogOnForm

Die Java Klassen, die wir nur für einen Prozess benötigen, gruppieren wir jeweils in einem eigenen Paket. Für den Anmeldeprozess erstellen wir das package

¹⁶Dieses ändert nichts daran, dass das Passwort trotzdem im Klartext übertragen wird. Deshalb sollte man für reale Anwendungen Tomcat für ssl-Verbindungen konfigurieren, was wir an dieser Stelle nicht weiter betrachten wollen.

videoautomat.logon. Um die Daten der **logon.jsp** aufzunehmen, müssen wir eine neue Klasse von **ActionForm**¹⁷ ableiten.

```
package videoautomat.logon;
public class LogOnForm extends ActionForm {
    private String username;
    private String password;
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public void reset(ActionMapping mapping,
                      HttpServletRequest request) {
        username = "";
        password = "";
    }
    public ActionErrors validate(ActionMapping mapping,
                                HttpServletRequest request) {
        ActionErrors errors = new ActionErrors();
        if (username==null || username.length()==0) {
            errors.add("username",
                      new ActionMessage("logon.error.username.required"));
        }
        return errors;
    }
}
```

Passend zu den in **logon.jsp** angegebenen properties **username** und **password** wurden hier gleichnamige lokale Variablen und passende Getter und Setter erstellt. Was auffällt

¹⁷org.apache.struts.action.ActionForm

ist, dass die Setter jeweils Strings übergeben bekommen. Dies ist nicht nur hier so, sondern sollte generell so gehandhabt werden, weil der Browser die eingegebenen Daten nur als Zeichenketten überträgt und nicht als **int** oder ähnliches. Die Methode

```
public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest request)
```

ist dazu gedacht die Eingaben auf Gültigkeit zu prüfen. Hier wird zum Beispiel geprüft, ob ein Benutzername eingegeben wurde. Falls nicht wird eine neue **ActionMessage** erzeugt und als Bestandteil von **ActionErrors** unter dem Schlüssel **username** zurückgegeben. Wenn Fehler zurückgegeben werden, wird man zurück auf die Seite geschickt, von der man gerade kommt. Dort sollte dann der Programmierer die Fehler für den Endnutzer anzeigen.¹⁸ Damit auch eine Meldung angezeigt werden kann, müssen wir in der **MessageResources.properties** noch die Zeile

```
logon.error.username.required=Username is required!
```

ergänzen. Wie wir sehen, gibt es auch noch eine Methode **public void reset(...)**, die vor jeder Benutzung des ActionForms aufgerufen wird. Sie ist deshalb wichtig, weil **Struts** die ActionForms mehrfach verwendet und nicht jedesmal eine neue Instanz der Klasse erzeugt. Hier werden Benutzername und Passwort auf leere Strings gesetzt. Man sollte dazu wissen, dass Webbrowser nicht unbedingt leere Zeichenketten an den Server übertragen müssen. Wenn also ein Benutzer aus Versehen ein Passwort eingegeben hätte, was falsch ist und ihm dann bewusst wird, dass er überhaupt kein Passwort gespeichert hatte, könnte er zwar im Browser das Passwortfeld leeren. Da aber der Browser das leere Feld nicht unbedingt übertragen muss, wird auch der Setter nicht erneut aufgerufen und das Feld würde weiterhin das zuerst eingegebene Passwort enthalten. Da wir das Passwort in der **reset**-Methode leeren, funktioniert jetzt ein späteres Löschen des Feldes.

Hinweis: Die Methoden **reset** und **validate** kann man auch sehr gut in Kombination nutzen, um zu überprüfen, ob der Nutzer die Daten auch richtig formatiert eingegeben hat. Dazu ein kleines Beispiel: Angenommen wir wollen, dass der Benutzer eine Zahl kleiner als 10 eingibt, dann könnten wir folgendes **ActionForm** verwenden:

```
public class ZahlForm extends ActionForm {
    private int zahl;
    ActionErrors errors;
    public int getZahl() {
        return zahl;
    }
}
```

¹⁸Das haben wir mit `<html:errors />` getan.

```
}
public void setZahl(String zahl) {
    try {
        this.zahl = Integer.parseInt(zahl);
    } catch (NumberFormatException e) {
        errors.add("number",
            new ActionMessage("number.parse.exception"));
    }
}
public void reset(ActionMapping mapping,
    HttpServletRequest request) {
    zahl = 0;
    errors = new ActionErrors();
}
public ActionErrors validate(ActionMapping mapping,
    HttpServletRequest request) {
    if (zahl>10)
        errors.add("number", new ActionMessage("number.invalid"));
    return errors;
}
}
```

In der **reset**-Methode setzen wir die Variable **errors** zurück und können so in **setZahl** überprüfen, ob man aus dem eingegebenem String eine Zahl machen kann und anderenfalls eine Fehlermeldung erzeugen. Die **validate**-Methode prüft dann nur noch, ob die eingegebene Zahl auch „inhaltlich korrekt“ also kleiner als 10 ist und gibt dann alle aufgetretenen Fehler zurück.

1.10.3 LogOnAction

Nachdem wir jetzt die Oberfläche für die Benutzereingabe in Form unserer **logon.jsp** haben und die Daten daraus in unser **LogOnForm** übertragen werden können, müssen wir noch die in der **logon.jsp** angegebene **action="/LogOnAction"** implementieren. Diese Aktion muss auf die beiden Buttons der Seite reagieren können. Dazu erstellen wir eine neue Klasse **LogOnAction**, die von **LookupDispatchAction** abgeleitet wird.

```
public class LogOnAction extends LookupDispatchAction {
    protected Map getKeyMethodMap() {
```

```

    Map map = new HashMap();
    map.put("videoautomat.ok", "logon");
    return map;
}
public ActionForward cancelled(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response) {
    .
    .
}
public ActionForward logon(ActionMapping mapping,
                            ActionForm form,
                            HttpServletRequest request,
                            HttpServletResponse response) {
    .
    .
}
}
}

```

Da die Beschriftungen der Buttons in der **MessageResources.properties** definiert sind und für jede Sprache anders aussehen können, muss man eine Rückwärtszuordnung von Beschriftung zu Schlüssel vornehmen, um dann für jeden Schlüssel¹⁹ eine Methode zu definieren, die ausgeführt werden soll, wenn der Button betätigt wird. Genau dieses erledigt die **LookupDispatchAction** für uns. Wir müssen dazu nur die Methode **getKeyMethodMap** implementieren. Diese Methode muss eine Map zurückgeben, die als Schlüssel die „Schlüssel“ für die Beschriftungen der Buttons enthält und als Werte die dazugehörigen Funktionsnamen. Den „cancel“-Button brauchen wir dabei nicht berücksichtigen. Für ihn wird automatisch die Methode „cancelled“ ausgeführt. Wir müssen nur noch die als Werte angegebenen Methoden und entsprechend die „cancelled“-Methode implementieren.

Beginnen wir mit dem Cancel-Button. Gemäß unserem Zustandsübergangdiagramm (Abb. 8) kommt man damit zum Endzustand. Diesen benennen wir für unsere Implementierung mit „exit“.

```

public ActionForward cancelled(ActionMapping mapping,

```

¹⁹Also jeden Button, denn die Beschriftungen der Button sollten eindeutig sein, damit man sie unterscheiden kann.

```

        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response) {
    return mapping.findForward("exit");
}

```

Das **mapping** in der Methode beschreibt die Konfiguration der Aktion in der **struts-config.xml**, auf die wir später noch zurückkommen.

Das **findForward("exit")** sucht uns aus der Konfiguration das Forward mit dem Namen „exit“ raus. Das heißt, an dieser Stelle wird wiederum nicht direkt auf eine bestimmte JSP verwiesen, sondern nur auf ein bestimmtes **forward**. Damit hat man die Möglichkeit diese **LogOnAction** an mehreren Stellen wiederzuverwenden und jedes Mal eine andere Seite anzugeben.

Nun zu unserer **logon**-Methode:

```

public ActionForward logon(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response) {
    LogOnForm lof = (LogOnForm) form;
    User user =
        UserManager.getGlobalUM().getUser(lof.getUsername());
    if (user!=null) {
        if (user.isPassWd(User.garblePassWD(
            lof.getPassword().toCharArray())) {
            request.getSession().invalidate();
            new SessionContext(request).attach(user);
            return mapping.findForward("loggedin");
        } else {
            // wrong password
            ActionMessages messages = new ActionMessages();
            messages.add("logon", new ActionMessage("logon.error"));
            saveErrors(request, messages);
            return mapping.getInputForward();
        }
    } else {
        // username does not exist
        ActionMessages messages = new ActionMessages();

```

```

        messages.add("logon", new ActionMessage("logon.error"));
        saveErrors(request, messages);
        return mapping.getInputForward();
    }
}

```

Hier müssen wir zum ersten Mal auf die im **LogOnForm** gespeicherten Daten zugreifen. Das **LogOnForm** bekommen wir als **ActionForm** übergeben und müssen es nur noch auf die richtige Klasse casten. Danach nutzen wir die Methode **getUser** des **UserManager**, um uns den **User** mit dem im **LogOnForm** angegebenen Namen liefern zu lassen bzw. **null**, falls ein solcher nicht existiert. Für den Fall, dass der **User** existiert und das Passwort korrekt ist, wird zuerst die aktuelle Session mit **invalidate()** entfernt. Damit löschen wir alle evtl. noch vorhandenen Daten eines anderen Nutzers, der sich evtl. nicht korrekt abgemeldet hat. Dann fügen wir dem **SessionContext** den neu angemeldeten User hinzu. Damit wissen wir später immer, welcher Nutzer zur Zeit angemeldet ist. Das **mapping.findForward(...)** kennen wir ja bereits. Falls das Passwort falsch ist oder ein Nutzer mit dem Namen nicht existiert, erzeugen wir eine neue Fehlermeldung und speichern die Fehler mit **saveErrors**. Anschließend wird der Benutzer mit **mapping.getInputForward()** zurück auf die Eingabeseite für diese Aktion geschickt.

Sicher ist euch aufgefallen, dass wir noch so etwas wie

```
logon.error=Wrong username and or password!
```

der Datei **MessageResources.properties** hinzufügen müssen. Dann haben wir alles um die **logon.jsp** und die Klassen **LogOnForm** und **LogOnAction** miteinander zu verbinden. Das tun wir in den Dateien **transitions.xml** und **struts-config.xml**. Dazu benötigen wir das Zustandsdiagramm (Abb. 8) und passen es an unsere vorgenommenen Bezeichnungen an.

Jeder Übergang ist entweder durch eine **Action** gekennzeichnet oder ein einfacher Link, so dass die Pfeilbeschriftung leer bleibt. Nun können wir mit Hilfe des Zustandsübergangendiagrammes (Abb. 9) die Übergänge in der **transitions.xml** definieren, da sie ein reines Abbild des Graphen darstellt.

```

<transition>
    <page>/welcome</page>
    <page>/LogOn</page>

```

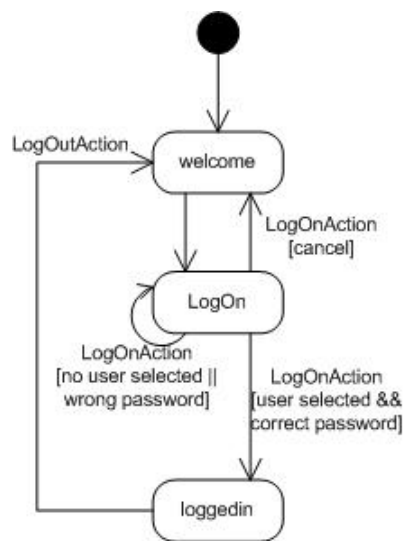


Abbildung 9: angepasstes Zustandsübergangsdiagramm

```

<page action="/LogOnAction" input="/LogOn">
  <forward>/welcome</forward>
</page>
<page>/loggedin</page>
<page action="/Logout"/>
  <page>/welcome</page>
</transition>

```

Damit wird der genaue Ablauf festgelegt, auf dem sich der Benutzer bewegen darf. Normalerweise besteht eine **transition** aus einer Abfolge von **pages**, welche Zustände aus unserem Zustandsübergangsdiagramm darstellen. Jedoch werden für die Erstellung der **struts-config.xml** einige Zusatzinformationen benötigt. Zum Beispiel werden **Actions** durch ein **action**-Attribut im **page**-Objekt gekennzeichnet. Ihnen kann zusätzlich eine **input**-Seite hinzugefügt werden, was nötig ist, wenn die **Action** mit einer **Form** gekoppelt ist und somit eine „Rücksprungadresse“ bei Fehlern gebraucht wird. Zu jeder **Action** können beliebig viele **Forwards** definiert werden, wobei die darauf folgende **page** ebenfalls als ein **Forward** gewertet wird.

Jedes Mal, nachdem wir Änderungen in der **transitions.xml** vorgenommen haben, sollten wir die **struts-config.xml** aktualisieren, damit die Daten konsistent bleiben. Dies geschieht durch einen Rechtsklick auf die **build.xml -> Run As -> Ant Build...** Dann im Tab „Targets“ den Prozess

„updateStrutsConfig“ auswählen (siehe Abb. 10) und mit „Run“ bestätigen²⁰

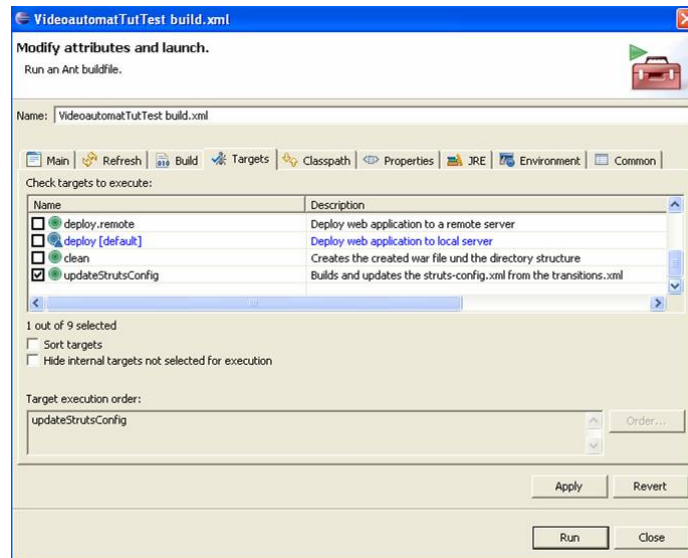


Abbildung 10: „updateStrutsConfig“ als Ant Build ausführen

Nun sollte die **struts-config.xml** wie in Abb. 11 aussehen.²¹

```

<!-- ***** Action Mapping Definitions -->
<action-mappings>
<action path="/welcome" forward="/pages/welcome.jsp" />
<action path="/LogOnAction"
  type="** fully qualified Java class name of the Action subclass **"
  name="** name of the form bean **"
  scope="** request_or_session **"
  validate="** true if the validate method of the ActionForm bean should be called **"
  input="/logon.do"
  roles="** list of security role names **"
  parameter="method **remove if no form-bean is used**">
  <forward name="**" path="/welcome.do" />
  <forward name="**" path="/loggedin.do" />
</action>
<action path="/LogOutAction"
  type="** fully qualified Java class name of the Action subclass **"
  name="** name of the form bean **"
  scope="** request_or_session **"
  validate="** true if the validate method of the ActionForm bean should be called **"
  input="null.do"
  roles="** list of security role names **"
  parameter="method **remove if no form-bean is used**">
  <forward name="**" path="/welcome.do" />
</action>
</action-mappings>

```

Abbildung 11: automatisch erzeugte **Action-Mappings**

Nach dieser Aktualisierung arbeiten wir in der aktualisierten **struts-config.xml** weiter: Wir informieren Struts, dass es eine neue **form-bean** gibt, der wir den Namen „LogOnForm“ geben.

```

<form-beans>
  <form-bean name="LogOnForm"
    type="videoautomat.logon.LogOnForm"/>
</form-beans>

```

²⁰Vorher alle anderen markierten Prozesse entfernen.

²¹Dass sich durch das automatische Aktualisieren die Formatierung verändert, lässt sich leider nicht ändern.

Daraufhin kommen wir endlich zu der in `logon.jsp` angegebenen Action `LogOnAction`. Wir müssen den automatisch erzeugten `Action`-Eintrag überarbeiten. Die Attribute sind dabei schon vorgegeben, sie müssen nur noch mit „Leben gefüllt“ werden.

```
<action-mappings>
.
.
  <action path="/loggedin" forward="/pages/loggedin.jsp"/>
  <action path="/LogOn" forward="/pages/logon.jsp"/>
  <action
    path="/LogOnAction"
    type="videoautomat.logon.LogOnAction"
    name="LogOnForm"
    scope="request"
    validate="true"
    input="/LogOn.do"
    parameter="method">
    <forward name="exit" path="/welcome.do"/>
    <forward name="loggedin" path="/loggedin.do"/>
  </action>
</action-mappings>
```

Wir legen für die Aktion den Pfad `/LogOnAction` an. Der Typ entspricht der gerade von uns erstellten Java Klasse. Mit `validate="true"` sorgen wir dafür, dass unsere `validate`-Methode auch wirklich aufgerufen wird. Als Parameter ist hier „konsistent“ zu der `logon.jsp` für unsere Buttons „method“ angegeben.²² In unserer Java Klasse `LogOnAction` haben wir mehrfach ein `mapping.findForward(...)` verwendet. Die benutzten Forwards definieren wir hier innerhalb unserer `LogOn`-Aktion.

Bis jetzt existiert noch keine Seite `loggedin.jsp`. Für den Anfang erzeugen wir einfach eine Kopie der Datei `welcome.jsp`. Danach fügen wir in der ursprünglichen `welcome.jsp` noch einen Link auf unseren neuen Prozess hinzu:

```
<br>
<html:link page="/LogOn.do">
  <bean:message key="videoautomat.logon"/>
</html:link>
```

²²Wir hatten dort `method` als `property` für den submit-Button angegeben.

und in der **MessageResources.properties** einen Eintrag:

```
videoautomat.logon=Login
```

1.10.4 LogOut

Benutzer sollen sich natürlich auch wieder vom System abmelden können. Die Zustandsübergangsdiagramme (Abb. 8 und Abb. 9) verdeutlichen dies und zeigen auch, von wo aus dem Benutzer die Möglichkeit dazu gegeben werden soll. In der **transitions.xml** haben wir die Eintragungen gemäß Zustandsübergangsdiagramm bereits in 1.10.3 vorgenommen.

Also fügen wir der **loggedin.jsp** noch

```
<html:link page="/Logout.do">
    <bean:message key="videoautomat.logout"/>
</html:link>
```

hinzu und ergänzen in der **MessageResources.properties** ein:

```
videoautomat.logout=Logout
```

Jetzt müssen wir eine passende Aktion implementieren und dann in der Datei **struts-config.xml** konfigurieren. Da die Aktion diesmal keine verschiedenen Buttons verarbeiten soll, sondern immer das Gleiche tut, reicht es, dass wir die Klasse von **Action**²³ ableiten.

```
package videoautomat.logon;
public class LogoutAction extends Action {
    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response) {
        new SessionContext(request).detachUser();
        request.getSession().invalidate();
        return mapping.findForward("exit");
    }
}
```

²³**org.apache.struts.action.Action**

Die Aktion meldet den aktuellen **User** ab und löscht die gerade aktive Sitzung, um dann ein `mapping.findForward("exit")` zurückzugeben. Zur Konfiguration in der `struts-config.xml` reicht folgende Änderung:

```
<action-mappings>
    .
    .
    <action
        path="/Logout"
        type="videoautomat.logon.LogOutAction"
        scope="request">
        <forward name="exit" path="/welcome.do"/>
    </action>
</action-mappings>
```

Jetzt können wir uns nach einem **deploy** das Ergebnis im Browser anschauen und ausprobieren.

1.10.5 Ein „kleiner“ Schönheitsfehler

Bisher wird an noch keiner Stelle im Login-Vorgang überprüft, ob der Benutzer auch das Recht hat, die `/loggedin.do` aufzurufen. Denn normalerweise sollte das nur ein User dürfen, der auch wirklich eingeloggt ist. Aufgrund der klar definierten Zustandsübergänge ist es zwar in diesem Fall nicht möglich, das Login zu übergehen und auf interne Seiten zu gelangen. Trotzdem sollten wir der Vollständigkeit halber schon hier die Rechtedefinition einführen, da sie später noch eine wichtigere Rolle spielen wird. Wir brauchen also eine Möglichkeit bestimmte Aktionen nur für bestimmte Nutzer zugänglich zu machen. Hierfür gibt es im Framework eine vorgefertigte Rechteverwaltung. Man kann Benutzern Rechte zuteilen und Aktionen so konfigurieren, dass nur noch Nutzer mit den angegebenen Rechten Zugriff darauf haben. Für Rechte gibt es das Interface **Capability** und die Klasse **CapabilityImpl**, die dieses Interface implementiert. Fügen wir also unserem Automatenbenutzer mit der Methode **setCapability** ein neues Recht **user** hinzu:

```
public AutomatUser(String user_ID, char[] passWd) {
    .
    .
    setCapability(new CapabilityImpl("user"));
}
```

Wie man sieht, erwartet der Konstruktor von **CapabilityImpl** den Namen für das Recht. Zusätzlich kann man einen **boolean** als zweiten Parameter angeben, ob dieses Recht gerade gewährt wird oder nicht. Jetzt müssen wir in der **struts-config.xml** unsere **loggedin**-Aktion wie folgt anpassen:

```
<action path="/loggedin"  
        forward="/pages/loggedin.jsp"  
        roles="user"/>
```

Damit können nur noch angemeldete Nutzer²⁴, die mindestens eine²⁵ der angegebenen Rollen/Rechte haben, die Aktion ausführen. Offensichtlich ändert sich für uns aber erst einmal nichts.

1.11 Das Geld

Für die Implementation des Leih- oder Rückgabeprozesses fehlt eine wichtige Grundlage: Die Verwaltung des Geldes. Diese soll im folgenden Kapitel geschaffen werden.

1.11.1 Die Währung

Das im Videoautomaten befindliche Geld kann nicht durch eine einzelne, numerische Variable repräsentiert werden. Es muss mit verschiedenen Münzen und Scheinen umgegangen werden und diese sind deshalb separat darzustellen. Ansonsten hätte das Programm keine Kenntnis von den im Automat befindlichen Geldgrößen und wäre nicht einmal in der Lage, Wechselgeld auszugeben.

Im Grunde genommen lassen sich Münzen oder Geldscheine bereits durch das Frameworkmodell der Kataloge und Bestände realisieren, indem im Katalog die möglichen Größen definiert werden und in einem zählenden Bestand die Anzahl der jeweiligen Größe vermerkt wird. Darüberhinaus existieren in WebPoint bereits Ableitungen von **CatalogImpl** und **CountingStockImpl**, die auf Währungseinheiten ausgerichtet sind. Das hat den Vorteil, dass alle Münzen und Scheine der jeweiligen Währung bereits im Katalog enthalten sind. Außerdem bietet ein solcher Währungskatalog die Möglichkeit, eine Geldsumme entsprechend formatiert auszugeben, was sich z. B. bei der Anzeige der Videopreise günstig auswirken wird.

²⁴Das heißt **User**, die dem **SessionContext** hinzugefügt wurden.

²⁵In der **struts-config.xml** kann man auch mehrere Rollen angeben (z. B. **roles="user, admin"**).

Ein auf das europäische Zahlungsmittel ausgerichteter Währungskatalog ist die Klasse **EUROCurrencyImpl**. Eine Instanz dieser Klasse wird als Grundlage für den Zahlungsverkehr am Automaten im Konstruktor von **VideoShop** erzeugt.

```
public class VideoShop extends Shop {
    .
    .
    public static final String C_CURRENCY = "CurrencyCatalog";
    public VideoShop() {
        .
        .
        addCatalog(new EUROCurrencyImpl(C_CURRENCY));
    }
    .
    .
    public static EUROCurrencyImpl getCurrency() {
        return (EUROCurrencyImpl) Shop.getTheShop()
            .getCatalog(C_CURRENCY);
    }
}
```

Ein Blick in das API der Klasse **EUROCurrencyImpl** verrät die im Katalog vorhandenen Geldgrößen. Es ließe sich nun die Menge der möglichen Münzen und Scheine durch das Entfernen von entsprechenden Katalogeinträgen einschränken. Vermutlich wird man beispielsweise auf 500€ Scheine verzichten wollen. In diesem fiktiven Beispiel wird der Einfachheit halber auf den vollständigen Katalog zurückgegriffen.

1.11.2 Der Geldbeutel

Analog zum Währungskatalog gibt es auch eine auf Zahlungsmittel ausgerichtete Implementation eines Bestandes, die Klasse **MoneyBagImpl**. Es handelt sich um eine Spezialisierung von **CountingStockImpl**, die das Interface **MoneyBag** implementiert. **MoneyBag** ist ein sogenanntes Markerinterface, das heißt, es definiert keinerlei Konstanten oder Methoden. Es dient lediglich der Markierung, dass dieser Bestand für Währungseinheiten bestimmt ist. Somit lässt sich ein solcher Geldbeutel genauso wie ein einfacher **CountingStock** behandeln.

Es wird für die Verwaltung des im Automaten befindlichen Geldes ein **MoneyBagImpl** im Konstruktor von **VideoShop** angelegt. Der zuvor angelegte Währungskatalog wird als

Parameter übergeben. Außerdem wird wie gehabt eine statische Rückgabemethode für den neu erzeugten Bestand implementiert.

```
public class VideoShop extends Shop {
    .
    public static final String MB_MONEY = "Money";
    public VideoShop() {
        .
        .
        addStock(
            new MoneyBagImpl(
                MB_MONEY,
                (EUROCurrencyImpl) getCatalog(C_CURRENCY));
        }
        .
        .
    public static MoneyBagImpl getMoneyBag() {
        return (MoneyBagImpl) Shop.getTheShop()
            .getStock(MB_MONEY);
    }
}
```

Zur Schaffung eines Grundkapitals wird, wie zuvor beim Videobestand und der Nutzerverwaltung, eine Methode der Klasse **MainPlugIn** hinzugefügt, in der dem Geldbeutel des Automaten ein paar Einträge zugefügt werden. Die neu definierte Methode wird in der **init**-Methode aufgerufen.

```
public class MainPlugIn implements PlugIn {
    public void init(ActionServlet servlet,
        ModuleConfig config) {
        .
        .
        initializeMoney();
    }
    .
    .
    public static void initializeMoney() {
        MoneyBagImpl mbi = (MoneyBagImpl)
```

```

        Shop.getTheShop().getStock(VideoShop.MB_MONEY);
        mbi.add(EUROCcurrencyImpl.CENT_STCK_10, 100, null);
        mbi.add(EUROCcurrencyImpl.CENT_STCK_20, 100, null);
        mbi.add(EUROCcurrencyImpl.CENT_STCK_50, 100, null);
        mbi.add(EUROCcurrencyImpl.EURO_STCK_1, 100, null);
        mbi.add(EUROCcurrencyImpl.EURO_STCK_2, 50, null);
        mbi.add(EUROCcurrencyImpl.EURO_SCHEIN_10, 100, null);
        mbi.add(EUROCcurrencyImpl.EURO_SCHEIN_20, 10, null);
    }
}

```

Die Namen der Katalogeinträge in **EUROCcurrencyImpl** sind, wie im Beispiel zu sehen ist, als Konstanten in derselben Klasse abrufbar.

1.11.3 Formatierung der Preise

Auf der Grundlage des Währungskatalogs ist es nun mehr möglich, die Anzeige des Videobestandes zu verbessern. Zur Erinnerung: Die Preise der Videos werden in Cent und ohne Währungssymbol angezeigt. Es bedarf lediglich eines kleinen Eingriffs in der Klasse **TEDVideoStock** um den Schönheitsfehler zu korrigieren. Durch Neudefinition der Methode **getCellRenderer(int nIndex)** lässt sich konkret auf die Formatierung der angezeigten Daten Einfluss nehmen. Hier wird für die Formatierung der Videopreise eine Instanz der Klasse **CurrencyRenderer** mittels des Währungskatalogs erzeugt. Eine solche nimmt ein **NumberValue** als Eingabe und liefert auf der Basis des jeweiligen Währungskatalogs die korrekt formatierte Darstellung des Wertes.

```

public TableCellRenderer getCellRenderer(int nIndex) {
    switch (nIndex) {
        case 1 : return new CurrencyRenderer(
                    VideoShop.getCurrency());
    }
    return super.getCellRenderer(nIndex);
}

```

Da wir aber keine €-Zeichen haben wollen, sondern die HTML-Codierung **€** müssen wir noch im Konstruktor von **VideoShop** die statische Methode **setHtmlOutput(true)** von **EUROCcurrencyImpl** aufrufen.

```

public VideoShop() {

```

```
.  
.   
    EUROCurrencyImpl.setHtmlOutput(true);  
}
```

Nach erfolgreichem **deploy** kann die Formatierung der Preise im Browser betrachtet werden.

1.12 Die Zeit

Bevor im nächsten Kapitel der Ausleihvorgang beschrieben wird, soll eine automateninterne Zeitrechnung implementiert werden, die es gestattet, zeitliche Abläufe zu simulieren.

1.12.1 Definition eines Timers

Beim Verleih einer Videokassette muss die Ausleihzeit abgespeichert werden, um beim Rückgabevorgang eine korrekte Abrechnung zu ermöglichen. Natürlich ließe sich dabei auf die Systemzeit zurückgreifen, jedoch wäre es in dieser Simulation wünschenswert, wenn im Zuge des Testens nicht jedes Mal die Systemuhr umgestellt werden muss, will man ein paar Tage überspringen.

Glücklicherweise bietet das Framework auch hierfür ein vorgefertigtes Konstrukt. So existieren zwei Interfaces **Time** und **Timer**, welche eine abstrakte Lösung für die Zeitverwaltung bieten. Dabei stellt **Timer** eine Art Zeitmanager und **Time** die Zeit selbst dar. Eine Implementation des Zeitmanagers ist **StepTimer**, eine Klasse für die Zeit **CalendarTime**. Diese beiden Klassen sollen im Videoautomaten zur Anwendung kommen.

Wird innerhalb der Anwendung die aktuelle Zeit benötigt oder ein Objekt muss auf gewisse Zeitereignisse reagieren, so kommt grundsätzlich nur der Zeitmanager zum Tragen. Dennoch wird ein **Time**-Objekt für die Konstruktion des Zeitmanagers benötigt. Das Zeitobjekt wird quasi im Zeitmanager verwaltet. Die meisten Anfragen an den Manager, z. B. die Frage nach dem aktuellen Datum, werden an das Zeitobjekt delegiert. Darüberhinaus bietet der Manager jedoch zusätzliche Funktionalitäten wie beispielsweise das Informieren über gewisse Zeitereignisse.

Die Initialisierung von **StepTimer** wird innerhalb des Konstruktors von **VideoShop** vorgenommen und über die Methode **setTimer(Timer t)** wird die neue Instanz zum globalen Zeitmanager erhoben.

```
public class VideoShop extends Shop {  
    .
```

```

    .
    public VideoShop() {
        .
        .
        setTimer(new StepTimer(new CalendarTime(
                                System.currentTimeMillis())));
    }
    .
    .
}

```

Dem Konstruktor von **CalendarTime** muss ein **long**-Wert übergeben werden, der die initiale Zeit repräsentiert. Hier wird die Systemzeit zur Initialisierung genutzt. Der Aufruf **System.currentTimeMillis()** liefert einen **long**-Wert, der die Anzahl der vergangenen Millisekunden seit dem 1. Januar 1970 0:00 Uhr GMT bis zum jetzigen Zeitpunkt repräsentiert.

1.12.2 Die Zeit weiterschalten

Auf der Basis des Zeitmanagers soll nun die Möglichkeit geschaffen werden, die Zeit in Tagesschritten vorzustellen.

Das manuelle Zeitfortschalten dient vor allem Testzwecken und soll deshalb nicht „direkt“ für Benutzer/Administratoren des Videoautomaten zu sehen sein. Auch dafür gibt es schon ein vorgefertigtes Konstrukt im Framework. Über die URL <http://127.0.0.1:8080/Videoautomat/shop/> gelangt man auf Shopseiten, die zum Testen gedacht sind. Dort findet man jetzt bereits Aktionen zum Laden und Speichern des aktuellen Warenbestandes. An dieser Stelle wollen wir auch das Zeitweiterschalten unterbringen. Die JSP-Seiten zum Shop finden wir direkt im Ordner **shop** und die **struts-config.xml** für den Shop im Ordner **WEB-INF/shop**.²⁶

Zum Zeitweiterschalten schreiben wir uns eine neue Klasse **GoAheadAction**, die von **Action** abgeleitet ist.

```

package videoautomat;
public class GoAheadAction extends Action {
    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,

```

²⁶Dadurch, dass der Shop seinen eigenen Unterordner und auch eine eigene **struts-config.xml** hat, ist es später sehr leicht möglich, die „Testseiten“ komplett zu entfernen.

```

        HttpServletRequest request,
        HttpServletResponse response){
    // den Timer weiterschalten
    Shop.getTheShop().getTimer().goAhead();
    return mapping.findForward("shop");
    }
}

```

Die vom Zeitmanager verwendete Methode **goAhead()** in der Aktion erhöht die Zeit um ein zuvor definiertes Intervall. Sofern noch kein Intervall definiert ist, wird das Standardintervall des internen Zeitobjektes, hier das **CalendarTime**-Objekt, genutzt. Das Standardintervall der Klasse **CalendarTime** entspricht genau 24h, so dass in diesem Fall keine Anpassungen vorgenommen werden müssen.

Jetzt müssen wir die Aktion noch in der **struts-config.xml**²⁷ konfigurieren. Dies geschieht analog zu unserer **LogOutAction** jedoch ohne den Umweg über die **transitions.xml**, da wir in diesem „Testbereich“ eine Manipulation ausschließen bzw. vernachlässigen können.

```

<action path="/goAhead" type="videoautomat.GoAheadAction">
    <forward name="shop" path="/shop.do"/>
</action>

```

Als nächstes fügen wir der Datei „**shop.jsp**“ noch einen Link auf unsere neue **Action** hinzu

```

<body bgcolor="#FFFFFF">
    <html:link page="/goAhead.do">
        <bean:message key="shop.goAhead"/>
    </html:link> <br>
    .
    .
</body>

```

und dazu passend in der **MessageResources.properties** die Zeile:

```
shop.goAhead=+1Day
```

²⁷ **Achtung:** Hier ist die **struts-config.xml** im Ordner **\WEB-INF\shop** gemeint!

Um später überprüfen zu können, ob die Weiterschaltung korrekt funktioniert bzw. um bei weiteren Tests die Übersicht über das aktuelle Datum zu behalten, muss die Zeit noch angezeigt werden.

```
<body bgcolor="#FFFFFF">
  <%= web.sale.Shop.getTheShop().getTimer().getTime() %> <br>
  .
  .
</body>
```

1.13 Die Leih-Kassette

Auf Grundlage des Zeitmanagers ist es nunmehr möglich, das Ausleihdatum einer Videokassette zu bestimmen. Dadurch ist man wiederum in der Lage, eine Klasse zu definieren, die die Verleihkassette repräsentiert.

Zur Erinnerung: die Klasse **AutomatUser** kapselt einen **StringStock**, in welchem die entliehenen Videos des Kunden gespeichert werden sollen. Im Gegensatz zum **CountingStock** sind die Bestandseinträge eines **StringStock** eigenständige Objekte mit individuellen Eigenschaften (z. B. die Ausleihzeit) und nicht nur Referenzen auf Katalogeinträge, versehen mit einer entsprechenden Anzahl. Einträge in einem solchen Bestand müssen das Interface **StockItem** implementieren. Entsprechend bedarf es einer Klasse für die entliehenen Videos, die die Schnittstelle erfüllt. Die neue Klasse **VideoCassette** erfüllt diese Bedingungen. Es handelt sich um eine Ableitung von **StockItemImpl**, der frameworkinternen Implementation von **StockItem**.

```
package videoautomat;
public class VideoCassette extends StockItemImpl {
    private long rentTime;
    public VideoCassette(String key) {
        super(key);
        rentTime = ((Date) Shop.getTheShop().getTimer()
                    .getTime()).getTime();
    }
}
```

Dem Konstruktor der neuen Klasse muss der **String**-Identifier des zugehörigen Katalogeintrags übergeben werden. Der **String** wird über den **super(String s)**-Aufruf

an den Konstruktor der Oberklasse delegiert. Außerdem wird die aktuelle Zeit über den globalen Zeitmanager abgerufen und einer **long**-Variable zugeordnet.

Hinweis: Damit die Typumwandlung (class cast) zu **Date** funktioniert, muss **java.util.Date** und nicht **sale.Date** importiert werden.

Abschließend werden noch zwei Methoden implementiert, eine für die Rückgabe der Tage, die das Video bereits entliehen ist und eine zur Ermittlung der bis dato entstandenen Kosten.

```
package videoautomat;
public class VideoCassette extends StockItemImpl {
    .
    .
    public int getDays() {
        long l = ((Date) Shop.getTheShop().getTimer()
                .getTime()).getTime() - rentTime;
        l /= 86400000;
        if(l<1) return 1;
        return (int) Math.ceil(l);
    }
    public NumberValue getCost() {
        return (NumberValue) MainPlugIn
                .RENT_VALUE_DAY.multiply(getDays());
    }
}
```

Bei der Berechnung der Leihstage wird zunächst die Ausleihzeit vom aktuellen Zeitpunkt abgezogen und die resultierende Differenz wird durch die Millisekundenanzahl eines Tages dividiert. Ist das Ergebnis kleiner eins, wird eins zurückgegeben andernfalls der Quotient. Das erklärt sich aus der Forderung des Videoverleihs, wonach bei der Ermittlung der Verleihkosten je angefangene 24h die Kosten eines Tages abgerechnet werden müssen.

In der Methode der Kostenberechnung wird auf eine Konstante zurückgegriffen, die noch in der Klasse MainPlugIn definiert werden muss. Es handelt sich um die Leihkosten pro Tag.

```
public class MainPlugIn implements PlugIn {
    public static NumberValue RENT_VALUE_DAY =
```

```

        new IntegerValue(200);
    .
    .
}

```

Damit sind alle Voraussetzungen erfüllt, um mit der Implementierung des Leihvorgangs zu beginnen.

1.14 Der Ausleihprozess

Im folgenden Kapitel soll der eigentliche Ausleihvorgang beschrieben werden.

Ausgehend vom Anmeldeprozess gelangt der Kunde zum Startzustand des Ausleihprozesses, in welchem die gewünschten Videos ausgewählt werden können. Wurde eine

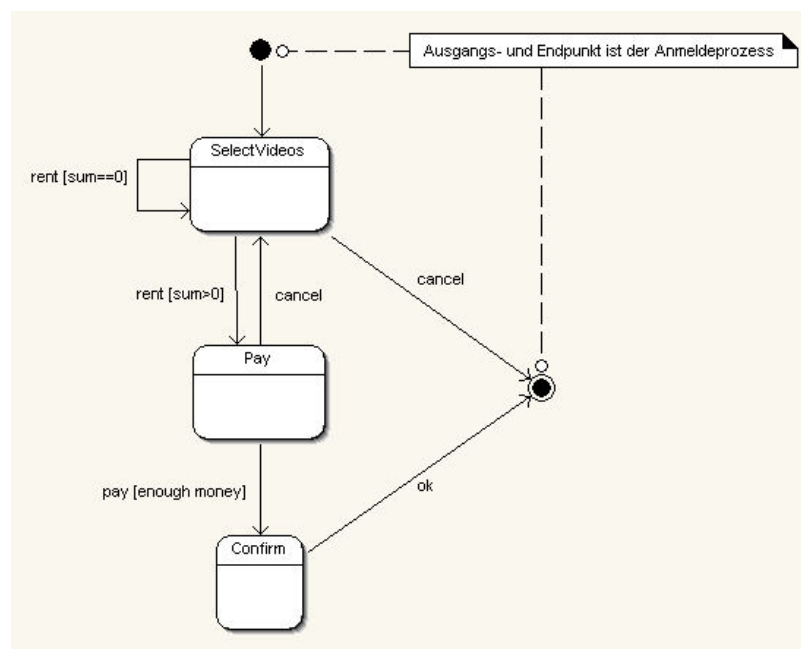


Abbildung 12: Videoautomat - Ausleihprozess

Auswahl getroffen und bestätigt, gelangt man in den Bezahlzustand, wo die entsprechende Geldsumme beglichen werden muss. Sind ausreichend Münzen und/oder Scheine eingezahlt worden, kann zum Folgezustand gewechselt werden. Vor dem Wechsel werden die Änderungen persistent gemacht. Im Zustand **Confirm** werden dann die gewählten Videos sowie etwaiges Wechselgeld angezeigt. Mit dem Bestätigen der Anzeige wechselt man zurück zum Anmeldeprozess. Befindet sich der Prozess im Bezahlzustand, kann zum

vorherigen Ausgangszustand zurück gewechselt werden. Die bereits eingeworfenen Münzen werden in diesem Fall wieder ausgegeben. Abbildung 12 verdeutlicht den Ablauf des Prozesses anhand eines Zustandsübergangsdiagramms. Mit unserem Wissen über den Prozessablauf passen wir dieses vereinfachte Zustandsübergangsdiagramm nun an (siehe Abb. 13).

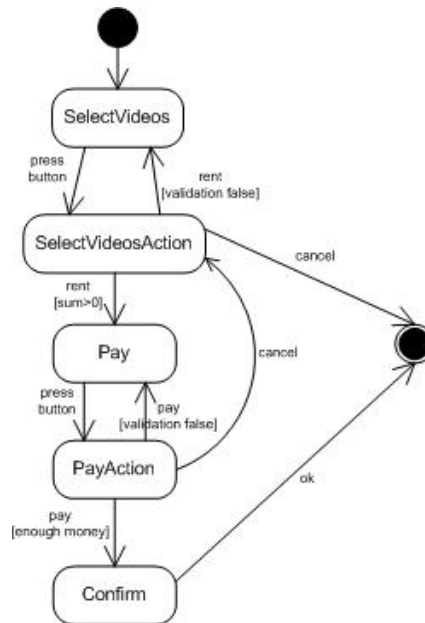


Abbildung 13: Videoautomat - angepasster Ausleihprozess

Genau wie bei der Anmeldung (9) können wir das Zustandsübergangsdiagramm 1:1 als neue **transition** in die **transitions.xml** umschreiben:

```

<transitions>
.
.
<transition>
<page>/loggedin</page>
<page>/SelectVideos</page>
<page action="/SelectVideosAction"
input="/SelectVideos">
<forward>/loggedin</forward>
</page>
<page>/pay</page>
<page action="/PayAction"
input="/pay">

```

```

<forward>/SelectVideos</forward>
</page>
<page>/confirm</page>
<page>/loggedin</page>
</transition>
</transitions>

```

Natürlich übernehmen wir die Veränderungen durch ein „updateStrutsConfig“²⁸ in die **struts-config.xml**.

In den folgenden Abschnitten werden wir die Zustände und Übergänge dieses Diagramms schrittweise umsetzen. Aber zunächst erstellen wir für diesen Prozess ein neues Paket **videoautomat.rent**. In diesem werden wir alle Java-Klassen zusammenfassen, die für die Umsetzung benötigt werden.

1.14.1 Select Videos

In dem Zustand „SelectVideos“ soll der Benutzer Videos zum Entleihen auswählen können.

WebPoint unterstützt uns an dieser Stelle mit dem Konzept des Zeit-Tabellen-Formulars. Ein **TwoTableForm** enthält eine linke und rechte Tabelle (siehe Abb. 14), die jeweils die Repräsentation einer frameworkinternen Datenstruktur darstellt. Zum Beispiel könnte die

| | Name | Price | Count |
|----------------------------------|---------|---------|-------|
| <input type="radio"/> | Video-0 | 30,00 € | 5 |
| <input type="radio"/> | Video-1 | 30,00 € | 3 |
| <input type="radio"/> | Video-2 | 30,00 € | 5 |
| <input type="radio"/> | Video-3 | 30,00 € | 5 |
| <input checked="" type="radio"/> | Video-4 | 30,00 € | 4 |
| <input type="radio"/> | Video-5 | 30,00 € | 5 |
| <input type="radio"/> | Video-6 | 30,00 € | 5 |
| <input type="radio"/> | Video-7 | 30,00 € | 5 |
| <input type="radio"/> | Video-8 | 30,00 € | 5 |
| <input type="radio"/> | Video-9 | 30,00 € | 5 |

| | Name | Count |
|-----------------------|---------|-------|
| <input type="radio"/> | Video-1 | 2 |
| <input type="radio"/> | Video-4 | 1 |

Abbildung 14: Zwei-Tabellen-Formular

linke Seite einen Bestand B1 und die rechte Seite einen anderen Bestand B2 darstellen. Aufgrund der vorhandenen Verschiebebuttons in einem derartigen Tabellenformular ist der

²⁸Wie beim letzten Mal durch
build.xml -> Run As -> Ant Build... -> updateStrutsConfig

Anwender in der Lage, Einträge zwischen B1 und B2 hin- und herzuschieben. Das Verschieben wird gänzlich vom Formular und den beteiligten Datenstrukturen geleistet. Der Programmierer braucht sich um nichts weiter zu kümmern.

Beim **LogOnForm** brauchten wir drei Komponenten:

- eine JSP für die Anzeige
- ein **ActionForm** für die Daten
- und eine **Action**, die für die Verarbeitung zuständig ist

Das ist hier prinzipiell nicht anders. Beginnen wir diesmal mit dem **ActionForm**. Dazu erstellen wir eine neue Klasse **SelectVideosForm**, die wir von **TwoTableForm**²⁹ ableiten.

```
package videoautomat.rent;
public class SelectVideosForm extends TwoTableForm {
    public void reset(ActionMapping mapping,
                     HttpServletRequest request) {
    }
}
```

In der **reset**-Methode müssen wir unserem Formular mitteilen, welche Datenstrukturen dargestellt werden sollen. Dazu bietet das **TwoTableForm** bereits diverse vorgefertigte **reset**-Methoden, die nahezu alle Kombinationen³⁰ der unterschiedlichen Datenstrukturen berücksichtigen und ein entsprechendes Zwei-Tabellen-Formular initialisieren. Optional kann bei der Initialisierung dieser auf Transaktionen ausgerichteten Formulare ein Datenkorb übergeben werden.

An dieser Stelle soll kurz auf die Funktionsweise eines Datenkorbs eingegangen werden. Nehmen wir an, der Kunde wählt verschiedene Videos aus. Die gewählten Videos werden in den Leihbestand des Kunden aufgenommen. Einen Moment später überlegt er sich die Sache noch einmal und kommt zu dem Schluss, dass er das Geld doch lieber nicht dafür ausgeben und seine Zeit anders nutzen will. Der Leihvorgang wird abgebrochen. Die Videos müssen nun wieder aus dem virtuellen Bestand des Kunden entfernt und in den Bestand des Automaten aufgenommen werden, aber welche? Möglicherweise existieren bereits entliehene Videos im Bestand des Kunden, die nicht entfernt werden sollen.

²⁹`web.data.stdforms.TwoTableForm`

³⁰Detailliertere Informationen zu den vorhandenen **reset**-Methoden findet man in der JavaDoc zu `WebPoint`.

Ein Datenkorb hilft hierbei als eine Art Puffer. Wird beim Entfernen oder Hinzufügen von Einträgen einer Datenstruktur ein Datenkorb übergeben, zeichnet der Datenkorb die Aktion auf. Später kann ein **rollback** ausgeführt werden, welches sämtliche protokollierte Aktionen rückgängig macht, oder es erfolgt ein **commit**, um die Änderungen dauerhaft zu machen. Wird beim Hinzufügen oder Entfernen statt des Datenkorbs **null** übergeben, werden die gemachten Änderungen sofort dauerhaft gemacht.

In der **reset**-Methode der Klasse **SelectVideosForm** rufen wir jetzt die entsprechende **reset**-Methode des **TwoTableForm** für einen **CountingStock** auf der linken und einen **DataBasket** auf der rechten Seite auf. Die Videos sollen vorläufig nur dem Datenkorb und nicht dem Bestand des Kunden hinzugefügt werden. Die Komparatoren dienen der jeweiligen Ordnung der Einträge der rechten und linken Tabelle. Ebenso können **TableEntryDescriptor**-Objekte für beide Seiten übergeben werden. Wird an Stelle eines TEDs **null** übergeben, wird der Standard-TED der betreffenden Datenstruktur benutzt. Die boolsche Variable bestimmt, ob nicht vorhandene Elemente des **CountingStock** angezeigt werden sollen. Außerdem kann noch eine **MoveStrategy** übergeben werden, die das Verhalten der Verschiebe-Buttons implementiert. Wird als **MoveStrategy null** übergeben, wird die Standard-**MoveStrategy** für beteiligte Datenstrukturen gesetzt.

```
public void reset(ActionMapping mapping,
                 HttpServletRequest request) {
    SessionContext sc = new SessionContext(request);
    reset(VideoShop.getVideoStock(),
        sc.getBasket(),
        null,
        null,
        null,
        false,
        new videoautomat.TEDVideoStock(),
        new DefaultCountingStockDBETableEntryDescriptor(),
        null);
}
```

An dieser Stelle noch ein paar Worte zum verwendeten **SessionContext**. Das Rahmenwerk WebPoint bietet über den **SessionContext** vereinfachten Zugriff auf einige Variablen der Session. Wir haben den Kontext z. B. beim Anmeldeprozess genutzt, um in der Session zu speichern, welcher Nutzer gerade angemeldet ist. Darüberhinaus kann man

im **SessionContext** genau einen Datenkorb speichern. Mit **getBasket** kann man sich diesen zurückliefern lassen. Für den Fall, dass die Session derzeit noch keinen Datenkorb enthält, fügt **getBasket** ihr selbstständig einen neuen hinzu. Wenn dieses Verhalten nicht erwünscht sein sollte, kann man es mit **getBasket (false)** verhindern. Wie gesagt, man kann über den **SessionContext** nur genau einen Datenkorb speichern. Es kommt jedoch nicht selten vor, dass man für eine Anwendung innerhalb einer Session mehrere benötigt. Abhilfe schafft hier die Definition von Sub-Datenkörben. Durch die Methode **setCurrentSubBasket (String s)** kann man einen solchen bestimmen. Ein Rollback oder Commit lässt sich begrenzt auf einen Sub-Datenkorb ausführen.

```
public void reset (ActionMapping mapping,
                  HttpServletRequest request) {
    SessionContext sc = new SessionContext (request);
    sc.getBasket ().setCurrentSubBasket (Constants.SUB_SHOP_VIDEO);
    .
    .
}
```

Da wir die Namen der Sub-Datenkörbe mehrfach benötigen, ist es sinnvoll, sich diese als statische Konstanten in einer Klasse **Constants** zu definieren. Um Namenskollisionen zu vermeiden, beginnen wir jede Konstante mit dem Namen des Paketes. Der Quelltext von **Constants** sieht dann so aus:

```
package videoautomat.rent;
public class Constants {
    protected static final String PACKAGE = "videoautomat.rent";
    public static final String SUB_SHOP_VIDEO =
        PACKAGE + ".videos_cs";
}
```

Damit hätten wir unser Formular fertig. Was noch fehlt sind die JSP und die **Action**. Schauen wir uns zunächst die JSP an: Dazu erstellen wir eine Datei **SelectVideos.jsp** im Ordner **/pages** mit folgendem Inhalt:

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<%@ taglib uri="/tags/webpoint" prefix="wp" %>
```

```
<html:html>
<head>
  <title><bean:message key="videoautomat.caption"/> -
    <bean:message key="rent.choose.videos"/></title>
  <html:base/>
</head>
<body bgcolor="#FFFFFF">
  <html:errors/>
  <br>

  <html:form action="/SelectVideosAction">
    <wp:TwoTableForm />
    <html:submit property="method">
      <bean:message key="videoautomat.rent"/>
    </html:submit>
    <html:submit property="method">
      <bean:message key="videoautomat.cancel"/>
    </html:submit>
  </html:form>

</body>
</html:html>
```

Da wir ein HTML-Formular erzeugen wollen, nutzen wir wieder den `html:form`-Tag. Für die Darstellung des Zwei-Tabellen-Formulars ist `<wp:TwoTableForm/>` zuständig. Dieser Tag benötigt keine weiteren Parameter, da er sich alle Informationen, die er zur Darstellung braucht, aus der Konfiguration (in der `struts-config.xml`) des umgebenden Form-Tags holt. Dort wird später unser `SelectVideosForm` mit angegeben, was ja bereits mit dem Aufruf der `reset`-Methode initialisiert wird. Der Tag kann also nur richtig arbeiten, wenn ein entsprechendes `reset` aufgerufen wurde. Auf den `<html:errors/>` sollte an dieser Stelle nie verzichtet werden, auch wenn man selbst keine Fehler auf der Seite ausgeben möchte. Das Zwei-Tabellen-Formular benötigt eine Fehlerausgabe, um dem Benutzer mitzuteilen, wenn ein von ihm ausgewähltes Video nicht mehr vorhanden ist. Wenn wir uns das Zustandsübergangsdiagramm (Abb. 12) anschauen, sehen wir, dass aus dem Zustand `SelectVideos` Transitionen mit den Namen „cancel“ und „rent“ weggehen. Wenn man dieses mit der `SelectVideos.jsp` vergleicht, stellt

man fest, dass diese durch Buttons realisiert sind. Für die Beschriftungen benötigen wir noch die Zeilen

```
rent.choose.videos=Choose your videos!
videoautomat.rent=Rent
```

in der Datei **MessageResources.properties**.

Es reicht natürlich nicht aus, die Buttons in der JSP anzuzeigen. Wir müssen noch die Funktion realisieren. Damit kommen wir zu der dritten Komponente: der **Action**. Wir haben bisher zwei Klassen kennengelernt, von denen wir Aktionen ableiten können. Dieses wären:

- **Action** - für allgemeine Aktionen und
- **LookupDispatchAction** - für Aktionen, die auf verschiedene Buttons reagieren sollen

Da wir an dieser Stelle auf mindestens zwei Buttons, nämlich „cancel“ und „rent“ reagieren wollen, brauchen wir eine **LookupDispatchAction**. Allerdings, wenn man sich das in Abb. 14 abgebildete Formular anschaut, gibt es dort auch noch die Buttons, um Einträge zwischen den beiden Tabellen hin und her zu bewegen. Deshalb leiten wir die neue Klasse **SelectVideosAction** von **TwoTableAction** ab. **TwoTableAction** ist eine **LookupDispatchAction**, die bereits die Verschiebe-Buttons implementiert. Wir brauchen uns also nur noch um die Verarbeitung von „rent“ und „cancel“ kümmern. Dazu müssen wir wieder die Methode **getKeyMethodMap** überschreiben.

```
package videoautomat.rent;
public class SelectVideosAction extends TwoTableAction {
    protected Map getKeyMethodMap() {
        Map map = super.getKeyMethodMap();
        map.put("videoautomat.rent", "rent");
        map.put("videoautomat.cancel", "cancel");
        return map;
    }
}
```

An dieser Stelle ist es wichtig, dass wir die **getKeyMethodMap** erweitern, denn die bestehenden Einträge müssen erhalten bleiben, um die Funktionalität der Verschiebe-Buttons zu gewährleisten. Als nächstes schreiben wir die Methode für die Funktion des „cancel“-Buttons.

```

public ActionForward cancel(ActionMapping mapping,
                           ActionForm form,
                           HttpServletRequest request,
                           HttpServletResponse response) {
    SelectVideosForm selectVideos = (SelectVideosForm) form;
    DataBasket db = (DataBasket) selectVideos.getRightTableSource();
    db.setCurrentSubBasket(Constants.SUB_SHOP_VIDEO);
    db.rollbackCurrentSubBasket();
    return mapping.findForward("cancel");
}

```

Hier wird mit **rollbackCurrentSubBasket** auf dem aktuellen Sub-Basket ein Roll-back durchgeführt. Dazu ist es wichtig, dass vorher auch der richtige Sub-Basket gesetzt wurde!

Jetzt kann die Aktion für den „pay“-Button definiert werden. Für den Fall, dass die Summe Null ist, bleiben wir im Zustand **SelectVideos**, ansonsten wechseln wir in den Zustand **Pay**. (vergleiche Abb. 12)

```

public class SelectVideosAction extends TwoTableAction {
    .
    .
    public ActionForward rent(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response) {
        final SelectVideosForm selectVideos =
            (SelectVideosForm) form;
        final DataBasket db =
            (DataBasket) selectVideos.getRightTableSource();
        db.setCurrentSubBasket(Constants.SUB_SHOP_VIDEO);

        NumberValue nv_sum = (NumberValue) db.sumSubBasket(
            Constants.SUB_SHOP_VIDEO, null, new BasketEntryValue() {
                public Value getEntryValue(DataBasketEntry dbe) {
                    try {
                        CatalogItem ci = VideoShop.getVideoCatalog().get(
                            dbe.getSecondaryKey(), null, false);
                        int count = ((Integer) dbe.getValue()).intValue();
                    }
                }
            });
    }
}

```

```

        return ((QuoteValue) ci.getValue()).getOffer()
            .multiply(count);
    } catch (VetoException e) {
        e.printStackTrace();
        db.rollback();
    }
    return null;
}
}, new IntegerValue(0));

if (nv_sum.isAddZero()) {
    return mapping.getInputForward();
} else {
    selectVideos.setSum(nv_sum);
    return mapping.findForward("pay");
}
}
}

```

Die Methode **sumBasket** für das Aufsummieren des Sub-Datenkorbs erwartet den Namen desselben. Außerdem bedarf es eines **DataBasketCondition**-Objekts, sofern nur gewisse Einträge des Datenkorbs zu untersuchen sind. Sollen alle Einträge berücksichtigt werden, wie in diesem Fall, kann **null** übergeben werden. Ferner braucht die Methode ein **BasketEntryValue**-Objekt, das über die Methode **getEntryValue(DataBasketEntry dbe)** den Wert der Datenkorbeinträge liefert, der aufsummiert werden soll. In diesem Fall wird die Implementierung des Interface **BasketEntryValue** gleich im Methodenaufruf vorgenommen. Als letzten Parameter erwartet die Methode für das Aufsummieren des Sub-Datenkorbs einen Startwert, von dem ausgegangen werden soll. In diesem Fall ist es ein **IntegerValue**, das den Wert Null repräsentiert.

Damit wir später erneut auf die berechnete Summe zugreifen können, speichern wir uns diese im **SelectVideosForm**. Dazu müssen wir dieses noch etwas erweitern:

```

public class SelectVideosForm extends TwoTableForm {
    private NumberValue nv_sum;
    .
    .
    public NumberValue getSum() {

```

```

        return nv_sum;
    }
    public void setSum(NumberValue nv) {
        nv_sum = nv;
    }
}

```

Jetzt haben wir alle drei Komponenten und können diese in der **struts-config.xml** miteinander verbinden. Dazu definieren wir zuerst unser Formular:

```

<form-beans>
    .
    .
    <form-bean name="SelectVideosForm"
        type="videoautomat.rent.SelectVideosForm"/>
</form-beans>

```

dann eine Action als Forward auf unsere JSP.

```

<action-mappings>
    <action path="/SelectVideos"
        forward="/pages/SelectVideos.jsp"
        roles="user"
        parameter="method"/>
    .
    .
</action-mappings>

```

An dieser Stelle muss als Parameter, das gleiche angegeben werden, wie bei der nachfolgenden **Action**. Der **TwoTableForm**-Tag erwartet diese Information, um für die Verschiebe-Buttons als **property** genau das angeben zu können, wie für die von uns manuell erstellten Buttons. Zum Schluss noch die eigentliche **Action**, die wir ja nur noch anpassen müssen:

```

<action-mappings>
    .
    .
    <action path="/SelectVideosAction"
        type="videoautomat.rent.SelectVideosAction"
        name="SelectVideosForm"

```

```

        scope="session"
        validate="true"
        input="/SelectVideos.do"
        roles="user"
        parameter="method">
        <forward name="cancel" path="/loggedin.do"/>
        <forward name="pay" path="/pay.do"/>
    </action>
</action-mappings>

```

Man beachte: Hier haben wir bei allen Aktionen **roles="user"** mit angegeben, damit nur eingeloggte Nutzer darauf zugreifen können. Wir haben außerdem auch schon das entsprechende **forward** für **pay** mit angegeben, welches wir im nächsten Abschnitt erstellen.

Jetzt brauchen wir noch einen Link auf der Seite **loggedin.jsp**, um zu unserem neuen „Prozess“ zu gelangen.

```

    .
    <html:link page="/SelectVideos.do">
        <bean:message key="videoautomat.rent"/>
    </html:link>
</body>
</html:html>

```

1.14.2 Pay

Die Auswahl der Videos kann jetzt getroffen werden. Die erforderliche Summe soll dann im Bezahlzustand durch den Kunden beglichen werden.

Da für die Anwendung kein realer Automat und damit auch kein Erkennungsmechanismus für eingeworfene Münzen zur Verfügung steht, muss bei der Bezahlung etwas getrickst werden. Der Einwurf des Geldes wird durch ein Zwei-Tabellen-Formular mit dem Währungskatalog als Quelle und einem Geldbeutel als Ziel simuliert.

Dazu erstellen wir und die Klasse **PayForm**:

```

package videoautomat.rent;
public class PayForm extends TwoTableForm {
    .
    .
}

```

Bevor wir das Formular in der **reset**-Methode entsprechend initialisieren, schreiben wir uns einen Komparator für die Geldeinträge. Bei den Tabellenformularen werden, sofern bei der Initialisierung kein Komparator übergeben wird, die Einträge standardmäßig nach ihrem Namen sortiert. Bei den Geldeinträgen ist diese Art der Anordnung jedoch irritierend, da beispielsweise auf 1-Cent gleich 1-Euro folgt. Wünschenswert wäre eine Anordnung nach den Werten. Folgende Implementierung des Interface Comparator leistet dieses:

```
package videoautomat;

public class ComparatorCurrency implements Comparator,
                                           Serializable {

    public ComparatorCurrency() {
    }

    public int compare(Object arg0, Object arg1) {
        if (arg0 instanceof CatalogItem) {
            return ((CatalogItem) arg0).getValue().compareTo(
                ((CatalogItem) arg1).getValue());
        }
        if (arg0 instanceof CountingStockTableModel.Record) {
            Value v1 =
                ((CountingStockTableModel.Record) arg0)
                    .getDescriptor()
                    .getValue();
            Value v2 =
                ((CountingStockTableModel.Record) arg1)
                    .getDescriptor()
                    .getValue();
            return v1.compareTo(v2);
        }
        return 0;
    }
}
```

Die zu implementierende Methode **compare(Object o1, Object o2)** untersucht, ob es sich bei den zu vergleichenden Objekten um Tabelleneinträge eines Katalogs (Währungskatalog) oder eines zählenden Bestands (Geldbeutel) handelt, und delegiert den Vergleich an die zu untersuchenden Werte. Wichtig ist, dass der Komparator das Interface **Serializable** implementiert. Damit signalisiert man Speicherfähigkeit des Objektes.

Jetzt lässt sich das Formular für den Geldtransfer in der Methode **reset** initialisieren.

Dafür wird zusätzlich eine Variable des Typs **MoneyBagImpl** deklariert. Der deklarierte Geldbeutel wird an Stelle des Automaten-Geldbeutels verwendet, da es gewiss nicht wünschenswert ist, dass der Kunde die gesamte Barschaft im Automaten zu Gesicht bekommt.

```
public class PayForm extends TwoTableForm {
    private MoneyBagImpl mb_temp =
        new MoneyBagImpl("mb_user", VideoShop.getCurrency());

    public void reset(ActionMapping mapping,
        HttpServletRequest request) {
        SessionContext sc = new SessionContext(request);
        sc.getBasket()
            .setCurrentSubBasket(Constants.SUB_TMP_MONEY);
        reset(VideoShop.getCurrency(),
            mb_temp,
            sc.getBasket(),
            new ComparatorCurrency(),
            new ComparatorCurrency(),
            false,
            null,
            null,
            null);
    }

    public Value moneyThrownIn() {
        getDataBasket()
            .setCurrentSubBasket(Constants.SUB_TMP_MONEY);
        return mb_temp.sumStock(getDataBasket(),
            new CatalogItemValue(),
            new IntegerValue(0));
    }

    public MoneyBagImpl getMb_temp(){
        return mb_temp;
    }
}
```

Wie zu sehen ist, wird der Währungskatalog und der neu initialisierte Geldbeutel sowie

der Datenkorb des Prozesses für die Erzeugung des Formulars an die passende **reset**-Methode übergeben. Außerdem haben wir unserem Formular noch die Methode **moneyThrownIn** spendiert, die bestimmt, wie viel Geld schon in den Automaten geworfen wurde. Wir müssen die verwendete Konstante natürlich noch definieren.

```
package videoautomat.rent;
public class Constants {
    .
    .
    public static final String SUB_TMP_MONEY = PACKAGE
                                                + ".money_temp";
}
```

Als nächstes erstellen wir eine Datei **pay.jsp** für die Anzeige:

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<%@ taglib uri="/tags/webpoint" prefix="wp" %>

<html:html>
<head>
    <title><bean:message key="videoautomat.caption"/> -
        <bean:message key="pay.throw.money"/></title>
    <html:base/>
</head>
<body bgcolor="#FFFFFF">
    <html:errors/>
    <html:form action="/PayAction">
        <wp:TwoTableForm />
        <html:submit property="method">
            <bean:message key="videoautomat.pay"/>
        </html:submit>
        <html:submit property="method">
            <bean:message key="videoautomat.cancel"/>
        </html:submit>
    </html:form>
</body>
</html:html>
```

Auch hier wurden die Auslöser für die Transitionen als Button realisiert. Was dem Benutzer noch fehlt, ist eine Anzeige, wie viel er zu bezahlen hat. Dazu erweitern wir unser **SelectVideosForm** um folgende Methode:

```
public String getSumHtml() {
    return VideoShop.getCurrency().toString(getSum());
}
```

Diese Methode gibt uns den formatierten Betrag als **String** zurück. Jenen können wir dann auf unserer **pay.jsp** mit

```
.
<html:errors/>
<bean:message key="pay.to.pay"/>
<bean:write name="SelectVideosForm"
    property="sumHtml"
    filter="false"/>
.
```

anzeigen. Das **filter="false"** gibt dabei an, dass der **String** nicht nach HTML umcodiert werden soll, denn er entspricht bereits der HTML-Codierung. Für die Textmeldungen brauchen wir noch die Einträge

```
videoautomat.pay=Pay
pay.throw.money=Throw the money in the slot, please.
pay.to.pay=You have to pay:
```

in der **MessageResources.properties**. Jetzt haben wir ein **Form**-Objekt und eine JSP. Was noch fehlt, ist die passende **Action**:

```
package videoautomat.rent;
public class PayAction extends TwoTableAction {
    protected Map getKeyMethodMap() {
        Map map = super.getKeyMethodMap();
        map.put("videoautomat.pay", "pay");
        map.put("videoautomat.cancel", "cancel");
        return map;
    }

    public ActionForward cancel(ActionMapping mapping,
```

```

        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response) {
    PayForm payForm = (PayForm) form;
    DataBasket db = (DataBasket) payForm.getDataBasket();
    db.setCurrentSubBasket(Constants.SUB_TMP_MONEY);
    db.rollbackCurrentSubBasket();
    return mapping.findForward("cancel");
}

public ActionForward pay(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response) {
    PayForm payForm = (PayForm) form;
    SelectVideosForm selectVideos = (SelectVideosForm) request
        .getSession().getAttribute("SelectVideosForm");

    if (payForm.moneyThrownIn()
        .compareTo(selectVideos.getSum())>=0) {
    } else {
        return mapping.getInputForward();
    }
}
}

```

Analog zu der **SelectVideosAction** wurden hier die Methoden **getKeyMethodMap** und **cancel** implementiert. Die Methode **pay** ist hier nur angedeutet und wird gleich sinnvoll ergänzt. Da wir gemäß Zustandsübergangsdiagramm (Abb. 12) für den **rent**-Button unterscheiden sollen, ob der bereits gezahlte Betrag größer oder gleich der zu zahlenden Summe ist, brauchen wir Zugriff auf diesen. Dazu benötigen wir unser **SelectVideosForm**. Weil wir später bei der Konfiguration **scope=session** angegeben werden, wird das Formular unter dem vereinbarten Namen in der Session gespeichert und wir können mit

```

        (SelectVideosForm) request
        .getSession().getAttribute("SelectVideosForm");

```

darauf zugreifen. Sollte der Nutzer noch nicht genug Geld eingeworfen haben, wird er

zurück zur Eingabeseite geleitet. Was passiert, wenn der Nutzer ausreichend Geld in den Automaten geworfen hat, implementieren wir schrittweise: Zunächst erfolgt lediglich das Hinzufügen der Videokassetten zum Bestand des Kunden. Dazu muss ein neuer Sub-Datenkorb definiert und gesetzt werden, der nur die Transaktionen in den Bestand des Kunden protokolliert. Anschließend iteriert man über alle Einträge des Sub-Datenkorbs, der die vorerst entfernten Videos des Automatenbestandes enthält. Mit Hilfe der Namen der vom **Iterator** gelieferten Einträge erzeugt man entsprechende Instanzen der Klasse **VideoCassette** und fügt sie mit Hilfe des Datenkorbs dem Bestand des Kunden hinzu.

```

.
if (payForm.moneyThrownIn().compareTo(selectVideos.getSum())>=0) {
    SessionContext sc = new SessionContext(request);
    // first add new rent-cassettes to the user`s stock
    sc.getBasket().setCurrentSubBasket(Constants.SUB_USER_VIDEO);
    StoringStock ss_user =
        ((AutomatUser) sc.getUser()).getVideoStock();
    Iterator i = sc.getBasket().subBasketIterator(
        Constants.SUB_SHOP_VIDEO,
        DataBasketConditionImpl.ALL_ENTRIES);
    while (i.hasNext()) {
        CountingStockItemDBEntry dbe =
            (CountingStockItemDBEntry) i.next();
        int count = ((Integer) dbe.getValue()).intValue();
        for (int a=0; a<count; a++) {
            VideoCassette vc = new VideoCassette(dbe.getSecondaryKey());
            ss_user.add(vc, sc.getBasket());
        }
    }
} else {
.
.

```

Dabei darf die verwendete Konstante **SUB_USER_VIDEO** nicht vergessen werden:

```

package videoautomat.rent;
public class Constants {
.
.

```

```

public static final String SUB_USER_VIDEO = PACKAGE
                                         + ".video_ss";
public static final String SUB_SHOP_MONEY = PACKAGE
                                         + ".money_shop";
}

```

Die zweite Konstante, die wir hier definieren benötigen wir für den nächsten Abschnitt. Als nächstes muss berechnet werden, wie viel Wechselgeld der Kunde erhält. Anschließend soll das Geld aus dem Geldbeutel des Prozesses in den des Automaten verschoben werden:

```

while (i.hasNext()) {
    CountingStockItemDBEntry dbe =
        (CountingStockItemDBEntry) i.next();
    int count = ((Integer) dbe.getValue()).intValue();
    for (int a=0; a<count; a++) {
        VideoCassette vc = new VideoCassette(dbe.getSecondaryKey());
        ss_user.add(vc, sc.getBasket());
    }
}
// calculate the change
NumberValue nv = (NumberValue)payForm.moneyThrownIn()
                 .subtract(selectVideos.getSum());
// put the content of the temporar moneybag to the shop's one
sc.getBasket().setCurrentSubBasket(Constants.SUB_SHOP_MONEY);
VideoShop.getMoneyBag().addStock(payForm.getMb_temp(),
                                  sc.getBasket(), true);

// get the change
try {
    VideoShop.getMoneyBag().transferMoney(
        payForm.getMb_temp(), sc.getBasket(), nv);
} catch (NotEnoughMoneyException e) {
    sc.getBasket().rollbackSubBasket(Constants.SUB_USER_VIDEO);
    sc.getBasket().rollbackSubBasket(Constants.SUB_TMP_MONEY);
    sc.getBasket().rollbackSubBasket(Constants.SUB_SHOP_MONEY);
    ActionMessages messages = new ActionMessages();
    messages.add("change", new ActionMessage("pay.no.change"));
    saveErrors(request, messages);
    return mapping.getInputForward();
}

```

```

    }
    return mapping.findForward("confirm");
} else {
    return mapping.getInputForward();
}
.
.

```

Das Verschieben der Einträge aus dem Geldbeutel des Prozesses in den des Automaten vollzieht sich durch die Methode **addStock(Stock st, DataBasket db, boolean b)**, wobei der übergebene boolsche Parameter darüber entscheidet, ob die Einträge des übergebenen Bestandes gleichzeitig aus diesem entfernt werden sollen, so wie in diesem Fall, oder nicht.

Für den nächsten Schritt wird ein Algorithmus benötigt, der für einen gegebenen Wert die passenden Geldgrößen (das Wechselgeld) aus dem Geldbeutel des Automaten herausgibt. Dafür existiert die Methode **transferMoney(MoneyBag mb, DataBasket db, NumberValue nv)** in der Klasse **MoneyBagImpl**. Die Methode verschiebt dem übergebenen Wert entsprechende Einträge des Objektes in den übergebenen Geldbeutel mittels des Datenkorbs. Gibt es keine Kombination der vorhandenen Einträge, die dem Wert entspricht (existiert kein Wechselgeld), oder es ist nicht genügend Geld vorhanden, wird eine **NotEnoughMoneyException** geworfen.

Auf Basis dieser Methode lässt sich das Wechselgeld zurückgeben. Ist nicht genug bzw. kein passendes Wechselgeld verfügbar, so werden die bisher in der Aktion ausgeführten Transaktionen durch Rollbacks der entsprechenden Sub-Datenkörbe rückgängig gemacht und der Prozess wechselt zurück zum Bezahlzustand, wo dem Benutzer eine Fehlermeldung angezeigt wird. Diese müssen wir noch in der **MessageResources.properties** definieren.

```

pay.no.change=There is not enough change in here. Please insert
                the correct amount of money or contact the hotline.

```

³¹ Jetzt haben wir alle benötigten Komponenten zusammen, um die Konfiguration in der **struts-config.xml** vorzunehmen. Zuerst definieren wir das **PayForm**

```

<form-beans>
.
.

```

³¹Der Text steht komplett in einer Zeile und wurde hier nur aufgrund der Seitenbreite umgebrochen.

```

    <form-bean name="PayForm" type="videoautomat.rent.PayForm"/>
</form-beans>

```

und dann die benötigten Aktionen:

```

<action-mappings>
    .
    .
    <action path="/pay"
        forward="/pages/pay.jsp"
        roles="user"
        parameter="method"/>
    <action path="/PayAction"
        type="videoautomat.rent.PayAction"
        name="PayForm"
        scope="session"
        validate="true"
        input="/pay.do"
        roles="user"
        parameter="method">
        <forward name="cancel" path="/SelectVideos.do"/>
        <forward name="confirm" path="/confirm.do"/>
    </action>
</action-mappings>

```

1.14.3 Confirm

Es fehlt nur noch die Anzeige des Bestätigungszustands. In diesem Zustand wollen wir dem Benutzer die gerade entliehenen Videos und das Wechselgeld anzeigen. Dazu erstellen wir eine Datei **confirm.jsp**.

```

<%@ page import="web.sale.SessionContext"%>
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<%@ taglib uri="/tags/webpoint" prefix="wp" %>

<html:html>
<head>

```

```

        <title><bean:message key="videoautomat.caption"/> -
            <bean:message key="confirm.caption"/></title>
        <html:base/>
    </head>
    <body bgcolor="#FFFFFF">
        <html:errors/>
        <% SessionContext sc = new SessionContext(request); %>
        <bean:message key="confirm.rented.videos"/>
        <wp:DataBasketTable
            db="<%= sc.getBasket() %>"
            dbc="<%= data.DataBasketConditionImpl.allStockItemsWithDest(
                ((videoautomat.AutomatUser) sc.getUser())
                .getVideoStock() ) %>"
            ted="<%= new data.swing
                .DefaultStoringStockDBETableEntryDescriptor() %>"/>
        <br>
    </body>
</html:html>

```

Diese Seite zeigt dem Benutzer alle gerade ausgeliehenen Videos an. Dazu wird der **DataBasketTable**-Tag genutzt. Diesem übergeben wir als Parameter den anzuzeigenden Datenkorb, einen **TableEntryDescriptor** und außerdem eine **DataBasketCondition** (DBC). Die DBC grenzt die Anzeige des Datenkorbes auf bestimmte Bedingungen ein. Für die benötigte **DataBasketCondition** wird auf die Methode **DataBasketConditionImpl.allStockItemsWithDest(Stock s)** zurückgegriffen. Wie der Name schon sagt, filtert das zurückgegebene Objekt alle Einträge heraus, die als Ziel den übergebenen Bestand haben. Bisher noch nicht erwähnt wurde die Anweisung

```
<%@ page import="web.sale.SessionContext" %>
```

Dieses entspricht einem **import** in einer Java Klasse. Wir importieren also hier den **SessionContext** für diese JSP.

Als nächstes zeigen wir das Wechselgeld an und erstellen einen Link, um zurück zum Anmeldeprozess zu kommen. Um das Wechselgeld anzuzeigen, nutzen wir erneut einen Tabellen-Tag. Nachdem wir das Formular angezeigt haben, können wir auf allen benutzten Sub-Datenkörben ein **commit** ausführen und die Änderungen somit dauerhaft machen.

```

.
<br>
<bean:message key="confirm.change"/>
<wp:CountingStockTable
    cs="<%= (PayForm) session.getAttribute("PayForm") ).getMb_temp() %>"
    db="<%=sc.getBasket() %>"/>
<html:link page="/loggedin.do">
    <bean:message key="videoautomat.ok"/>
</html:link>
<%
    sc.getBasket().commitSubBasket(Constants.SUB_SHOP_MONEY);
    sc.getBasket().commitSubBasket(Constants.SUB_SHOP_VIDEO);
    sc.getBasket().commitSubBasket(Constants.SUB_TMP_MONEY);
    sc.getBasket().commitSubBasket(Constants.SUB_USER_VIDEO);
    session.removeAttribute("SelectVideosForm");
    session.removeAttribute("PayForm");
%>
.
.

```

Mit `session.removeAttribute(...)` löschen wir die beiden benutzten Formulare aus der Session, da sie jetzt nicht mehr benötigt werden. Damit wir **PayForm** und **Constants** so benutzen können, müssen wir noch die Import-Anweisung erweitern:

```

<%@ page import="web.sale.SessionContext,
                videoautomat.rent.PayForm,
                videoautomat.rent.Constants"%>

```

In der `MessageResources.properties` definieren wir die verwendeten Meldungen:

```

confirm.caption=Confirm your transaction!
confirm.rented.videos=All your rented videos:
confirm.change=The money you'll get back:

```

In der `struts-config.xml` erstellen wir die **Action** als **forward** auf unsere JSP:

```

<action path="/confirm"
        roles="user"
        forward="/pages/confirm.jsp"/>

```

Der Ausleihprozess ist nun wirklich abgeschlossen. Nach einem erneuten **deploy** können nach Herzenslust Videos ausgeliehen werden.

1.15 Das Protokollieren

Auf den Rückgabeprozess wird in diesem Tutorial nicht weiter eingegangen. Die Implementierung ähnelt stark der des Ausleihprozesses. Der Aufwand ist sogar um einiges geringer, da weniger Interaktion mit dem Kunden stattfindet.

Die in der Aufgabenstellung erwähnte, gesetzliche Forderung nach der Aufzeichnung der Leihvorgänge soll in diesem Kapitel umgesetzt werden. Hilfreich dafür sind die im Paket **Log** zusammengefassten Klassen.

1.15.1 Log-Einträge

Objekte der Klasse **Log** des Frameworks repräsentieren Protokolldateien. Es können verschiedene Logdateien für diverse Zwecke angelegt werden. Zusätzlich lässt sich eine globale Logdatei über entsprechende statische Methoden in der Klasse **Log** setzen und wiedergeben. Im Konstruktor von **VideoShop** wird die globale Protokolldatei der Anwendung definiert.

```
public class VideoShop extends Shop {
    .
    .
    public static final String FILENAME = "automat.log";
    .
    .
    public VideoShop() {
        .
        .
        try {
            Log.setGlobalOutputStream(
                new FileOutputStream(FILENAME, true));
        } catch (IOException ioex) {
            System.err.println("Unable to create log file.");
        }
    }
    .
}
```

```

    }

```

Diese Datei soll für die Aufzeichnung der Verleihvorgänge sowie der Rückgabe von Videos benutzt werden.

Einträge einer Protokolldatei werden über die Klasse **LogEntry** realisiert. Objekte dieser Klasse können über entsprechende Methoden eine Bezeichnung und ein Datum zurückgeben. Wir brauchen Log-Einträge für das Entleihen und Zurückgeben von Videos. Beide haben gewisse Gemeinsamkeiten, so z. B. einen **User**, der die Aktion tätigt, ein Video, das bewegt wurde, und eine Uhrzeit, zu der das Ereignis stattgefunden hat. Was sich unterscheidet, ist der Text, der später als Logeintrag angezeigt werden soll. Aus diesem Grund erstellen wir für die Einträge eine gemeinsame abstrakte Oberklasse **LogEntryVideo**.

```

package videoautomat;

public abstract class LogEntryVideo extends LogEntry {
    private String user_ID;
    private String video_name;
    private Date date;
    public LogEntryVideo(String user_ID, String video) {
        this.user_ID = user_ID;
        this.video_name = video;
        date = (Date) Shop.getTheShop().getTimer().getTime();
    }
    protected abstract String getAction();
    public String toString() {
        return ("The user " + user_ID + getAction() + video_name);
    }
    public Date getLogDate() {
        return date;
    }
}

```

Die Methode **toString** liefert für die Beschreibung des zu protokollierenden Ereignisses **getLogDate()** die Zeit. Für die konkreten Logeinträge erstellen wir dann zwei Unterklassen von **LogEntryVideo**, in denen wir die Methode **getAction** entsprechend implementieren.

```

package videoautomat;

```

```
public class LogEntryGaveBackVideo extends LogEntryVideo {
    public LogEntryGaveBackVideo(String user_ID, String video) {
        super(user_ID, video);
    }
    protected String getAction() {
        return " gave back ";
    }
}

package videoautomat;
public class LogEntryRentedVideo extends LogEntryVideo {
    public LogEntryRentedVideo(String user_ID, String video) {
        super(user_ID, video);
    }
    protected String getAction() {
        return " has rented ";
    }
}
```

Wie bereits angesprochen werden Protokolldateien durch die Klasse `Log` repräsentiert. Die Protokolleinträge können aber nicht direkt an ein solches `Log`-Objekt übergeben werden. Es existiert jedoch eine Methode `log(Loggable l)`, mit der indirekt über ein `Loggable`-Objekt Protokolleinträge erzeugt und dem Protokolldateistrom zugefügt werden können. `Loggable` ist ein Interface, dessen einzige Methode den Protokolleintrag zurückgeben muss, der hinzugefügt werden soll. Entsprechend wird das Interface wie folgt implementiert:

```
package videoautomat;
public class LoggableImpl implements Loggable, Serializable {
    private String user_ID;
    private String video_name;
    private boolean rented;
    public LoggableImpl(String user_ID, String video,
                        boolean rented) {

        this.user_ID = user_ID;
        this.video_name = video;
        this.rented = rented;
    }
}
```

```
public LogEntry getLogData() {
    if (rented)
        return new LogEntryRentedVideo(user_ID, video_name);
    return new LogEntryGaveBackVideo(user_ID, video_name);
}
}
```

Der Parameter **rented** dient hierbei der Erkennung, ob es sich um den Verleih oder die Rückgabe eines Videos handelt.

1.15.2 Ein Zuhörer schreibt mit

Mit Hilfe der neuen Klassen kann die Protokollierung erfolgen. Die Frage ist jedoch, an welcher Stelle des Programms in die Datei geschrieben werden soll. Eine naheliegende Antwort wäre im Ausleih- bzw. Rückgabeprozess. Allerdings dürfte erst dann protokolliert werden, wenn der Vorgang wirklich abgeschlossen ist und keine Möglichkeit eines Abbruchs besteht. Eine andere Realisierung lässt sich mit einem **StockChangeListener** umsetzen. Ein solcher Zuhörer ist Bestandteil eines speziellen Observer-Entwurfsmusters, dem sogenannten Event-Delegations-Muster.

Das Prinzip ist denkbar einfach. Einem Objekt, in diesem Fall der Videobestand des Kunden, können diverse Zuhörer (hier **StockChangeListener**) zugeordnet werden, die sich für Änderungen dieses Objekts interessieren. Die Zuhörer implementieren ein zuvor definiertes Schnittstellenverhalten. Treten Änderungen auf (z. B. wenn Elemente endgültig hinzugefügt werden), informiert das Objekt über die von der Schnittstelle definierten Methoden seine Zuhörerschaft. Das Interface **StockChangeListener** definiert verschiedenste Methoden, bezogen auf die Veränderungen eines Bestandes. Hier sind nur die Methoden **commitAddStockItems (StockChangeEvent e)** und **commitRemoveStockItems (StockChangeEvent e)** von Interesse.

Jetzt wollen wir das Observer-Muster anwenden. Dazu erstellen wir eine neue Klasse, die wir von **StockChangeAdapter** ableiten, wobei die Methoden, die in Folge des endgültigen Hinzufügens bzw. Entfernens von Elementen angesprochen werden, überschrieben werden müssen:

```
package videoautomat;
public class StockChangeLogger extends StockChangeAdapter {
    private String user_ID;
    public StockChangeLogger(String user_ID) {
        this.user_ID = user_ID;
    }
}
```

```
    }
    public void commitAddStockItems (StockChangeEvent event) {
        Iterator it = event.getAffectedItems();
        while (it.hasNext()) {
            try {
                Log.getGlobalLog().log(
                    new LoggableImpl(
                        user_ID,
                        ((StockItem) it.next()).getName(),
                        true));
            } catch (LogNoOutputStreamException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    public void commitRemoveStockItems (StockChangeEvent event) {
        Iterator it = event.getAffectedItems();
        while (it.hasNext()) {
            try {
                Log.getGlobalLog().log(
                    new LoggableImpl(
                        user_ID,
                        ((StockItem) it.next()).getName(),
                        false));
            } catch (LogNoOutputStreamException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Der Zuhörer muss noch dem Bestand des Kunden bekannt gemacht werden. Der Konstruktor der Klasse **AutomatUser** wird dafür um folgende Zeile ergänzt.

```

public AutomatUser(String user_ID, char[] passWd) {
    .
    ss_videos = new StoringStockImpl(user_ID,
                                    VideoShop.getVideoCatalog());
    ss_videos.addStockChangeListener(
                                    new StockChangeLogger(user_ID));
    .
}

```

Die Protokollierung ist damit vollständig implementiert.

Im letzten Kapitel soll gezeigt werden, wie man sich den Inhalt der Protokolldatei anzeigen lassen kann.

1.16 Das Protokoll ansehen

Um das Protokoll anzuzeigen, gibt es in der Tab-Library von WebPoint den Tag **LogTable**. Diesem muss als Parameter mindestens ein **LogInputStream** oder ein **LogFileContent** übergeben werden. Wir wollen an dieser Stelle auf den **LogInputStream** zurückgreifen. Dabei handelt es sich um einen Eingabestrom für Protokolldateien. Außerdem lässt sich für den Eingabestrom ein besonderer Filter definieren, so dass nur bestimmte Protokolleinträge angezeigt werden.

Es soll im Folgenden ein solcher Filter, eine Implementation des Interface **LogEntryFilter** geschaffen werden. Er filtert Instanzen heraus, die nicht vom Typ **LogEntryVideo** sind. Das ist erforderlich, da in der globalen Logdatei auch andere Einträge vorhanden sein können, die nicht angezeigt werden sollen.

Ein **LogEntryFilter** muss eine Methode **accept(LogEntry le)** definieren, die bestimmt, welche Protokolleinträge akzeptiert werden und welche nicht:

```

package videoautomat;
public class LogEntryFilterImpl implements LogEntryFilter {
    public boolean accept(LogEntry le) {
        if (le instanceof LogEntryVideo)
            return true;
        return false;
    }
}

```

Als nächstes können wir uns eine Datei **showlog.jsp** für die Anzeige erstellen. Dabei nutzen wir den **LogTable**-Tag und den gerade erstellten Filter. Weil wir auf eine Datei

zugreifen, muss eine möglicherweise auftretende **IOException** abgefangen werden.

```
<%@ page import="log.LogInputStream,
                videoautomat.LogEntryFilterImpl,
                java.io.FileInputStream"%>
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<%@ taglib uri="/tags/webpoint" prefix="wp" %>

<html:html>
<head>
    <title><bean:message key="videoautomat.caption"/> -
        <bean:message key="admin.showlog.caption"/></title>
    <html:base/>
</head>
<body bgcolor="#FFFFFF">
    <html:errors/>
    <%
        try {
            LogInputStream lis = new LogInputStream(
                new FileInputStream(videoautomat.VideoShop.FILENAME),
                new LogEntryFilterImpl());
        %>
        <wp:LogTable lfc="<%= new log.LogFileContent(lis) %>"/>
    <% } catch (java.io.IOException e) { %>
        <bean:message key="admin.showlog.empty.logfile"/>
    <% } %>
    <br>
    <html:link page="/loggedin.do">
        <bean:message key="videoautomat.ok"/>
    </html:link>
</body>
</html:html>
```

Die verwendeten Schlüssel müssen wir dann noch in den **MessageResources** definieren.

```
admin.showlog.caption=Logged information
```

```
admin.showlog.empty.logfile=The log file was found empty.
videoautomat.admin=Administratrate
```

Die letzte Zeile dient später der Beschriftung des Buttons, um zu der Aktion zu gelangen.

In der **transitions.xml** müssen nur die 2 Übergänge über einen „normalen“ Link eingegeben werden: ³²

```
.
.
<transition>
<page>/loggedin</page>
<page>/showlog</page>
<page>/loggedin</page>
</transition>
```

Da wir wollen, dass nur der Administrator sich die protokollierten Daten anschauen darf, geben wir bei der Konfiguration in der **struts-config.xml** **roles="admin"** an.

```
<action path="/showlog"
        roles="admin"
        forward="/pages/showlog.jsp"/>
```

Damit der Administrator auch wirklich das Recht „admin“ besitzt, müssen wir es ihm im Konstruktor hinzufügen.

```
public AutomatAdmin(String user_ID, char[] passWd) {
    super(user_ID, passWd);
    this.setCapability(new CapabilityImpl("admin"));
}
```

Jetzt müssen wir noch einen Link in unserem Anmeldeprozess oder genauer auf der **loggedin.jsp** hinzufügen, damit der Benutzer unseren neuen „Prozess“ starten kann:

```
<html:link page="/showlog.do">
    <bean:message key="videoautomat.admin"/>
</html:link>
```

Nach einem **deploy** können wir die Anwendung im Browser testen. Dabei stellen wir fest, dass beim Administrator alles funktioniert. Allerdings bekommen alle anderen Nutzer,

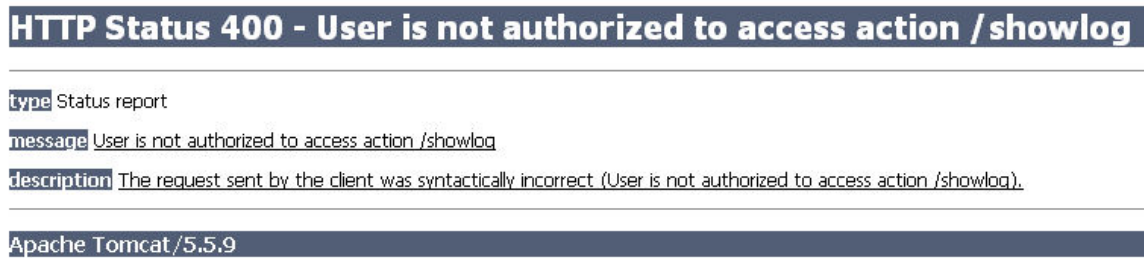


Abbildung 15: Videoautomat - Zugriff nicht erlaubt

wenn Sie auf **Administrate** klicken, eine unschöne Fehlermeldung angezeigt (Abb. 15).

Um das zu vermeiden, schreiben wir uns eine neue **AdministrateAction**, die wir dem **showlog** vorschalten. Diese hat die Aufgabe zu überprüfen, ob der Nutzer überhaupt Administrator ist und ihm ansonsten eine etwas freundlichere Meldung anzuzeigen.

```
package videoautomat.admin;

public class AdministrateAction extends Action {
    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response) {
        SessionContext sc = new SessionContext(request);
        Capability capability = sc.getUser().getCapability("admin");
        if (capability!=null && capability.isGranted()) {
            return mapping.findForward("isAdmin");
        } else {
            ActionMessages messages = new ActionMessages();
            messages.add("access", new ActionMessage("admin.no.access"));
            saveErrors(request, messages);
            return mapping.findForward("noAdmin");
        }
    }
}
```

Für die Fehlermeldung müssen wir noch die Zeile

```
admin.no.access=Access denied!!!
```

³²Da hier keine **Actions** involviert sind, brauchen wir auch kein automatisches Update der **struts-config.xml** durchzuführen.

in der **MessageResources.properties** hinzufügen. Dann passen wir die **transitions.xml** dem geänderten Transaktionsablauf an:

```
.  
.br/><transition>  
<page>/loggedin</page>  
<page action="/AdministrateAction">  
<forward>/loggedin</forward>  
</page>  
<page>/showlog</page>  
<page>/loggedin</page>  
</transition>  
.br/.
```

führen ein **updateStrutsConfig** aus und können nun die Aktion in der **struts-config.xml** wie folgt konfigurieren:

```
<action path="/AdministrateAction"  
        type="videoautomat.admin.AdministrateAction"  
        scope="request"  
        roles="user">  
    <forward name="isAdmin" path="/showlog.do"/>  
    <forward name="noAdmin" path="/loggedin.do"/>  
</action>
```

Hier haben wir nur die Einschränkung **roles="user"** angegeben, damit jeder angemeldete Benutzer auf diese Seite zugreifen kann. Dadurch, dass wir diese Einschränkung vornehmen, brauchen wir auch nicht in der **AdministrateAction** überprüfen, ob der **User** evtl. **null** ist. Zuletzt müssen wir noch unseren Link in der **loggedin.jsp** in

```
<html:link page="/AdministrateAction.do">  
    <bean:message key="videoautomat.admin"/>  
</html:link>
```

ändern.

1.17 Die fertige Anwendung

Den kompletten Videoautomaten findet ihr in der Datei **videoautomat-complete.war**. Das war-Archiv muss genauso wie die **blank.war** in Eclipse eingefügt werden und kann dann mit **deploy** auf den Server übertragen werden.

Anlage B

Quelltext

| | | |
|----------------|------------------------|-----|
| Inhalt: | CustomRequestProcessor | 117 |
| | UpdateStrutsConfig | 122 |


```

package web.sale;

import java.io.File;
import java.io.IOException;
import java.net.MalformedURLException;
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.TreeMap;
import java.util.Map.Entry;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

import users.User;
import web.sale.RequestProcessor;
import web.sale.SessionContext;

/**
 * Extends the <code>RequestProcessor</code> at the predefined hook ({@link
CustomRequestProcessor#processPreprocess(HttpServletRequest, HttpServletResponse)}).
 * Prevents that non-legal transmissions could be done, for example by typing
 * the link in the adress line as an authorized user.
 *
 * @author Daniel Woithe
 *
 */
public class CustomRequestProcessor extends RequestProcessor {

    /**
     * Saves the last page of all users combined with the last access time.
     */
    private TreeMap<String, SessionInfo> map = new TreeMap<String, SessionInfo>();

    /**
     * Counter variable for controlling the garbage collector.
     */
    private int count = 0;

    /**
     * Validates the transitions.
     *
     * Based on the <code>transitions.xml</code> checks if the transition is
     * valid from the previous *.jsp or Action to the requested site.
     *
     * @param request
     * @param response
     * @return <code>true</code> if it's a legal transition.
     *         <code>false</code> otherwise
     * @see
     org.apache.struts.action.RequestProcessor#processPreprocess(javax.servlet.http.HttpServ
letRequest,
     *         javax.servlet.http.HttpServletResponse)
     */
    @SuppressWarnings("unchecked")
    @Override
    protected boolean processPreprocess(HttpServletRequest request,
        HttpServletResponse response) {

```

```

// to remove all invalidates sessions from map
count++;
if (count == 50) {
    garbageCollect(request);
    count = 0;
}

// initialize variables
HttpSession session = request.getSession();
SessionContext sc = new SessionContext(request);
String path=null;
    path = getServletContext().getRealPath("/WEB-INF/transitions.xml");
/*String path = "./webapps" + request.getContextPath()
    + "/WEB-INF/transitions.xml";*/
String id = session.getId();
String next = null;
try {
    next = processPath(request, response);
} catch (IOException e) {
    e.printStackTrace();
}

if (sc.getRequestId() != null
    && !id.equals(sc.getRequestId())
    && map.get(sc.getRequestId()) != null) {
    // session was invalidated
    String tmp = map.get(sc.getRequestId()).getForward();
    map.remove(sc.getRequestId());
    map.put(id, new SessionInfo(tmp, session.getLastAccessedTime()));
}
String prev = null;
if (map.get(id) != null) {
    prev = map.get(id).getForward();
}
if (prev == null && next.equals(startState(path))) {
    // first visit on start site
    map.put(id, new SessionInfo(next, session.getLastAccessedTime()));
    sc.setAttribute("test", map.toString());
    return true;
}

ArrayList<String> poss = nextTransition(prev, path);
if (poss.contains(next)) {
    // allowed transition
    map.remove(id);
    map.put(id, new SessionInfo(next, session.getLastAccessedTime()));
    sc.setAttribute("test", map.toString());
    return true;
}
if (next.equals(startState(path))) {
    // exiting the application by calling the start site
    map.remove(id);
    session.invalidate();
    map.put(request.getSession().getId(), new SessionInfo(next, request
        .getSession().getLastAccessedTime()));
    sc.setAttribute("test", map.toString());
    return true;
}
// invalid transition
try {
    map.remove(id);
    User user = sc.getUser();
    session.invalidate();
    new SessionContext(request).attach(user);
    map.put(request.getSession().getId(), new SessionInfo(prev, request

```

```

        .getSession().getLastAccessedTime()));
        // response.sendRedirect(sc.getContextPath());
        response.sendError(HttpServletResponse.SC_FORBIDDEN,
            "invalid transition from" + " " + prev + " " + "to" + " "
            + next);
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
sc.setAttribute("test", map.toString());
return false;
}

/**
 * Searches for all in <code>path</code> defined followup states for the
 * <code>prev</code> state.
 *
 * @param prev
 *         State which followup states are searched for.
 * @param path
 *         Path of the XML-file where all transitions are stored
 * @return A list of all possible followup states.
 */
@SuppressWarnings("unchecked")
private ArrayList<String> nextTransition(String prev, String path) {
    ArrayList<String> erg = new ArrayList<String>();
    if (prev == null)
        return erg;
    SAXBuilder sax = new SAXBuilder();
    File file = new File(path);
    try {
        Document doc = sax.build(file);
        Element root = doc.getRootElement();
        Iterator<Element> it = root.getChildren("transition").iterator();
        while (it.hasNext()) {
            Element trans = it.next();
            ListIterator<Element> elts = trans.getChildren("page")
                .listIterator();
            // search all possible following states
            while (elts.hasNext()) {
                Element akt = elts.next();
                // current state has only one successor
                if (akt.getTextTrim().equals(prev) && elts.hasNext()) {
                    // add the next element to result list
                    Element succ = elts.next();
                    if (succ.getAttributeValue("path") != null) {
                        erg.add(succ.getAttributeValue("path"));
                    } else if (succ.getAttributeValue("action") != null) {
                        erg.add(succ.getAttributeValue("action"));
                    } else {
                        erg.add(succ.getTextTrim());
                    }
                }
                elts.previous();
            }
            // current state has several successors
            if (prev.equals(akt.getAttributeValue("path"))
                || prev.equals(akt.getAttributeValue("action"))) {
                // add children (forwarding) states to result list
                Iterator<Element> children = akt.getChildren("forward")
                    .iterator();
                while (children.hasNext())
                    erg.add(children.next().getTextTrim());
                // add the next element to result list
                if (elts.hasNext()) {
                    Element succ = elts.next();

```

```

        if (succ.getAttributeValue("path") != null) {
            erg.add(succ.getAttributeValue("path"));
        } else if (succ.getAttributeValue("action") != null) {
            erg.add(succ.getAttributeValue("action"));
        } else {
            erg.add(succ.getTextTrim());
        }
        elts.previous();
    }
    // add input element to result list
    if (akt.getAttributeValue("input") != null) {
        erg.add(akt.getAttributeValue("input"));
    }
    }
}
} catch (JDOMException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}
return erg;
}
}

/**
 * Searches for the start state in the <code>path</code> file.
 *
 * @param path
 *         Path of the XML-file where all transitions are stored
 * @return The start state of the application.
 */
private String startState(String path) {
    SAXBuilder sax = new SAXBuilder();
    File file = new File(path);
    try {
        Document doc = sax.build(file);
        Element root = doc.getRootElement();
        return (root.getAttributeValue("start"));
    } catch (JDOMException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}

/**
 * Filters all entries which <code>Session</code>s are timed out and
 * removes them from the {@link CustomRequestProcessor#map}.
 *
 * @param request
 *         It's important to access the <code>Session</code>.
 */
private void garbageCollect(HttpServletRequest request) {
    Iterator<Entry<String, SessionInfo>> it = map.entrySet().iterator();
    ArrayList<String> list = new ArrayList<String>();
    while (it.hasNext()) {
        Entry<String, SessionInfo> entry = it.next();
        if (entry.getValue().getTime()
            + request.getSession().getMaxInactiveInterval() * 1000 < (new Date())
                .getTime()) {
            list.add(entry.getKey());
        }
    }
    Iterator<String> listIt = list.iterator();

```

```
        while (listIt.hasNext()) {
            map.remove(listIt.next());
        }
    }

/**
 * Wrapper class only for storing two information (the last requested page
 * and the time the session was last accessed.
 *
 * @author Daniel Woithe
 *
 */
class SessionInfo {
    private String forward;

    private long time;

    SessionInfo(String s, long t) {
        forward = s;
        time = t;
    }

    public String getForward() {
        return forward;
    }

    public long getTime() {
        return time;
    }

    public String toString() {
        return forward + " " + Long.toString(time);
    }
}
```

```

package web.util.ant.tasks;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.ListIterator;

import org.jdom.Attribute;
import org.jdom.Content;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;

public class UpdateStrutsConfig {

    public static void main(String[] args) throws IOException {
        updateStrutsConfig(args[0],args[1]);
    }

    @SuppressWarnings("unchecked")
    private static void updateStrutsConfig(String source, String dest) {
        SAXBuilder sax = new SAXBuilder();
        try {
            Document trans = sax.build(new File(source));
            Document struts = sax.build(new File(dest));
            Iterator<Element> transitions = trans.getRootElement().getChildren(
                "transition").iterator();

            while (transitions.hasNext()) {
                ListIterator<Element> pages = transitions.next().getChildren(
                    "page").listIterator();
                while (pages.hasNext()) {
                    Element curPage = pages.next();
                    if (curPage.getAttribute("action") != null) {
                        String curAction = curPage.getAttributeValue("action");
                        Iterator<Element> actions = struts.getRootElement()
                            .getChild("action-mappings").getChildren(
                                "action").iterator();
                        boolean isExistent = false;
                        Element newAction = new Element("action");
                        while (!isExistent && actions.hasNext()) {
                            Element strutsAction = actions.next();
                            if (curAction.equals(strutsAction
                                .getAttributeValue("path"))) {
                                // defined Action pre-existent in struts-config

                                isExistent = true;

                                // copy all attributes to the new Element
                                // "action"
                                Iterator<Attribute> attr = strutsAction
                                    .getAttributes().iterator();
                                while (attr.hasNext()) {
                                    Attribute curAttr = attr.next();
                                    newAction.setAttribute(curAttr.getName(),
                                        curAttr.getValue());
                                }

                                // update the "input" attribute
                                if (curPage.getAttribute("input") == null)

```

```

        newAction.removeAttribute("input");
    else
        newAction.setAttribute("input", curPage
            .getAttributeValue("input")
            + ".do");

    // update all "forward" elements
    Iterator<Element> pageForwards = curPage
        .getChildren("forward").iterator();
    while (pageForwards.hasNext()) {
        Element curForward = pageForwards.next();
        newAction = updateAction(strutsAction,
            newAction, curForward.getTextTrim()
                + ".do");
    }
    // checks if a following page element exists in
    // the source that must be a forward element in
    // the destination
    String nextPage = nextPage(pages);
    if (nextPage != null) {
        newAction = updateAction(strutsAction,
            newAction, nextPage + ".do");
    }
    struts.getRootElement().getChild(
        "action-mappings").removeContent(
        strutsAction);
    }
}
if (!isExistent) {
    // defined Action has to be created in struts-config
    newAction = newAction(pages, curAction, curPage);
}

struts.getRootElement().getChild("action-mappings")
    .addContent(newAction);
XMLOutputter out = new XMLOutputter(Format
    .getPrettyFormat());
File file = new File(dest);
file.delete();
file.createNewFile();
out.output(struts, new FileOutputStream(file));
System.out.println(curAction + " " + isExistent);
}
}
}
} catch (JDOMException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}

@SuppressWarnings("unchecked")
private static Element newAction(ListIterator<Element> pages,
    String curAction, Element curPage) {
    Element newAction = new Element("action");
    // add all possible attributes
    newAction.setAttribute("path", curAction);
    newAction.setAttribute("type",
        "*** fully qualified Java class name of the Action subclass ***");
    newAction.setAttribute("name", "*** name of the form bean ***");
    newAction.setAttribute("scope", "*** request or session ***");
}

```

```

newAction
    .setAttribute("validate",
        /** true if the validate method of the ActionForm bean should be called
**");
newAction.setAttribute("input", curPage.getAttributeValue("input")
    + ".do");
newAction.setAttribute("roles", /** list of security role names **");
newAction.setAttribute("parameter",
    "method **remove if no form-bean is used**");
Iterator<Element> forwards = curPage.getChildren("forward").iterator();
// add all defined forwards
ArrayList<Content> forwardList = new ArrayList<Content>();
while (forwards.hasNext()) {
    Element forward = new Element("forward");
    forward.setAttribute("name", "****");
    forward.setAttribute("path", forwards.next().getTextTrim() + ".do");
    forwardList.add(forward);
}
String nextPage = nextPage(pages);
if (nextPage != null) {
    // add the next element as forward element
    Element forward = new Element("forward");
    forward.setAttribute("name", "****");
    forward.setAttribute("path", nextPage + ".do");
    forwardList.add(forward);
}
newAction.setContent(forwardList);
return newAction;
}

private static String nextPage(ListIterator<Element> pages) {
    if (pages.hasNext()) {
        Element succ = pages.next();
        if (succ.getAttributeValue("path") != null)
            return succ.getAttributeValue("path");
        else if (succ.getAttributeValue("action") != null)
            return succ.getAttributeValue("action");
        pages.previous();
        return succ.getTextTrim();
    }
    return null;
}

@SuppressWarnings("unchecked")
private static Element updateAction(Element strutsAction,
    Element newAction, String forward) {
    Iterator<Element> actionForwards = strutsAction.getChildren("forward")
        .iterator();
    boolean added = false;
    while (!added && actionForwards.hasNext()) {
        Element actionForward = actionForwards.next();
        if (forward.equals(actionForward.getAttributeValue("path"))) {
            newAction.addContent((Element) actionForward.clone());
            added = true;
        }
    }
    if (!added) {
        Element newForward = new Element("forward");
        newForward.setAttribute("name", "****");
        newForward.setAttribute("path", forward);
        newAction.addContent(newForward);
    }
    return newAction;
}
}
}

```

Literaturverzeichnis

- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
- [RW05] Conrad Reisch und Torsten Walter. WebPoint: ein Framework für webbasierte Verkaufsanwendungen, 2005.
- [SC] Struts Console. <http://www.jamesholmes.com/struts/console/index.html>
- [Sch] Dimitri Schischkin. Struts. Eine Einführung in das Java-Framework für Webanwendungen.
- [Tut] Videoautomat Tutorial.
<http://www-st.inf.tu-dresden.de/SalesPoint/v3.1/tutorial/videoautomat/>
- [Zsc00] Steffen Zschaler. Das Framework „SalesPoint“: Technische Beschreibung der Version 2.0 und weiterer Ausbaumöglichkeiten, 2000.