ISBN 3-8266-1431-3

3 Der Controller

Der Controller ist das Herzstück des Struts-Frameworks. In der klassischen Datenverarbeitung (Eingabe --> Verarbeitung --> Ausgabe) übernimmt der Controller die Eingabe, das Model die Verarbeitung und die View die Ausgabe der Daten. Wie bereits in Kapitel 1 beschrieben, übernimmt Struts im Wesentlichen die Aufgabe des Controllers und der View. Die Tendenz ist, dass in weiteren Struts-Versionen sehr viel mehr Augenmerk auf den Controller gelegt wird. Dies hat vielfältige Gründe:

- Im Bereich der Webentwicklung gibt es inzwischen sehr viele verschiedene Möglichkeiten, die Daten zu präsentieren, wie z.B. JSP-, XML-Seiten oder das Velocity-Framework (siehe Kapitel 4, Abschnitt 5.6). In der 1.x-Serie versuchten die Entwickler des Struts-Frameworks, die verschiedenen Ausgabemöglichkeiten an Struts anzupassen. Besser wäre es aber, auf Standards zu setzen, wie z.B. die JSTL (Java Standard Tag Library). Genau dieser Trend wird mit Struts 2.0 verfolgt.
- Im Bereich des Models, der Geschäftslogik, hat Struts zurzeit nicht viel zu bieten. Die Einstiegspunkte, die Actions, sind vorhanden. Es existiert aber kein schlüssiges Konzept, wie Businesslogik sinnvoll an Struts angebunden werden kann. In Zukunft soll ein neues Jakarta-Framework (agility) die Brücke zwischen Struts und der Geschäftslogik schließen.

Der Struts-Controller hat schematisch gesehen folgende Funktionalität:

- Anfragen, die von der Weboberfläche in Form von HTTP-Requests kommen, werden angenommen und bearbeitet. Dabei bestimmt das Servlet-Mapping, welche Requests von dem Struts-Controller bearbeitet werden sollen.
- Der zweite Schritt ist, dass der Controller ermitteln muss, was der User mit diesem Request erreichen will (welche Action soll aufgerufen werden etc.).

- Daten werden aus der Geschäftslogik geholt, die für eine bestimmte Darstellungsart benötigt werden.
- Die nächste View finden, die diese Daten zur Anzeige bringt.

3.1 Eine kurze Geschichte des Servlets

Seit es Java gibt (Einführung ca. 1997) gibt es auch die Java-Servlet-API, die es ermöglicht, serverseitiges HTML zu erzeugen. Zu Beginn dieser Entwicklung haben Servlets Anfragen (Requests) entgegengenommen und den HTML-Code generiert und in Form einer Antwort (Response) zurückgegeben. Relativ schnell wurde klar, dass eine solche Herangehensweise für Webdesigner nicht tragbar ist, weil Servlets aus reinem Java-Code bestehen. Aus diesem Grund wurden die Java Server Pages eingeführt, die vom Aufbau eher dem HTML-Design entsprechen. Java Server Pages werden durch den Java-Compiler in Servlets umgewandelt, sodass im Grunde genommen das Servlet weiterhin den zentralen Punkt im serverseitigen Java im Webumfeld einnimmt.

Ohne die Hinzuziehung eines Frameworks gibt es mehrere Möglichkeiten, Anwendungen mit Servlets zu implementieren. Es ist möglich, für jeden Request ein Servlet zu bemühen. Es ist außerdem möglich, Aufrufe direkt von einer JSP-Seite zu einer anderen zu leiten. Außerdem sind alle Zwischenschritte denkbar.

Grundsätzlich wird sich in einem größeren Projekt immer Folgendes durchsetzen:

- Die Aufrufe werden immer an ein Servlet geleitet, das als Dispatcher fungiert und die Aufrufe an andere Servlets oder an eine JSP-Seite weiterleitet.
- Weil durch diese Herangehensweise sehr große Case-Statements entstehen, wird man sich eine generische Lösung einfallen lassen, z.B. Konfiguration von bestimmten Servlets oder Klassen in einer XML-Datei. Die Aufrufe werden dann an andere Java-Klassen delegiert.

Alle, die bereits seit längerem mit Servlets arbeiten, werden bereits ein solches »kleines« Framework entwickelt haben, so auch wir, die Autoren

dieses Buches. Genau dieses »Problem« ist Ausgangspunkt von Struts. Struts implementiert das Front-Controller-Pattern, das kurz skizziert folgende Eigenschaften hat:

- Alle Aufrufe werden grundsätzlich durch den Front-Controller bearbeitet. Es gibt keinen Aufruf, der an dem Controller vorbeigeht. Damit verbieten sich `<jsp:forward>`-Aufrufe in einer JSP-Seite.
- Der Front-Controller allein entscheidet mithilfe von Konfigurationswissen, welche Komponente für die Abarbeitung eines Aufrufs verantwortlich ist. Im Struts-Umfeld befinden sich diese Informationen in der `struts-config.xml`.
- Nachdem ein Request abgearbeitet wurde, entscheidet der Front-Controller, was der nächste Schritt ist bzw. welche Komponente als Nächstes aufgerufen wird, damit die Antwort ausgeliefert werden kann.
- Die Aufgabenverteilung des Controllers ist Sache der Anwendungsarchitektur. Bei kleinen Anwendungen reicht es aus, mithilfe von Case-Statements andere JavaBeans oder JSP-Seiten durch den Controller zu kontaktieren. Bei großen Anwendungen macht es Sinn, neben dem Front-Controller weitere Instanzen hinzuzufügen, die Teilaufgaben des Controllers übernehmen.
- Im Struts-Framework ist diese Aufgabenteilung wie folgt gelöst:

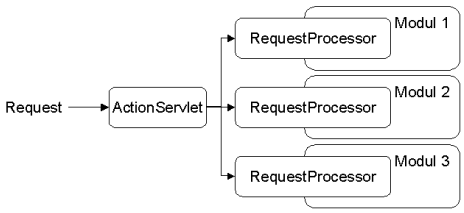

Abbildung 3.1: `ActionServlet` *als Front-Controller*

- Der Front-Controller wird durch eine Instanz der `ActionServlet`-Klasse repräsentiert. Dieses Action-Servlet initialisiert das Framework und leitet jeden Request an einen Request-Prozessor weiter.

- Für jedes Struts-Modul existiert eine `RequestProcessor`-Instanz. Der Request-Prozessor führt eine Reihe von Aktivitäten durch (siehe Abschnitt 3.3 »Der Request-Prozessor«) und ermittelt vor allem eine `Action`-Instanz, die letztendlich für die Abarbeitung des Requests zuständig ist.
- Eine `Action` ist im Kontext von Struts die kleinste Einheit, die Anfragen entgegennehmen und bearbeiten kann. Die Action-Klasse kann wiederum JavaBeans oder andere Komponenten in Anspruch nehmen, um die angeforderte Aufgabe zu bewältigen.

Im Folgenden werden wir die drei Hauptkomponenten des Controllers vorstellen. Außerdem wollen wir einen Blick in die Zukunft werfen und das Framework Struts-Chain vorstellen.

3.2 Das Action-Servlet

Das Action-Servlet ist der Controller im Model-View-Controller-Design-Pattern für Webanwendungen. Struts implementiert die Model-2-Architektur, die im Folgenden kurz skizziert wird:

- Die Benutzeroberfläche wird grundsätzlich mit Java Server Pages erstellt. In diesen Seiten ist keine Geschäftslogik enthalten. Die JSP repräsentiert die View der MVC-Architektur.

> Es gibt im Struts-Umfeld mehrere Extensions, die auch andere Präsentationsarten wie XSLT oder Velocity mit Struts verbinden. Wir werden in diesem Buch im Wesentlichen auf die JSP-Variante zurückgreifen, die Struts auch direkt mit seinen Tag-Bibliotheken unterstützt, und außerdem einen Blick auf Velocity werfen.

- Alle Requests, die Geschäftslogik ansprechen wollen (`form`-Tags sowie Hyperlinks), adressieren immer das Action-Servlet.
- Es existiert nur ein Action-Servlet in der Anwendung. Das Action-Servlet delegiert Aufrufe an einen Request-Prozessor, der den »Controller-Part« in einem bestimmten Modul übernimmt.

- Statt eine Ressource direkt zu adressieren, wird die aufgerufene Action ein Forward zurückgeben, das vom Request-Prozessor ausgewertet wird. Der Request-Prozessor hat die Verantwortlichkeit, eine andere Ressource mittels Forward oder Redirect anzusprechen.

3.2.1 Forward vs. Redirect

An dieser Stelle wollen wir kurz den Unterschied zwischen einem Forward und einem Redirect erläutern, weil in der `struts-config.xml` bei jedem Mapping entschieden werden muss, ob ein Forward oder ein Redirect durchgeführt werden soll. Ein Redirect sollte normalerweise durchgeführt werden, wenn eine Ressource eine Anfrage nicht beantworten kann. Das Redirect ist also eine erneute Anfrage.

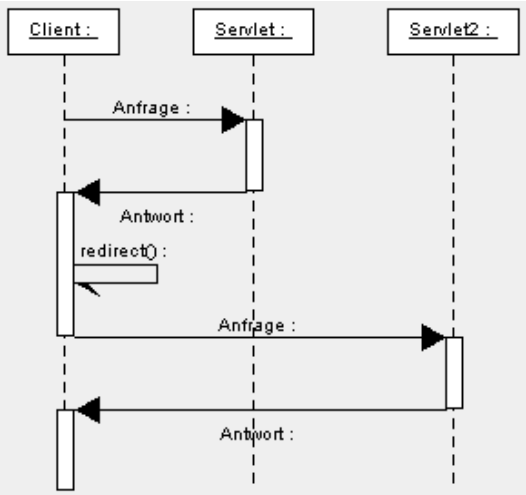

Abbildung 3.2: Request `redirect()`

Ein Forward ist eine Weiterleitung einer Anfrage. Beim Forward wird der Request, mit allen enthaltenen Parametern an eine andere Ressource weitergeleitet. Bei einem Redirect wird ein neuer Request aufgebaut. Die gespeicherten Request-Attribute gehen verloren.

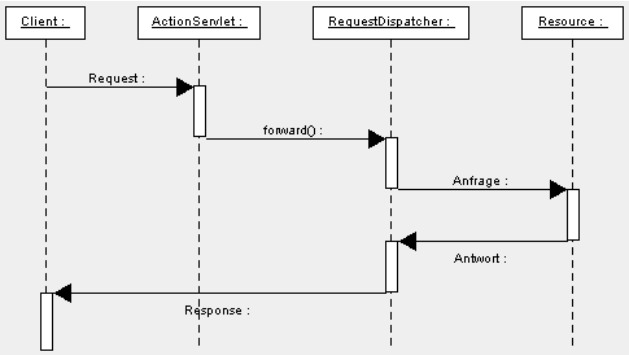

Abbildung 3.3: Request `forward()`

In den beiden Diagrammen wird der Unterschied in vereinfachter Darstellung deutlich gemacht. Kurz zusammengefasst: Das Redirect ist ein erneuter Aufruf, während das Forward eine Weiterleitung ist.

3.2.2 Die Initialisierungsphase

Das Action-Servlet führt folgende Aktivitäten in der Initialisierungsphase durch (dargestellt an Funktionsaufrufen innerhalb des Servlets):

- **initInternal:** Initialisiert das eigene Resource-Bundle.
- **initOther:** Initialisiert das zentrale Verhalten des Servlets, wenn eine Ressource nicht gefunden wurde. Normalerweise gibt das Servlet dann den Wert `null` zurück. Es kann aber auch so parametrisiert werden, dass ein Objekt mit dem initialisierten 0-Wert zurückgegeben wird (String = "", Integer = `0`).

- **initServlet:** Initialisiert das Servlet-Mapping. Dabei kann wie in Kapitel 2 beschrieben ein Pfad-Mapping oder ein Extension-Mapping konfiguriert sein.

> Bedenken Sie bitte, dass bei einer Konfiguration, die mehrere Module umfasst, immer nur das Extension-Mapping benutzt werden darf. Am besten benutzen Sie das Pfad-Mapping gar nicht.

- **initModuleConfigFactory:** Überprüft, ob eine andere Modul-Factory für das Erzeugen der Modul-Factory benutzt werden soll als die »normale« Modulkonfiguration.

> Mit diesem Hebel haben Sie seit Struts-Version 1.1 die Möglichkeit, Ihre Struts-Module anders zu definieren als mithilfe der `struts-config.xml`. Insbesondere wenn Sie die Modulkonfiguration zur Laufzeit ändern wollen, kann dies mit der jetzigen XML-Beschreibung nicht durchgeführt werden, weil es keinen Transaktionskontext für die Änderungen gibt. Hier könnten Sie jetzt eingreifen, um z.B. die Module datenbankgestützt zu konfigurieren.

- Für jedes Modul wird jetzt die Konfiguration durch folgende Aufrufe vollzogen:

Aufruf	Bedeutung
`initModuleConfig`	Initialisiert die Workflow-Informationen des Struts-Moduls
`initModuleMessageResources`	Initialisiert die Message-Ressourcen für das Modul
`initModuleDataSources`	Initialisiert die Datenbankverbindungen für das Modul

Aufruf	Bedeutung
`initModulePlugIns`	Initialisiert alle konfigurierten Plug-Ins
`freeze`	Friert die Modulkonfiguration ein

Die Modulkonfiguration muss eingefroren werden, weil alle Informationen in nicht synchronisierten `Map`-Objekten gespeichert werden. Dieses Einfrieren führt zu regelmäßigen Diskussionen in der Struts-Mailingliste. Die Autoren haben einen Vorschlag erarbeitet, wie das Einfrieren verhindert werden kann. Allerdings müssen dann auch »Nebenwirkungen« wie fehlende Transaktionalität in Kauf genommen werden. Auf der Webseite zu diesem Buch, unter der Rubrik »Artikel«, finden Sie einen Artikel zu diesem Thema.

3.2.3 Verantwortlichkeiten

In früheren Versionen des Struts-Frameworks hatte das Servlet noch sehr viel Verantwortung. Seit der Version 1.1, der Einführung des Request-Prozessors, hat das Servlet kaum noch etwas zur Laufzeit zu tun. Dies soll kurz durch Dokumentation der `process()`-Methode verdeutlicht werden:

```
protected void process(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException, ServletException {

    ModuleUtils.getInstance().selectModule(
            request, getServletContext());
    getRequestProcessor(getModuleConfig(request)).
            process(request, response);
}
```

Listing 3.1: Die `process()`*-Methode des Action-Servlets*

Das Servlet muss zunächst ermitteln, welches Modul für den hereinkommenden Request zuständig ist. Als Nächstes wird der Request-Prozessor für das entsprechende Modul ermittelt und der Request an den Request-Prozessor zur Bearbeitung übergeben.

3.2.4 Konfiguration

Grundsätzlich wird es meistens nicht nötig sein, das Action-Servlet zu überladen. Soll zusätzliches Wissen über die Konfiguration der Anwendung hinzugefügt werden, sollten andere Mechanismen, wie z.B. Plug-Ins, benutzt werden, um XML-Dateien zu lesen oder bestimmte Initialisierungen aus Datenbanken durchzuführen. Deshalb kann das Servlet immer auf die gleiche Weise in der `web.xml` konfiguriert werden:

```
<!-- Standard Action Servlet Configuration-->
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>
        org.apache.struts.action.ActionServlet
    </servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>
           /WEB-INF/struts-config.xml
       </param-value>
    </init-param>
    <init-param>
      <param-name>debug</param-name>
      <param-value>2</param-value>
    </init-param>
    <init-param>
      <param-name>detail</param-name>
      <param-value>2</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

```
<!-- Standard Action Servlet Mapping -->
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Listing 3.2: Standardkonfiguration des `ActionServlet`

Falls Sie dennoch Initialisierungen direkt im Servlet vornehmen wollen, sollten Sie nur die `init()`-Methode überschreiben und nach Aufruf von `super.init()` Ihre eigenen Methoden aufrufen. In diesem Fall müssen Sie in der `<servlet-class>`-Konfiguration Ihr eigenes ActionServlet eintragen.

3.3 Der Request-Prozessor

Wie bereits erwähnt übernimmt der Request-Prozessor die meisten Aufgaben des Servlets. Für jedes Modul kann ein eigener Request-Prozessor vereinbart werden.

Wenn Sie mit Tiles arbeiten wollen, müssen Sie den `TilesRequestProcessor` benutzen. Standardmäßig wurde die `struts-blank` in einigen Nightly Builds mit diesem Request-Prozessor ausgeliefert. Dies ist häufig eine Ursache für Probleme beim Ausprobieren. Wenn Sie Tiles nicht benutzen wollen, brauchen Sie keinen Prozessor anzugeben, weil der Standardprozessor voreingestellt ist.

3.3.1 Prozesskette des Prozessors

Der Request-Prozessor durchläuft eine gesamte Prozesskette, um einen Request abzuarbeiten. Wir dokumentieren diese Kette wieder durch die Beschreibung der Methoden, die durchlaufen werden:

- **processMultipart:** Wenn ein Multipart-Request vorliegt, z.B. bei einem Datei-Upload wird dieser in eine Klasse verpackt (in einen Wrapper).
- **processPath:** Der Pfad wird extrahiert, der für das Action-Mapping benötigt wird. Hier wird auch die Erweiterung beim Extension-Mapping herausgefiltert. Innerhalb des Mappings wird dann der Pfad ohne Erweiterung benutzt.
- **processLocale:** Das momentan eingestellte Locale für den User wird hier ermittelt und in den Session-Schlüssel `Globals.LOCALE_KEY` geschrieben. Grundsätzlich wird der Schlüssel nur dann gesetzt, wenn er nicht schon besteht (also praktisch nur im ersten Request).

Wenn Sie den Session-Schlüssel selber setzen, können Sie sicher sein, dass diese Einstellung nicht überschrieben wird. So haben Sie die Möglichkeit, die Sprache selbst zu steuern.

- **processContent:** Wenn im Controller ein »Content-Typ« definiert wurde, wird dieser hier gesetzt.
- **processNoCache:** Wenn im Controller das `nocache`-Attribut auf `true` gesetzt wurde, wird in dieser Methode der Header in den Metainformationen auf `no-cache` gesetzt. Diese Methode weist den Browser an, die Seite nicht zwischenzuspeichern.
- **processPreprocess:** Diese Methode ist im Request-Prozessor nicht implementiert. Sie gibt abgeleiteten Klassen die Möglichkeit, einen Hook zu installieren. Grundsätzlich kann diese Methode `false` zurückliefern, um die Prozesskette hier zu beenden.

Den Hook an dieser Stelle zu platzieren ist nicht ganz glücklich. Wird an dieser Stelle eine Exception »geworfen«, kann sie nicht über die konfigurierten Struts-Exception-Handler behandelt werden. Besser ist es, den Hook in der Methode `processActionPerform` zu platzieren, wie das nächste Listing zeigt.

```
protected ActionForward processActionPerform(
    HttpServletRequest request,
    HttpServletResponse response,
    Action action,
    ActionForm form,
    ActionMapping mapping)
    throws IOException, ServletException {
    try {
       processPreprocess(request, response);
       return (action.execute(mapping, form,
                              request, response));
    } catch (Exception e) {
        return (processException(request, response,
                e, form, mapping));
    }
}
```

Listing 3.3: Ein Hook mit Exception-Handling

In diesem Beispiel kann die Hook-Methode eine Exception werfen, die über einen in der `struts-config.xml` konfigurierten Exception-Handler behandelt werden kann.

- **processCachedMessages:** Diese Methode erlaubt es ab Struts 1.2, dass Meldungen in der Session abgespeichert werden können. Wurden die Meldungen abgerufen, löscht diese Methode alle Messages. Wenn nicht, verbleiben die Meldungen in der Session.

Mit dieser neuen Funktion ist es möglich, Meldungen über mehrere Actions hinweg aufzusammeln. In der Struts-Mailing-Liste wurde diese Funktion sehr begrüßt. Dieser Mechanismus sollte nicht dazu benutzt werden, Fehlermeldungen, die an die Oberfläche gemeldet werden sollen, aufzusammeln. Dazu sollte der normale Mechanismus der Action genutzt werden, der die Meldungen in den Request stellt.

- **processMapping:** An dieser Stelle wird aus dem extrahierten Pfad ein Mapping aus der Modulkonfiguration identifiziert. Falls kein Mapping gefunden wird, wird ein Fehler-Response erzeugt und die Methode gibt `null` zurück. Falls das Mapping zur Laufzeit manipuliert werden soll, ist hier der richtige Ansatzpunkt.
- **processRoles:** Für jede Action kann in der `struts-config.xml` festgelegt werden, welche vordefinierten Rollen diese `Action` betreten dürfen. Beim Tomcat 5.x beispielsweise können diese Rollen in der `tomcat-users.xml` angelegt werden. Möglich sind auch so genannte Security Realms, bei denen die Rollen z.B. in der Datenbank gespeichert werden.
- **processActionForm:** An dieser Stelle wird die Action-Form für dieses Mapping erzeugt oder eine vorhandene Form recycelt. Die Form wird dann je nach konfiguriertem Gültigkeitsbereich in den Request- oder den Sessionkontext gespeichert.
- **processPopulate:** Populate ist die Übertragung von Werten im Request in die mit dem aktuellen Mapping verbundene Action-Form. Dabei werden alle Werte, die Namensgleichheit sowohl im Request als auch in der Action-Form haben, übertragen.

Die Menge der Attribute, die mithilfe des Populates übertragen werden können über Prä- und Suffixe eingeschränkt werden. Wird ein Prä- oder ein Suffix vereinbart, so werden nur die Werte übertragen, die ebenfalls dieses Prä- bzw. Suffix besitzen.

- **processForward:** Falls bei diesem Mapping ein Forward konfiguriert wurde, wird dieses Forward ausgeführt. Wenn nicht, gibt diese Methode `true` zurück und zeigt damit an, dass in dem Request kein Forward gefunden wurde, das ausgeführt werden soll.
- **processInclude:** Hier wird die gleiche Logik ausgeführt wie bei `processForward`, nur dass hier Includes abgearbeitet werden.
- **processActionCreate:** An dieser Stelle wird die Action-Klasse, die zu dem Mapping gehört, erzeugt oder eine vorhandene Instanz recycelt.

- **processActionPerform:** Die letzte Aktion des Request-Prozessors ist, die gefundene Action auszuführen und den Rückgabewert der Action auszuwerten. Der Prozessor führt die `execute()`-Methode aus und wertet in der nächsten Methode den Rückgabewert der Action, das Action-Forward, aus.
- **processForwardConfig:** Diese Methode ermittelt den modulrelativen Pfad für das Forward oder das Redirect. Wenn der Pfad nicht mit einem Slash (/) beginnt, wird das Forward nicht ausgeführt.

3.3.2 Überladen des Prozessors

Der Request-Prozessor muss immer dann überladen werden, wenn eine der folgenden Voraussetzungen vorliegt:

- Änderung der Prozesskette: Wenn die oben vorgestellte Prozesskette geändert werden soll, muss die `process()`-Methode überschrieben werden.

> Der nächste konsequente Schritt dies zu tun ist die Möglichkeit, die Prozesskette über eine Konfigurationsdatei variabel zu machen. Dies wird in Version 1.3 des Struts-Frameworks über die Einführung von Chains geschehen.

- Änderung der Implementierung einzelner Prozessschritte: Soll die Implementierung eines oder mehrerer Schritte geändert werden, muss die jeweilige `processXX()`-Methode überladen werden, um diesem Prozessschritt einen neue Funktionalität zu geben.

Alle anderen Änderungen sollten außerhalb des Request-Prozessors durchgeführt werden, nämlich im Action-Servlet (Konfiguration, Initialisierung) oder in der `Action`-Klasse.

3.4 Die Action-Klasse

Die dritte Instanz in der Dreierkette ist die Action-Klasse. Während das Action-Servlet und der Request-Prozessor mehr zum technischen

Struts-Framework gehören und normalerweise kaum angepasst bzw. Subklassen von ihnen gebildet werden müssen, gehört die Action-Klasse mehr zur fachlichen Umsetzung der Geschäftsvorfälle. Deshalb hier noch einmal kurz eine Beschreibung der Verantwortlichkeiten der verschiedenen Controller-Komponenten:

- **Action-Servlet:** Konfiguriert die gesamte Anwendung. Agiert auf Ebene der gesamten Anwendung.
- **Request-Prozessor:** Stellt die Prozesskette für die Abarbeitung eines Requests zur Verfügung. Agiert auf Ebene eines einzelnen Moduls.
- **Action:** Bearbeitet ein Event (einen Request). Agiert auf Ebene von atomaren Aktivitäten.

3.4.1 Aufgabe der Action-Klasse

Die Action ist kurz gesagt die Schnittstelle zwischen der Präsentationsschicht und der Geschäftslogik. Die Action hat eine sehr kleine Schnittstelle auf der Controllerseite: die `execute()`-Methode. Dabei sind Action-Klassen vom Umfang her je nach Design sehr unterschiedlich:

- Action-Klassen können ganze Geschäftsvorfälle abarbeiten. In diesem Fall wären sie Dispatcher, die im Wesentlichen die Arbeit von untergeordneten JavaBeans erledigen lassen.
- Action-Klassen können für einen Kanon von Operationen auf gleiche Datensätze erstellt werden, wie z.B. für das Erzeugen, Editieren, Anzeigen und Löschen von Daten verantwortlich sein.
- Action-Klassen können sehr feingranular entworfen werden, sodass sie jeweils nur einen kleinen Aspekt der Geschäftslogik abarbeiten und so Ketten von Struts-Action-Klassen nacheinander aufgerufen werden müssen, um eine Aufgabe vollständig zu erledigen.

Grundsätzlich gilt: Je feiner die Action-Klassen entworfen und implementiert werden, desto größer ist die Chance, dass Action-Klassen wiederverwendet werden können. Anderseits zeigt die Praxis:

Wiederverwendung von technischen Komponenten ist wahrscheinlicher als die Wiederverwendung von fachlichen Komponenten. Daraus könnte folgender Leitsatz abgeleitet werden: Verwenden Sie immer sehr feingranulare Action-Klassen bei technischen Komponenten. Bei fachlichen Komponenten sollten Operationen in Action-Klassen zusammengefasst werden (wie z.B. die klassischen Operationen `insert`, `update` und `delete`).

In real existierenden Projekten besteht immer die Gefahr, dass die `execute()`-Methoden völlig überfrachtet werden. Deshalb sollte sehr viel Gebrauch von den vorgefertigten Action-Klassen gemacht werden, die in Abschnitt 3.5 »Vordefinierte Action-Klassen« vorgestellt werden. Eine Action-Klasse wird grob gefasst immer folgende Dinge durchführen:

- Validierung des Status der User-Session. Normalerweise wird hier analysiert, ob ein User eingeloggt ist. Dies wird dadurch erreicht, dass ein User-Objekt in die Session gelegt wird. In diesem Fall kann ein Fehler ausgegeben werden oder direkt zur Login-Seite navigiert werden.
- Validierung der Form-Bean und Lesen von Werten aus der Form. Außerdem ist es zum Teil notwendig, Daten direkt aus dem Request zu lesen, um weitere Kontextinformationen zu bekommen.
- Zugriff auf die Geschäftslogik. Große Anwendungen werden Serviceobjekte oder Businesscontroller zur Verfügung stellen, um in »geordneter« Form auf die Geschäftslogik zugreifen zu können.
- Update der Action-Form. Die Ergebnisse aus der Geschäftslogik (Listen, Werte) müssen so aufbereitet werden, dass sie angezeigt werden können.
- Zurückgabe eines `ActionForward`-Objekts. Das Action-Forward ist der Return-Wert der `execute()`-Methode und zeigt an, welche Aktion als Nächstes durchgeführt werden soll.

3.4.2 Designtipps

Zu dem Umfang von Action-Klassen wurde bereits weiter oben etwas gesagt. In diesem Abschnitt geht es um grundsätzliche Tipps für das Design von Action-Klassen:

Code für eine multi-threaded Umgebung schreiben

Oft wird vergessen, dass der Controller für die gesamte Anwendung immer nur eine einzige Instanz der Action-Klasse erzeugt. Diese Instanz wird dann für alle Requests benutzt, die diese Action-Klasse benutzen wollen. Das bedeutet, Action-Klassen müssen thread-safe und reentrant (wiedereintrittsfähig) sein.

- **Lokale Variablen:** Die Methoden der Action-Klasse dürfen nur lokale Variablen und keine Instanzvariablen benutzen. Lokale Variablen werden auf dem Stack der JVM gespeichert. Für jeden Thread wird genau ein Stack erzeugt.
- **Lokale Methoden:** Die Übergabe von Werten an Methoden sollte ausschließlich über die Methodensignatur erfolgen. Auch diese Parameter werden auf dem Stack abgelegt, der immer nur einem Thread zugeordnet ist.
- **Pooling von Ressourcen:** Grundsätzlich kann es Skalierungsprobleme geben, wenn für jeden User Ressourcen, z.B. Datenbank-Connections, erzeugt und benutzt werden. Grundsätzlich sollten Ressourcen über Pools vergeben werden. Das `commons-pooling`-Framework bietet hier gute Möglichkeiten.
- **Synchronisierung:** Der schreibende Zugriff auf globale Ressourcen wie gemeinsam genutzte Variablen im Applikationskontext muss immer synchronisiert werden. Dabei ist zu beachten, dass Synchronisierung Zeit kostet.

Don't throw it – catch it

Die Action ist die vorletzte Instanz, in der Exceptions behandelt werden können. Grundsätzlich sollten hier möglichst alle Exceptions so behandelt werden, dass aussagefähige Fehler an der Oberfläche angezeigt werden.

Die letzte Instanz, der Request-Prozessor, kann nur noch Exceptions sinnvoll behandeln, für die es Exception-Handler gibt. Deshalb sollte von dieser Möglichkeit immer dann Gebrauch gemacht werden, wenn technische Exceptions (z.B. Datenbankfehler) auftreten.

Small is beautiful

In vielen Anwendungen explodiert der Code in den `execute()`-Methoden. Um diese Methoden besser zu strukturieren, gibt es mehrere Möglichkeiten:

- **Nutzung der vordefinierten Action-Klassen** (siehe Abschnitt 3.5 »Vordefinierte Action-Klassen«). Die meisten Fälle werden von diesen Klassen behandelt, sodass der Kontrollfluss nicht immer durch die `execute()`-Methode geführt wird.
- **Refactoring:** Werden Methoden zu groß, müssen sie mithilfe von Refactoring in »kleinere Stücke« geschnitten werden. Moderne IDEs wie Eclipse unterstützen diese Technik direkt.
- **Auslagerung der Geschäftslogik:** Geschäftslogik sollte in Service-Klassen ausgelagert werden. Die Action-Klasse fungiert dann nur noch als Dispatcher und enthält keine eigne Logik.

3.4.3 Das Action-Mapping

Mit dem Action-Mapping wird der Kontrollfluss der Struts-Anwendung festgelegt. Dabei wird festgelegt, wie eine entsprechende Aktion, dargestellt durch den Pfad (`/action.do`), auf eine Action-Instanz gemapped wird und wie der Kontrollfluss nach Bearbeitung der Action-Instanz weitergeführt wird.

Das Action-Mapping wird in der `<action-mappings>`-Sektion der `struts-config.xml` konfiguriert. Die Informationen über das Mapping werden in der Java-Klasse `ActionMapping` gekapselt. Die wichtigsten Parameter, die diese Klasse erwartet, sind:

- **type:** Der Typ der Java-Klasse, die die Action-Implementierung beherbergt. Die Klasse muss eine Subklasse von `org.apache.struts.action.Action` sein. Der Typ muss immer voll qualifiziert angegeben werden.
- **name:** Der Name einer Form-Bean-Instanz, die benutzt werden soll, um Daten zwischen dem Controller (hier der Action) und der JSP-Seiten zu transportieren. Form-Beans werden in der entsprechenden Sektion der `struts-config.xml` definiert (`<form-beans>`).

- **path:** Der Request-URI-Pfad, der zu diesem Mapping führt.

> Beachten Sie bitte, dass beim Extension-Mapping die Erweiterung (meistens `.do`) nicht in dem Pfad mit angegeben werden darf. Der Request-Prozessor filtert diese Erweiterung selbstständig aus.

- **unknown:** Eine Action darf das Attribut `unknown` auf `true` gesetzt haben. Dieses Attribut gibt an, dass diese Action die Haupt-Action einer Anwendung ist. Immer wenn kein konkretes Mapping gefunden wurde, wird die mit `unknown` bezeichnete Action aufgerufen.

> Diese Action sollte auf eine Hinweisseite zeigen, die vor allem dem Entwickler anzeigt, dass es Probleme mit dem Action-Mapping gegeben hat.

- **validate:** Gibt an, ob die Validate-Methode der Action-Form aufgerufen werden soll.
- **forward:** Der Request-URI-Pfad, zu dem verzweigt wird, nachdem die Action abgearbeitet wurde. Normalerweise wird das `type`-Attribut benutzt, um den Request an eine bestimmte Action weiterzuleiten.

Das folgende Beispiel zeigt ein typisches Action-Minimal-Mapping:

```
<action path="/initStart"
  type="de.gepackt.actions.login.InitStartAction"
  name="initStartForm">
  <forward name="hasSucceeded"
    path="/pages/start/start.jsp" redirect = "false"/>
</action>
```

Listing 3.4: Minimales Action-Mapping

Bei Aufruf der URI `initStart.do` wird der Request-Prozessor die Action-Instanz `InitStartAction` aufrufen. Die korrespondierende Form-Bean ist

die `initStartForm`. Nach Rückkehr der Action wird auf die JSP-Seite `start.jsp` verzweigt.

Typische Anwendungen benötigen *kein* komplizierteres Mapping, als in dem Minimal-Mapping vorgestellt.

Das Wildcard-Mapping

Große Anwendungen erfordern große Konfigurationsdateien. Mit der Einführung von Modulen kann die Unübersichtlichkeit von `struts-config.xml`-Dateien in Grenzen gehalten werden. Seit Struts 1.2 ist eine zusätzliche Erleichterung hinzugekommen: das Wildcard-Mapping. Mit diesem Mapping können Namenskonventionen, die in allen größeren Projekten eingesetzt werden, generisch umgesetzt werden. Beispiel:

- Jedes Pfad-Mapping, das einen Datensatz modifiziert, bekommt das Präfix `edit`, wie z.B. `/editCustomer.do`, `/editUser.do`, `/editScheduler.do`.
- Zu jedem dieser Mapping-Informationen wird eine `Action`-Klasse definiert, die `Edit*Action` heißt, also im vorherigen Beispiel `EditCustomerAction`, `EditUserAction` und `EditSchedulerAction`.
- Zu jeder Action gibt es eine korrespondierende Form, die ebenfalls den Namenskonventionen entsprechend `CustomerForm`, `UserForm` und `SchedulerForm` heißt. Dabei ist es unerheblich, ob diese Form-Klassen dynamisch oder statisch vereinbart werden.
- Im `success`-Fall wird auf eine JSP-Seite verzweigt, die generisch die Namen `Customer.jsp`, `User.jsp` und `Scheduler.jsp` heißt.

In diesem Fall ist es möglich, ein Wildcard-Mapping durchzuführen, das wie folgt vereinbar wird:

```
<action
    path="/edit*"
    type="de.gepackt.struts.action.Edit{1}Action"
    name="{1}Form"
    scope="request"
```

```
    validate="false">
    <forward
        name="failure"
        path="/menu.do"/>
    <forward
        name="success"
        path="/{1}.jsp"/>
</action>
```

Der Asterix (*) fungiert hier als Platzhalter und kann mit der Variablen {x} adressiert werden, wobei x für das x-te vereinbarte Wildcard steht. Dabei gelten folgende Regeln für Platzhalter:

*	Gilt für 0 oder mehr Zeichen, mit Ausnahme des Slash-Zeichens (/).
**	Gilt für 0 oder mehr Zeichen einschließlich des Slash-Zeichens.
\	Der Backslash wird verwendet, um anderen Zeichen zu ersetzen. * ist das Sternchen und \\ das Backslash-Zeichen selbst.

Wildcard-Mappings können für folgende Attribute eingesetzt werden:

- `type`, `roles`, `parameter`, `attribute`, `forward`, `include`, `input`

Das Action-`<forward>`-Element akzeptiert außerdem das Wildcard-Mapping im `path`-Attribut.

3.5 Vordefinierte Action-Klassen

Für immer wiederkehrende Aufgaben haben die Autoren des Struts-Frameworks im Laufe der Entwicklung der verschiedenen Versionen Standard-Action-Klassen entworfen, die in diesem Abschnitt vorgestellt werden.

Alle hier abgedruckten Codebeispiele können Sie als Webapplikation auf der Seite `http://www.struts-ge-packt.de/download/code.zip` oder auf der Webseite des Verlags `http://www.vmi-buch.de` downloaden.

3.5.1 Die DispatchAction

Wie bereits beschrieben, ist die `execute()`-Methode oft nicht mehr als ein Verteiler für unterschiedliche Funktionalitäten, die eine Action ausführen soll. Klassische Beispiele sind das Einfügen, Verändern und Löschen von Daten. Ohne die `DispatchAction` würde die `Action` für die verschiedenen Aufgaben wie folgt aufgerufen werden:

```
<html:link action="customer.do?method=add">
Sprung zur add-Methode </html:link><br>
<html:link action="customer.do?method=delete">
Sprung zur delete-Methode </html:link><br>
<html:link action="customer.do?method=modify">
Sprung zur modify-Methode
</html:link>
```

In der `execute()`-Methode würde dann eine Fallunterscheidung zu weiteren Methoden verzweigen. Die `DispatchAction` führt genau diese Fallunterscheidung durch und leitet dann automatisch zur richtigen Methode weiter.

Die `DispatchAction` ist abstrakt, kann also nur verwendet werden, indem eine neue Klasse erstellt wird. In dieser neuen Klasse wird für jede gewünschte Operation eine Methode zur Verfügung gestellt, die die gleiche Methodensignatur wie die `execute()`-Methode hat.

```
public class DispatchAction extends org.apache.struts.actions.Dis-
patchAction {
public ActionForward add(ActionMapping mapping,
                         ActionForm form,
                         HttpServletRequest request,
                         HttpServletResponse response)
            throws Exception {
    PropertyUtils.setSimpleProperty(form,
        "hinweis", "add");
        return mapping.findForward("hasSucceeded");
}
```

```
public ActionForward delete(ActionMapping mapping,
                        ActionForm form,
                        HttpServletRequest request,
                        HttpServletResponse response)
            throws Exception {
    PropertyUtils.setSimpleProperty(form,
        "hinweis", "delete");
        return mapping.findForward("hasSucceeded");
}
public ActionForward modify(ActionMapping mapping,
                        ActionForm form,
                        HttpServletRequest request,
                        HttpServletResponse response)
            throws Exception {
    PropertyUtils.setSimpleProperty(form,
        "hinweis", "modify");
        return mapping.findForward("hasSucceeded");
}
```

In der Struts-Konfiguration wird für alle Operationen, die diese Action durchführen, nur noch ein Eintrag konfiguriert:

```
    <action path="/customer"
         name="dataForm"
         type="de.gepackt.struts.actions.DispatchAction"
         parameter="method"
         scope="session">
         <forward name="hasSucceeded"
                  path="/pages/DispatchResult.jsp"/>
    </action>
```

Die Bezeichnung dieses Parameters, hier `method`, können Sie frei wählen.

Aufgerufen wird diese Action jetzt z.B. in einer Form:

```
<html:form action="customer?method=add">
```

Die `DispatchAction` würde in diesem Fall die Methode `add()` der eigenen `DispatchAction` aufrufen.

3.5.2 Die ForwardAction

In der MVC-Architektur gilt: Jeder Aufruf muss durch den Controller geführt werden. Damit verbieten sich direkte Aufrufe von einer JSP-Seite zu einer anderen. Diese Regel ist vernünftig, weil – wie in Abschnitt 3.3 »Der Request-Prozessor« gesehen – der Request-Prozessor eine Menge von Regeln abarbeitet, die sicherstellen, dass eine bestimmte Action aufgerufen werden darf. Besonders wenn mit Sicherheitsregeln (security roles) gearbeitet wird, ist eine solche Vorgehensweise unumgänglich.

Die `ForwardAction` ist generisch, muss also nicht angepasst werden und kann so benutzt werden. Die Vorgehensweise, wenn von einer JSP-Seite zu einer anderen gesprungen werden soll, ist immer die gleiche:

- Anstatt direkt zur JSP-Seite zu navigieren, wird eine Action aufgerufen.

```
<html:link action="/showDispatchSite.do">
DispatchAction</html:link>
```

- Die angesprochene Action leitet den Aufruf mittels der `ForwardAction` zur JSP-Seite weiter.

```
<action path="/showDispatchSite"
   parameter="/pages/Dispatch.jsp"
   type="org.apache.struts.actions.ForwardAction"/>
```

An dieser Stelle können jetzt auch Sicherheitsregeln eingebaut werden.

> Ohne eine Diskussion über gutes und schlechtes Design anfangen zu wollen: Auch wenn die `ForwardAction` umständlich erscheint, ist es wichtig, dass im Struts-Framework alle Aufrufe durch den Controller geführt werden.

Hier wird das richtige Modul ausgesucht und zu dem Modul das richtige Resource-Bundle, das wiederum benötigt wird, um Texte länderspezifisch auszugeben.

Die oben gezeigte Möglichkeit, das Forward in der `struts-config.xml` zu spezifizieren, kann noch vereinfacht werden, indem das `forward`-Attribut genutzt wird:

```
<action path="/showDisplaySite"
        forward="/pages/Dispatch.jsp "/>
```

Diese beiden Aufrufe sind synonym. Das bedeutet, auch wenn mit dem `forward`-Attribut gearbeitet wird, ruft Struts intern die `ForwardAction` auf.

An dieser Stelle soll noch auf eine andere Möglichkeit hingewiesen werden: Der Verweis auf ein `<global-forward>`-Element in der `struts-config.xml`:

```
<html:link forward="welcome">
Sprung zur Hauptseite per globalem Forward
</html:link><br>
```

Der Eintrag in der `struts-config.xml` sieht dann wie folgt aus:

```
<global-forwards>
  <forward name="welcome"
   path="/pages/Welcome.jsp"/>
</global-forwards>
```

Wenn Struts hinter einem Loadbalancer läuft, ist es besonders wichtig, dass bereits der erste Aufruf durch das Action-Servlet geleitet wird, damit bereits die erste aufgerufene JSP-Seite eine gültige Session-ID hat. Ansonsten kann der Balancer weitere Aufrufe nicht an die richtige Instanz des Webcontainers weiterrouten.

3.5.3 Die IncludeAction

Die `IncludeAction` funktioniert genauso wie die `ForwardAction`. Deshalb werden die verschiedenen Modelle, Vor- und Nachteile hier nicht noch einmal wiederholt. Im Gegensatz zur `ForwardAction` wird jedoch nicht auf eine andere Seite verzweigt, sondern eine weitere Seite eingebettet (inkludiert). Dazu muss in der JSP-Seite ein `<jsp:include>`-Tag eingebettet werden:

```
<jsp:include page="/includeHeaderForward.do"/>
```

Die Definition der Action sieht dann wie folgt aus:

```
<action path="/includeHeaderForward"
        type="org.apache.struts.actions.IncludeAction"
        parameter="/pages/HeaderForward.jsp"
/>
```

Die Seite `HeaderForward.jsp` wird dann in die aufrufende Seite eingefügt.

3.5.4 Die LocaleAction

Die `LocaleAction` ist mit Struts 1.2 dem Framework hinzugefügt worden. Mit dieser Action ist es möglich, die Sprach- und Ländereinstellungen für einen User zu ändern.

Die `LocaleAction` erwartet eine Form-Bean, in der folgende Parameter vorhanden sind:

- **language:** Die Angabe der Sprache. Die Sprache wird dabei mit kleinen Buchstaben geschrieben wie `de` für Deutsch, `en` für Englisch usw. Der `language`-Parameter ist als einziger Parameter notwendig; die beiden anderen Parameter können auch `null` sein.
- **country:** Eine Länderangabe. Die Länderangabe erfolgt in Großbuchstaben wie `DE` für Deutschland, `US` für USA.
- **page:** Page gibt eine Forward-URI an, zu der nach der Sprachumschaltung verzweigt wird. Ist `page` leer oder null, wird zu dem Forward-`success` verzweigt.

Insgesamt ist eine Sprachumschaltung notwendig, in dieser Form aber nicht wirklich nützlich, weil es nicht möglich ist, generisch die Sprache umzuschalten. Besser wäre die Möglichkeit, eine Sprachumschaltung so zu implementieren, dass sie in jede Seite eingebaut werden kann und immer ohne zusätzliches Konfigurationswissen funktioniert.

Falls die Sprachumschaltung häufiger gebraucht wird, bietet es sich an, eine Basis-Form mit den drei Properties zu entwerfen und immer Subklassen dieser Basis-Form zu bilden. Im Beispiel haben wir uns für eine dynamische Form entschieden:

```
<form-bean
  name="languageForm"
  type="org.apache.struts.action.DynaActionForm">
  <form-property
    name="language"
    type="java.lang.String"/>
  <form-property
    name="country"
    type="java.lang.String"/>
  <form-property
    name="page"
    type="java.lang.String"/>
</form-bean>
```

Wir haben außerdem ein einfaches Formular entworfen, in dem die Sprache eingegeben werden kann.

```
<html:form action="/processLanguage">
<table border="0">
<tr><td>
   <bean:message key="language.switch"/>
</td>
```

```
<td>
   <html:select property="language" size="1">
       <html:option key="label.german" value="de"/>
       <html:option key="label.english" value="en"/>
   </html:select>
   <html:hidden property="page" value="/menu.do"/>
</td></tr>

<tr>
<td align="center">
   <html:submit property="btnsubmit">
   <bean:message key="button.submit"/>
   </html:submit>
</td></tr>
</table>
</html:form>
```

In der `struts-config.xml` muss jetzt noch auf den Pfad reagiert werden.

```
<action path="/processLanguage"
        name="languageForm"
        type="org.apache.struts.actions.LocaleAction">
        <forward name="success" path="/menu.do"/>
</action>
```

3.5.5 Die LookupDispatchAction

Die `LookupDisplayAction` ist eine Subklasse der `DispatchAction`, also auch für die Verteilung von Requests zuständig. In diesem Fall geht es darum, mehrere Submit-Buttons auf einer JSP-Seite zu unterstützen.

Genau wie bei der `DispatchAction` können in einer Klasse mehrere Methoden vereinbart werden, die alle die gleiche Signatur haben müssen wie die `execute()`-Methode. Anders als bei der `DispatchAction` wird das Mapping zwischen Aufruf und Methode jedoch nicht mit direkter Hilfe eines Request-Parameters durchgeführt, sondern durch einen »Lookup« in einem

Resource-Bundle. Daher kommt der Name dieser Action. Interessant für dieses Mapping ist der Wert des Buttons (das `value`-Attribut). Da bei einer mehrsprachigen Anwendung dieser Wert ein Eintrag im Resource-Bundle sein wird, findet das Mapping mithilfe dieses Schlüssels statt.

In unserem Beispiel vereinbaren wir zwei Buttons: Der eine verzweigt auf die »Forward-Demo«, der andere auf die »Dispatch-Demo«:

```
<html:form action="/processLookupDispatch">
  <p>
  <html:submit property="lookup">
    <bean:message key="button.toforward"/>
  </html:submit>

  <html:submit property="lookup">
    <bean:message key="button.todispatch"/>
  </html:submit>
  </p>
</html:form>
```

Die beiden Schlüssel für die Buttons werden in der entsprechenden Message-Property-Datei hinterlegt:

```
button.toforward=Forward-Demo
button.todispatch=Dispatch-Demo
```

In der `struts-config.xml` wird jetzt festgelegt, dass bei Auslösen eines Buttons auf die selbst geschriebene `LookupDispatchAction` verzweigt werden soll:

```
<action path="/processLookupDispatch"
  parameter="lookup"
  name="dataForm"
  type="de.gepackt.struts.actions.LookupDispatchAction">
  <forward name="forward" path="/pages/Forward.jsp"/>
  <forward name="dispatch" path="/pages/Dispatch.jsp"/>
</action>
```

Beachten Sie bitte, dass der Parameter frei wählbar ist und nur mit dem `property`-Attribut im Submit-Button übereinstimmen muss. In diesem Fall wurde als Parametername `lookup` gewählt.

Fehlt noch die überladene `LookupDispatchAction`, die einerseits das Mapping und andererseits die Methoden enthält:

```
public class LookupDispatchAction extends org.apache.struts.
  actions.LookupDispatchAction {
  protected Map getKeyMethodMap() {
    Map map = new HashMap();
    map.put("button.toforward", "doForward");
    map.put("button.todispatch", "doDispatch");
    return map;
  }
  public ActionForward doForward(ActionMapping mapping,
          ActionForm form,
          HttpServletRequest request,
          HttpServletResponse response)
     throws Exception {
        return mapping.findForward("forward");
 }
  public ActionForward doDispatch(ActionMapping mapping,
          ActionForm form,
          HttpServletRequest request,
          HttpServletResponse response)
     throws Exception {
        return mapping.findForward("dispatch");
 }
}
```

3.5.6 Die MappingDispatchAction

Die `MappingDispatchAction` ist die dritte Verteiler-Action, die Struts von Haus aus mitbringt. Die `DispatchAction` hat nur einen Eintrag in der `struts-config.xml` und es wird dynamisch über einen frei wählbaren Parameter die Methode angegeben, zu der verzweigt werden soll. Bei der `MappingDispatchAction` wird bereits in der `struts-config.xml` festgelegt, welche Methode aufgerufen wird. Dadurch ist es möglich, verschiedene gleichartige Actions in einer Klasse zu bündeln.

> Im Grunde genommen werden hier die `execute()`-Methoden verschiedener Action-Klassen in einer Klasse zusammengefasst. Es wird also nicht verteilt, sondern gebündelt. Dies kann sinnvoll sein, wenn mehrere Action-Klassen die gleichen Hilfsmethoden benötigen, diese aber nicht durch eine Helperklasse zur Verfügung gestellt werden sollen.

Wie bei den anderen Dispatch-Actions auch, ist die `MappingDispatchAction` abstrakt und es muss eine konkrete Subklasse gebildet werden. Wieder muss für jeden Pfad eine Methode implementiert werden, die der Signatur der `execute()`-Methode entspricht.

```
public class MappingDispatchAction extends org.apache.struts.
  actions.MappingDispatchAction {
  public ActionForward add(ActionMapping mapping,
          ActionForm form,
          HttpServletRequest request,
          HttpServletResponse response)
      throws Exception {
      return mapping.findForward("success");
  }
}
```

In der `struts-config.xml` wird nun genau festgelegt, zu welcher Methode in der Action verzweigt werden soll:

```
<action path="/dispatchToDispatch"
  type="de.gepackt.struts.actions.MappingDispatchAction"
  name="dataForm"
  scope="request"
  parameter="add">
  <forward name="success" path="/menu.do"/>
</action>
```

3.5.7 Die SwitchAction

Seit Struts 1.1 ist es möglich, mehrere Module zu einer Struts-Anwendung zusammenzuführen. Dabei hat jedes Modul eine eigene Konfigurationsdatei mit allem, was dazu gehört. In Kapitel 2.3 »Struts Multiple Application Support«, ist ausführlich darauf eingegangen worden, wie mehrere Struts-Module konfiguriert werden und welche anderen Möglichkeiten als die `SwitchAction` es gibt, den Modulwechsel zu veranlassen. Deshalb hier kurz zusammengefasst den Modulwechsel durch die `SwitchAction`.

Die `SwitchAction` erwartet zwei Parameter:

- **prefix:** Gibt den Namen des Moduls an, in das gewechselt werden soll. Bleibt der Wert leer, so wird in das Default-Modul verzweigt.
- **page:** Gibt an, welche Seite im neuen Modul aufgerufen werden soll.

Die Aktionskette zum Modulwechsel mit der `SwitchAction` sieht dann wie folgt aus:

- In der JSP-Seite wird ein globales Forward für den Modulwechsel angesprungen:

```
<html:link forward="switchModul">
SwitchAction</html:link>
```

Natürlich kann dieses Forward auch ein lokales Forward als Ergebnis einer Action sein.

Dieses Forward wird in der `struts-config.xml` auf eine Action verzweigt, die die `SwitchAction` benutzt. Die beiden Parameter, die übergeben werden müssen, werden hier gesetzt.

```
<forward name="switchModul"
  path="/modul/switch.do?prefix=/modul&amp;page=/pages/Wel-
come.jsp"
  contextRelative="true"/>
```

Beachten Sie, dass der Parameter `contextRelative` auf `true` gesetzt wurde. Dieser Parameter gibt an, dass der Pfad nicht relativ zum Modul, sondern relativ zur gesamten Webanwendung gesetzt wird.

- Die Action-Definition sieht in diesem Fall wie folgt aus:

```
<action path="/switch"
  type="org.apache.struts.actions.SwitchAction"/>
```

Damit ist der Modulwechsel vollzogen. Der Rückweg muss dann in der `struts-config.xml` des anderen Moduls konfiguriert werden.

3.6 Exception-Handler

Der Controller ist grundsätzlich auch für die Behandlung von Exceptions verantwortlich. Struts bietet die Möglichkeit, sowohl global für die gesamte Anwendung als auch lokal für eine bestimmte Action-Klasse einen Exception-Handler zu installieren. Der Handler wird dann im Request-Prozessor mithilfe der Methode `processException()` zur Ausführung gebracht. Soll bei einer aufgetretenen Exception immer zu einer bestimmte Seite verzweigt werden, so braucht kein neuer Exception-Handler implementiert zu werden. Der Default-Handler hat folgende Funktionalität:

- Wenn das `path`-Attribut gesetzt ist, wird zu der angegebenen Ressource (Action oder JSP-Seite) verzweigt.

```
<exception
     key="errors.global.check"
     type="de.gepackt.exception.PersistenceException"
     path="/pages/error/persistenceError.jsp">
</exception>
```

Listing 3.5: Exception-Handler: Forward zu einer Ressource

- Wird dieser Pfad nicht angegeben, wird als Forward das Input-Forward des Mappings ausgefiltert.
- Aus der Exception wird der Fehler und die Property (falls angegeben) extrahiert. Der Fehler wird unter dem Schlüssel `Globals.ERROR_KEY` je nach Scope der Action im Request- oder im Sessionkontext hinterlegt.
- Das Forward wird ausgeführt.

Soll das Verhalten des Exception-Handlers verändert werden, gibt es die Möglichkeit, einen eigenen Handler zu schreiben.

```
<exception
     key="errors.global.check"
     type="de.gepackt.exception.PersistenceException"
     handler="de.gepackt.ExceptionHandler">
</exception>
```

Listing 3.6: Selbst definierter Exception-Handler

3.7 Das Struts-Chains-Framework

Wie wir in Abschnitt 3.3 »Der Request-Prozessor« beschrieben haben, arbeitet der Request-Prozessor eine ganze Reihe von Methoden ab. Jede einzelne Methode entscheidet durch den Rückgabewert, ob die Arbeit bereits getan ist oder ob noch weitere Methoden abgearbeitet werden. Diese Form der Abarbeitung von Methoden ist ein typisches Beispiel für das Entwurfsmuster »Zuständigkeitskette«, englisch »chain of responsibility«.

3.7.1 Das »chain of responsibility«-Entwurfsmuster

Der Zweck des Entwurfsmusters ist laut Gamma et al. wie folgt beschrieben: »Vermeide die Kopplung des Auslösers einer Anfrage mit seinem Empfänger, indem mehr als ein Objekt die Möglichkeit enthält, die Aufgabe zu erledigen. Verkette die empfangenen Objekte und leite die Anfrage an der Kette entlang, bis ein Objekt sie erledigt« (Gamma et al.: Entwurfsmuster; Addison-Wesley 1996).

Die Anfrage ist im Struts-Framework der HTTP-Request, die einzelnen Objekte der Zustandskette die `process()`-Methoden.

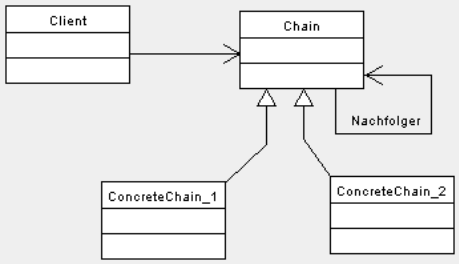

Abbildung 3.4: Struktur des Zuständigkeitskettenmusters

Im Struts-Chains-Framework sind die einzelnen Objekte der Zustandskette wie folgt definiert (für einige `process()`-Methoden gibt es kein Äquivalent mehr, so z.B. für die `processMultipart()`-Methode; dieser Mechanismus wird zukünftig in dem Servlet direkt ablaufen):

Kommando	Beschreibung
`AuthorizeAction`	Ermittelt, ob die Action überhaupt betretbar ist. Dies wird dadurch erreicht, das Security Roles abgefragt werden. Ersetzt die Methode `processRoles()` im Request-Prozessor.

Kommando	Beschreibung
`CreateActionForm`	Erzeugt eine Action-Form-Instanz. Ersetzt die Methode `processActionForm()` im Request-Prozessor.
`ExceptionHandler`	Handler zur Abarbeitung von Exceptions, die in einer Action auftreten können. Ersetzt die Methode `processException()` im Request-Prozessor.
`ExecuteAction`	Führt eine Action-Instanz aus. Ersetzt die Methode `processActionPerform()` im Request-Prozessor.
`LookupCommand`	Generischer Command, der verschiedene Funktionalitäten hat. Ersetzt die Methode `processPreprocess()` im Request-Prozessor.
`PerformForward`	Führt ein Forward auf eine andere Ressource aus. Ersetzt die Methode `processForward()` im Request-Prozessor.
`PerformInclude`	Führt ein Include auf eine andere Ressource aus. Ersetzt die Methode `processInclude()` im Request-Prozessor.
`PopulateActionForm`	Überträgt die Daten eines Requests in die korrespondierende ActionForm. Ersetzt die Methode `processPopulate()` im Request-Prozessor.
`RequestNoCache`	Setzt einen no-cache-Mimetyp, wenn der Controller entsprechend konfiguriert wurde. Ersetzt die Methode `processNoCache()` im Request-Prozessor.
`SelectAction`	Cached die Action-Konfiguration für diese Action, damit sie für weitere Aktionen zur Verfügung steht. Ersetzt die Methode `processPath()` im Request-Prozessor.
`SelectForward`	Erzeugt eine Forward-Konfiguration und gibt diese zurück.

Kommando	Beschreibung
`SelectInput`	Sucht das `Input`-Attribut, das in der Forward-Konfiguration angegeben wurde.
`SelectLocale`	Setzt das `Locale`-Objekt, das für diesen Request genutzt werden soll. Ersetzt die Methode `processLocale()` im Request-Prozessor.
`SelectModule`	Speichert die Modulkonfiguration und die Message-Ressourcen der Subapplikation, die für diesen Request verantwortlich ist.
`SetContentType`	Checkt, ob ein Content-Typ gesetzt ist, und setzt diesen in den Response. Ersetzt die Methode `processContent()` im Request-Prozessor.
`TilesPreProcessor`	Hat die Verantwortlichkeit des `TilesRequestProcessor` aus Struts 1.1. Dieser Command reichert den Chain-Kontext mit weiteren Informationen an. Ersetzt den `TilesRequestProcessor`.
`ValidateActionForm`	Validiert die Action-Form. Ersetzt die Methode `processValidate()` im Request-Prozessor.

3.7.2 Commons-Chains

Das Struts-Chains-Framework stützt sich auf das Jakarta `commons-chains` Framework ab. `commons-chains` implementiert dabei die grundlegenden Bestandteile des Zuständigkeitsketten-Entwurfmusters. Wir werden hier kurz die wenigen Kern-Interfaces dieses Frameworks beschreiben.

> Auch wenn Chains in der aktuellen Version des Frameworks noch nicht verwendet werden, wird dieses Feature ab Struts 1.3 zum festen Bestandteil des Frameworks. Außerdem ist in Planung, die Geschäftslogik ebenfalls mit diesen Mechanismen anzubinden. Von daher lohnt die Beschäftigung mit `commons-chains` auf jeden Fall.

Des Weiteren haben Sie jetzt schon die Möglichkeit, mithilfe von Chains individuelle Request-Prozessor-Objekte zu konfigurieren.

- **Command:** Der `Command` kapselt einen Auftrag, z.B. einen HTTP-Request. Die Aufgabe des `Command`-Objekts ist es, Daten zu betrachten oder zu verändern. Diese Daten werden durch einen `Context` repräsentiert.
- **Chain:** Ausgehend von der Konzeption ist der `Chain` die Ansammlung von `Command`-Objekten, die aber nach außen wie ein einzelnes `Command` agieren. Das `Chain`-Interface erweitert den `Command`; ein `Chain` ist also auch ein `Command`. Hier wird das Composite-Entwurfsmuster angewendet.
- **Context:** Ein Kontext hält Statusinformationen und wird von `Chain`- oder `Command`-Objekten verändert und ausgelesen. Typischerweise werden Context-Informationen mithilfe von JavaBean-Properties ausgelesen und gesetzt.

Im `commons-chains` Framework wird der Zugriff auf Properties mithilfe von Attribute-Property-Transparency ermöglicht. Dieses bedeutet, dass es für einen Client völlig transparent ist, ob ein Wert in einer Hashmap gespeichert ist oder direkt über `get()`- und `set()`-Methoden verfügbar ist. Es ist immer möglich, über

```
java.lang.Object get(java.lang.Object key);
```

auf alle Attribute zuzugreifen.

3.7.3 Konfiguration von Chains

Struts-Chains werden in der `chain-config.xml` konfiguriert. Wenn das Framework ab der Version 1.3 vollständig integriert ist, wird sich hier vielleicht noch etwas ändern. Streng nach der Konzeption, die eben vorgestellt wurde, werden dabei `chain`-Elemente konfiguriert, die ihrerseits `Command`-Objekte beinhalten. Die Funktionen des normalen Request-Prozessors werden in Struts-Chains wie folgt konfiguriert (Auszug; den kompletten Code entnehmen Sie bitte der Originaldokumentation):

```
  <chain      name="servlet-standard">
     <!-- Look up optional preprocess command -->
     <command
     className=
     "org.apache.commons.chain.generic.LookupCommand"
     catalogKey="catalog"
     name="servlet-standard-preprocess"
     optional="true"/>
     <!-- Establish exception handling filter -->
     <command
     className=
     "org.apache.struts.chain.ExceptionCatcher"
     exceptionCommand="servlet-exception"/>
     <!-- Identify the Locale for this request -->
     <command
     className=
     "org.apache.struts.chain.servlet.SelectLocale"/>
... und so weiter
</chain>
```

Listing 3.7: Konfiguration von `chains` *und* `commands`

Innerhalb der `struts-config.xml` wird dann das Struts-Chains-Framework an zwei Stellen konfiguriert:

```
  <controller>
    <set-property property="inputForward" value="true"/>
    <set-property
          property="processorClass"
        value="org.apache.struts.chain.
              legacy.ComposableRequestProcessor"/>
  </controller>
  <plug-in
     className="org.apache.struts.chain.
     legacy.CatalogConfiguratorPlugIn">
```

```
    <set-property
       property="resource"
       value="org/apache/struts/chain/chain-config.xml"/>
  </plug-in>
```

Listing 3.8: Konfiguration von `chains` *in der* `struts-config.xml`

3.7.4 Bewertung von Struts-Chains

Struts-Chains ist die Zukunft von Struts und wird so schnell wie möglich in das Framework integriert. Die Vorteile liegen auf der Hand:

- Die Abarbeitung und die Reihenfolge von Requests kann sehr genau auf eine Anwendung oder ein Modul der Anwendung abgestimmt werden. So können überflüssige Schritte ganz herausgelassen werden, um so die Performance zu erhöhen.
- Zusätzliche Schritte (z.B. Filter) können flexibel integriert werden.
- Durch Chains können andere Präsentationsmechanismen wie Faces, Velocity etc. sehr schnell in das Struts-Framework integriert werden, weil die view-spezifischen Dinge ausschließlich im Request-Prozessor abgearbeitet werden.