HANSER

Leseprobe

Donald Brown, Chad Michael Davis, Scott Stanlick

Struts 2 im Einsatz

Übersetzt aus dem Englischen von Jürgen Dubau

ISBN: 978-3-446-41575-1

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-41575-1

sowie im Buchhandel.

© Carl Hanser Verlag, München

3

3 Die Arbeit mit Actions

Die Themen dieses Kapitels:

- Actions in Pakete bündeln
- Actions implementieren
- `ModelDriven`-Actions und mit Objekten gepufferte Properties
- Dateiupload

Nachdem wir die Übersichten und Einführungen hinter uns haben, wollen wir uns nun an die Kernkomponenten von Struts 2 machen, und da zuerst einmal an die Actions. Wie wir bereits erfahren haben, erledigen die Actions die zentralen Arbeitsschritte für jede Anfrage. Sie enthalten die Businesslogik und die Daten und wählen dann das Result aus, das die Result-Seite rendern soll. Dieses Framework ist actionorientiert: Actions stehen im Mittelpunkt. Letzten Endes werden Sie einen Großteil Ihrer Zeit als Struts-2-Entwickler bei der Arbeit mit Actions zubringen.

In diesem Kapitel erfahren Sie alles Wissenswerte, um mit dem Erstellen eigener Actions für Ihre Anwendungen beginnen zu können. Unter Verwendung des auf XML basierenden Mechanismus für die deklarative Architektur erkunden wir alle verfügbaren Optionen, wenn wir Actions deklarieren, schauen uns einige Convenience-Klassen an, die bei der Programmierung von Actions helfen, und stellen die wichtigsten Arten vor, wie Daten in der Action übertragen werden. Das vorige Kapitel hat mit JavaBeans-Properties gearbeitet. Wir werden auch sehen, wie Interceptors in Zusammenarbeit mit Actions vieles von der Funktionalität des Frameworks umsetzen. Am Schluss steht die praktische Fallstudie einer Dateiupload-Action. Wir beginnen auch mit der Entwicklung der Beispielanwendung Struts 2 Portfolio, um die Konzepte und Techniken dieses Kapitels zu demonstrieren.

Wie sich herausstellt, können wir Actions nicht vorstellen, ohne ziemlich viel anderes Material einzuführen. Wir werden uns dabei so gut wie möglich nur auf die Actions konzentrieren. Doch wahrscheinlich werden Sie genug von den anderen Sachen mitkriegen, um Ihre eigene Struts-2-Anwendung ans Laufen zu bekommen. In diesem Kapitel gibt es eine ganze Menge zu lernen. Einmal tief Luft holen, dann kann’s gleich losgehen.

3.1 Die Actions in Struts 2

Als Grundlage umreißen wir die Rolle, die Actions im Framework spielen. Wir erklären den Zweck und die verschiedenen Rollen der Actionkomponente. Dann vergleichen wir die Action von Struts 2 mit der gleichnamigen Komponente in Struts 1. Und wir studieren die Aufgaben, die ein Objekt in der Rolle einer Action bei dem Framework insgesamt übernimmt. Struts 2 ist ein Verfechter der Gleichbehandlung. Jede Klasse kann eine Action sein, solange sie ihren Verpflichtungen gegenüber dem Framework nachkommt. Schauen wir uns an, was diese wichtigen Komponenten für das Framework machen.

3.1.1 Was macht eine Action?

Actions können drei Sachen: Zuerst (wie Sie nun schon wissen) ist aus Sicht der Architektur des Frameworks die wichtigste Rolle einer Action, dass sie die eigentliche Arbeit kapselt, die für eine bestimmte Anfrage zu erledigen ist. Die zweite Rolle ist, im automatischen Datentransfer des Frameworks von der Anfrage zur View als Kurier zu dienen. Schließlich muss die Action das Framework bei der Bestimmung unterstützen, welches Result die View rendern soll, die als Reaktion auf die Anfrage zurückgegeben wird. Schauen wir uns an, wie die Actionkomponente alle diese verschiedenen Rollen erfüllt.

Übrigens werden wir unsere Punkte in den folgenden Abschnitten mit Beispielen aus der HelloWorld-Anwendung von Kapitel 2 demonstrieren. Aber keine Sorge – ein paar Seiten weiter hinten beginnen wir mit der Erstellung der echten Struts 2 Portfolio-Anwendung.

3.1.1.1 Actions kapseln den Unit of Work

Weiter vorne in diesem Buch haben wir gesehen, dass die Action die Rolle des MVC-Models für das Framework erfüllt. Eine der zentralen Aufgaben dieser Rolle ist, die Businesslogik zu enthalten; zu diesem Zweck nutzen Actions die `execute()`-Methode. Der Code in dieser Methode soll sich nur mit der Logik der Arbeit beschäftigen, die zu dieser Anfrage gehört. Das folgende Code-Snippet aus der HelloWorld-Anwendung des vorigen Kapitels zeigt die von `HelloWorldAction` erledigte Arbeit:

```
public String execute()  {
    setCustomGreeting( GREETING + getName() );

  return "SUCCESS";
}
```

Diese Action soll den Benutzer persönlich begrüßen. Wie man sehen kann, erstellt diese `execute()`-Methode nur diese Begrüßung und mehr nicht. In diesem Fall macht die Businesslogik kaum mehr als eine Verkettung. Wäre es komplexer, hätten wir diese Logik sicher in eine Businesskomponente verschoben und diese Komponente in die Action injiziert. Das Framework unterstützt die Verwendung von Abhängigkeitsinjektionen (*dependency injections*). Damit wird solcher Code wie Actions saubergehalten und entkoppelt. Wir werden später einige Techniken lernen, die die Spring-Integration des Frameworks für

die Injektion dieser Komponenten einsetzen. Jetzt sollten Sie nur im Hinterkopf behalten, dass Actions unsere Businesslogik enthalten oder zumindest den Eintrittspunkt in die Businesslogik, und dass sie diese Logik so rein und kurz wie möglich halten sollen.

3.1.1.2 Actions bieten einen Ort für den Datentransfer

Weil die Action die Model-Komponente des Frameworks ist, heißt das auch, dass ihre Aufgabe ist, die Daten zu transportieren. Vielleicht glauben Sie, dass die Actions dadurch verkompliziert werden, aber sie bleiben tatsächlich eher sauberer. Weil die Daten für die Action lokal vorgehalten werden, stehen sie praktischerweise während der Ausführung der Businesslogik stets zur Verfügung. Ein paar JavaBeans-Properties könnten der Action noch einige Zeilen Code mehr mitgeben, doch wenn die `execute()`-Methode die Daten in diesen Properties referenziert, macht die Nähe der Daten diesen Code umso prägnanter.

Listing 3.1 (auch von `HelloWorldAction`) zeigt den Code, mit dem diese Action Anfragedaten transportieren kann.

Listing 3.1 Transfer von Anfragedaten in die JavaBeans-Properties der Action

```
private String name;
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
private String customGreeting;
public String getCustomGreeting() {
    return customGreeting;
}
public void setCustomGreeting( String customGreeting ){
    this.customGreeting = customGreeting;
}
```

Die Action implementiert lediglich JavaBeans-Properties für alle Daten, die es zu transportieren wünscht. Wir haben dies zusammen mit der HelloWorld-Anwendung in Aktion gesehen. Anfrageparameter aus dem Formular werden in die Properties mit den entsprechenden Namen verschoben. Wie bereits gesehen, erledigt das Framework das automatisch. In diesem Fall wird der Parameter `name` aus dem Formular zur Namensabfrage auf die Property `name` gesetzt. Zusätzlich zum Empfangen der eintreffenden Daten aus der Anfrage werden diese JavaBeans-Properties in der Action die Daten auch dem Result zur Verfügung stellen. Die Logik der `HelloWorld`-Action stellt die persönliche Begrüßung bei der Property `customGreeting` ein, und damit wird sie auch für das Result verfügbar.

Neben diesen einfachen JavaBeans-Properties gibt es noch einige andere Techniken für die Verwendung der Action als Datentransferobjekt. Wir werden diese Alternativen später in diesem Kapitel untersuchen und darüber hinaus auch die Mechanismen des eigentlichen Datentransfers. Momentan soll nur festgehalten werden, dass die Action als ein zentralisiertes Datentransferobjekt dient, über das die Anwendungsdaten in allen Schichten des Frameworks verfügbar sind.

Die Verwendung von Actions als Datentransferobjekte lässt vielleicht bei manchen wachsamen Struts-1-Entwicklern die Alarmglocken schellen. Bei Struts 1 gibt es nur eine In-

stanz einer bestimmten Action-Klasse. Wenn das immer noch gelten würde, könnten wir das Actionobjekt nicht als Datenkurier für die Anfrage nehmen. In einer Multithread-Umgebung wie einer Webanwendung wäre es problematisch, Daten wie angesprochen in Instanzfeldern zu speichern. Struts 2 löst dieses Problem, indem für jede Anfrage eine neue Instanz der Klasse dafür erstellt wird. Durch diesen fundamentalen Unterschied können Struts-2-Objekte als dedizierte Datentransferobjekte für jede Anfrage existieren.

3.1.1.3 Actions geben Control-String fürs Result-Routing zurück

Die letzte Pflicht einer Actionkomponente ist, einen Control-String zurückzugeben, der das zu rendernde Result wählt. Frühere Frameworks haben Routing-Objekte an die Einstiegsmethode der Action übergeben. Durch Rückgabe eines Control-Strings sind solche Objekte nicht mehr erforderlich. Das führt zu einer sauberen Signatur und einer Action, die viel weniger an einen speziellen Routing-Code gekoppelt ist. Der Wert des Rückgabe-Strings muss mit dem Namen des gewünschten Results übereinstimmen, wie es in der deklarativen Architektur konfiguriert wurde. Die `HelloWorldAction` gibt beispielsweise den String `"SUCCESS"` zurück. Wie Sie unserer XML-Deklaration entnehmen können, ist `SUCCESS` der Name einer der Result-Komponenten.

```
<action name="HelloWorld" class="manning.chapterOne.HelloWorld">
    <result name="SUCCESS">/chapterTwo/HelloWorld.jsp</result>
    <result name="ERROR">/chapterTwo/Error.jsp</result>
</action>
```

Die HelloWorld-Anwendung bestimmt über eine einfache Logik, welches Result sie wählen wird. Tatsächlich wählt sie stets `"SUCCESS"`. Die meisten realen Actions haben einen komplexeren Bestimmungsprozess, und zu den Result-Entscheidungen wird beinahe immer irgendeine Art Fehler-Result gehören, um mit Problemen umzugehen, die während der Interaktion der Action mit dem Model auftauchen könnten. Ungeachtet der Komplexität müssen Actions letzten Endes einen String zurückgeben, der auf eine der Result-Komponenten gemappt wird, die für das Rendern der View für diese Action verfügbar ist.

Sie sollten nun verstanden haben, was eine Action macht, doch bevor wir selbst eine schreiben, müssen wir die Pakete erstellen, die sie enthalten sollen. Im nächsten Abschnitt werden wir sehen, wie man Actions in Pakete organisiert, und werfen einen ersten Blick auf die Anwendung Struts 2 Portfolio, die Hauptanwendung dieses Buches.

3.2 Verpacken der Actions

Sie können Ihre Actionkomponenten mit XML oder Java-Annotationen deklarieren, doch wenn das Framework die Architektur der Anwendung erstellt, organisiert es Actions und andere Komponenten in logische Container, die sogenannten Pakete (*packages*). Diese sind bei Struts 2 genau so wie bei Java. Man kann mit ihnen Actions (basierend auf gemeinsamen Funktionen oder Aufgabenbereichen) gruppieren. Viele wichtige operationale Attribute wie der URL-Namensraum, auf den Actions gemappt werden, definiert man auf

Paketebene. Pakete haben außerdem einen Vererbungsmechanismus, über den Sie u.a. auch die bereits vom Framework definierten Komponenten vererben können. In diesem Abschnitt gehen wir auf die Details der Struts-2-Pakete ein und untersuchen den Verpackungsvorgang bei Struts 2 Portfolio.

Zuerst gibt es einen kleinen Überblick über Zweck und Funktionalität der Beispielanwendung, um die Aufteilung der Actions in separate Pakete vorzubereiten.

3.2.1 Das Struts 2 Portfolio

In diesem Buch werden wir eine exemplarische Anwendung namens Struts 2 Portfolio entwickeln und untersuchen. Künstler können damit ein Online-Portfolio ihrer Arbeiten schaffen. Das Portfolio ist im Prinzip eine Bildergalerie. Ein Künstler muss sich zuerst im System registrieren, um ein Portfolio zu erstellen. Das ist kostenlos, aber einige harmlose persönliche Informationen werden erfasst. Sobald der Künstler ein Konto besitzt, kann er oder sie sich in den sicheren Bereich der Anwendung einloggen, um solch sensible Geschäfte wie die Erstellung eines neuen Portfolios und auch das Hinzufügen und Löschen von Bildern durchzuführen. Die andere Seite des Portfolios ist die öffentliche Seite. Ein Besucher der öffentlichen Site kann die Bilder der Portfolios betrachten. Dieser öffentliche Teil des Portfolios wird nicht durch Sicherheitsvorkehrungen geschützt.

Diese Anwendung ist zwar ganz simpel, doch komplex genug, um damit die Kernkonzepte von Struts 2 zu demonstrieren, zu denen auch Strategien zum Zusammenstellen von Paketen gehören. Eine kurze Analyse unserer Anforderungen besagt, dass wir in unserer Webanwendung zwei verschiedene Bereiche haben. Einige Funktionen kann jeder nutzen (z.B. die Registrierung von Konten oder das Betrachten von Portfolios), und andere sind abgesichert, vor allem die der Kontoverwaltung. Am Ende werden diese Funktionalitäten mit Actions implementiert, und Sie können darauf wetten, dass die sicheren Actions andere Anforderungen haben als die nicht sicheren. Schauen wir uns an, wie wir mit Struts-2-Paketen die Actions in sichere und unsichere Pakete gruppieren können.

3.2.2 Die Organisation der Pakete

Sie können frei entscheiden, nach welcher Strategie Sie den Paketraum der Anwendung ordnen. Wir werden die Pakete von Struts 2 Portfolio basierend auf ähnlichen Funktionalitäten organisieren, denn das ist eine übliche Strategie. Hier soll man sehen, wie die Pakete deklariert und konfiguriert werden, um eine bestimmte organisatorische Struktur zu bekommen. Wie bereits erwähnt, können Sie die Architekturkomponenten der Anwendung mit XML-Dateien oder Java-Annotationen deklarieren, die sich in den Dateien der Action-Klassen befinden. Wir haben auch erwähnt, dass wir für den Beispielcode XML-Dateien nehmen werden. Das behalten wir im Hinterkopf und schauen uns nun die Datei chapterThree.xml an, die die Komponenten für den ersten Teil von Struts 2 Portfolio deklariert. Sie finden diese Datei in */WEB-INF/classes/manning/chapterThree*. Diese XML-Datei enthält Deklarationen zweier Pakete: eine für öffentliche und eine für sichere Actions. Listing 3.2 zeigt die Deklaration des sicheren Pakets.

Listing 3.2 Deklaration eines Pakets

```
<package name="chapterThreeSecure" namespace="/chapterThree/secure"
      extends="struts-default">
    <action name="AdminPortfolio" >
        <result>/chapterThree/AdminPortfolio.jsp</result>
    </action>
    <action name="AddImage" >
        <result>/chapterThree/ImageAdded.jsp</result>
    </action>
    <action name="RemoveImage" >
        <result>/chapterThree/ImageRemoved.jsp</result>
    </action>
</package>
```

Das in Listing 3.2 deklarierte Paket enthält alle sicheren Actions der Anwendung. Bei diesen Actions ist eine Benutzerauthentifizierung erforderlich. Ein kurzer Blick auf die Namen dieser Actions sollte ausreichen, um eine Vorstellung ihrer Funktionsaufgaben zu bekommen. Naheliegenderweise sollten jene Actions, die Bilder in einem Portfolio löschen oder ergänzen können, durch eine Authentifizierung gesichert sein. Wir wollen gewährleisten, dass dem Benutzer, der Bilder aus einem Portfolio entfernt, diese auch tatsächlich gehören. Wenn sie gemeinsam gruppiert werden, können wir die Deklaration von Komponenten zusammen nutzen, die für den Authentifizierungsmechanismus nützlich sein können. Zusätzlich haben wir uns für die Vergabe eines speziellen URL-Namensraums für diese sicheren Actions entschlossen. Die Benutzer sollen aus dem URL entnehmen können, dass sie einen sicheren Bereich der Website betreten haben.

Nun schauen wir uns die Paketdeklaration selbst an. Sie können bei einem `package`-Element nur vier Attribute setzen: `name`, `namespace`, `extends` und `abstract`. Tabelle 3.1 fasst diese Attribute zusammen.

Tabelle 3.1 Die Attribute des `package`-Elements von Struts 2

Attribute	Beschreibung
`name` (erforderlich)	Name des Pakets
`namespace`	Namensraum für alle Actions im Paket
`extends`	Übergeordnetes Paket, von dem geerbt werden soll
`abstract`	Bei `true` wird dieses Paket nur verwendet, um vererbbare Komponenten (keine Actions) zu definieren.

Es mag eine Herausforderung sein, die Strategie zu wählen, nach der Sie die Actions in Pakete aufteilen, doch die Deklaration ist einfach. Das einzig erforderliche Attribut ist `name`. Das ist einfach ein logischer Name, über den Sie das Paket referenzieren können. In Listing 3.2 haben wir unser Paket `chapterThreeSecure` benannt. Dies weist wie alle guten Namen auf den Zweck dieses Pakets hin: Es enthält die sicheren Actions der Kapitel-3-Version der Beispielanwendung Struts 2 Portfolio.

Als Nächstes setzen wir das Attribut `namespace` auf `/chapterThree/secure`. Wie bereits gesehen, wird das Attribut `namespace` zur Generierung des URL-Namensraums verwendet, auf den die Actions dieser Pakete gemappt werden. Im Falle der `AddImage`-Action aus Listing 3.2 wird der URL wie folgt erstellt:

http://localhost:8080/manningHelloWorld/chapterThree/ secure/AddImage.action

Wenn bei diesem URL eine Anfrage eintrifft, sucht das Framework im Namensraum `/chapterThree/secure` nach einer Action namens `AddImage`. Denken Sie daran, dass Sie mehreren Paketen den gleichen Namensraum geben können. Falls Sie das machen, werden die Actions aus den beiden Paketen auf den gleichen Namensraum gemappt. Das ist nicht unbedingt ein Problem. Sie könnten sich dafür entscheiden, Ihre Actions aus bestimmten funktionellen Gründen, für die kein bestimmter Namensraum erforderlich ist, in separate Pakete zu legen. In unserem Fall beschließen wir, dass der Benutzer eine Änderung des URL-Namensraums sehen soll, wenn er den sicheren Bereich der Anwendung betritt.

Anmerkung

Die Beispielanwendung Struts 2 Portfolio – Wir sollten kurz die Struktur dieser Beispielanwendung erläutern. Der gesamte Beispielcode, der in diesem Buch entwickelt wird, erscheint in einer einzigen WAR-Datei: Struts2InAction.war. Es ist anders gesagt eine einzige große Webanwendung. (Sie wissen ja noch, dass wir auch eine skelettale Neuverpackung des HelloWorld-Beispiels als Standalone-Webanwendung anbieten. Doch das ist nur als Bonus zu verstehen, damit Sie wissen, wie eine minimale Struts-2-Webanwendung aussieht.) Innerhalb der Anwendung gibt es sozusagen viele „Unteranwendungen“. Jedes Kapitel hat beispielsweise seine eigene Version von Struts 2 Portfolio. Wir haben die Art der Gruppierung in Pakete von Struts 2 verwendet, um diese Versionen voneinander zu isolieren. Sie haben alle ihren eigenen Namensraum und Struts-2-XML-Dateien. So können wir nicht nur verschiedene Versionen von Struts 2 Portfolio anbieten, die sich auf die speziellen Ziele eines jeden Kapitels beziehen (womit die Lernkurve weniger steil wird), sondern es dient auch der Demonstration, wie nützlich diese Pakete sind.

Wenn Sie das Attribut `namespace` nicht setzen, werden die Actions in den Default-Namensraum kommen. Dieser sitzt unterhalb aller anderen Namensräume und wartet darauf, Anfragen zu bearbeiten, die nicht zu einem expliziten Namensraum passen. Betrachten wir Folgendes:

http://localhost:8080/manningHelloWorld/chapterSeventy/secure/AddImage.action

Wenn unsere Beispielanwendung diese Anfrage bekommt, wird das Framework versuchen, den Namensraum `/chapterSeventy/secure` zu lokalisieren. Weil der nicht existiert, kann die Action `AddImage` nicht gefunden werden. Als letzte Zuflucht wird das Framework im Default-Namensraum nach dieser Action suchen. Wenn es dort fündig wird, kann der URL aufgelöst und die Anfrage erledigt werden. Beachten Sie, dass der Default-Namensraum im Grunde ein leerer String `""` ist. Sie können auch einen Root-Namensraum wie `"/"` definieren. Der Root-Namensraum wird wie alle anderen expliziten Namensräume behandelt und muss passen. Es ist wichtig, zwischen dem leeren Default-Namensraum zu unterscheiden, der alle Anfragemuster empfangen kann, solange der Actionname passt, und dem Root-Namensraum, der ein echter Namensraum ist, zu dem die Anfrage passen muss.

Das nächste, in Listing 3.2 zu setzende Attribut ist `extends`. Dieses wichtige Attribut benennt ein weiteres Paket, dessen Komponenten vom aktuellen Paket vererbt werden sollen. Dies ist ähnlich wie das Schlüsselwort `extends` in Java. Wenn Sie sich das benannte Paket

so vorstellen, dass es die Superklasse Ihres aktuellen Pakets ist, verstehen Sie, dass das aktuelle Paket alle Mitglieder des Superklassenpakets erben wird. Obendrein kann das aktuelle Paket Mitglieder aus dem Superklassenpaket überschreiben. Die Paketvererbung spielt eine besonders wichtige Rolle bei der Verwendung der intelligenten Defaults, die wir so nachdrücklich anpreisen. Die meisten intelligenten Defaults werden in einem eingebauten Paket namens `struts-default` definiert. Sie können die in diesem Paket definierten Komponenten erben und verwenden, indem Sie die Pakete `struts-default` erweitern.

Doch was erben Sie da genau?

3.2.3 Die Arbeit mit den Komponenten des struts-default-Pakets

Die intelligenten Default-Komponenten des `struts-default`-Pakets lassen sich leicht verwenden. Sie brauchen es nur zu erweitern, wenn Sie eigene Pakete erstellen. Unser `chapterThreeSecure`-Paket macht genau das. Per Definition sollte es bei einem intelligenten Default nicht erforderlich sein, dass ein Entwickler selbst noch Hand anlegt. Tatsächlich kommen viele der Komponenten automatisch ins Spiel, sobald Sie dieses Paket erweitern. Ein gutes Beispiel ist der Default-Interceptor-Stack, den wir bereits in der HelloWorld-Anwendung benutzt haben. Wie haben wir den verwendet?

Definition

Das `struts-default`-Paket (definiert in der Datei struts-default.xml des Systems) deklariert eine umfangreiche Gruppe häufig verwendeter Struts-2-Komponenten, die von kompletten Interceptor-Stacks bis zu allen möglichen Result-Typen reichen.

Hier lüften wir das Geheimnis: Die meisten der Interceptors, die Sie jemals brauchen werden, befinden sich im `struts-default`-Paket (deklariert in der struts-default.xml-Datei). Sie können sich diese wichtige Datei als eigenes Artefakt der deklarativen Architektur des Frameworks vorstellen. Das damit definierte `struts-default`-Paket enthält häufig verwendete Architekturkomponenten, die alle Entwickler nutzen können, indem sie einfach ihre eigenen Pakete davon ableiten. Wenn Sie die ganze Datei sehen wollen, finden Sie sie in der Distribution auf der Root-Ebene der JAR-Datei struts2-core.jar. Listing 3.3 zeigt die Elemente aus dieser Datei, die den von den meisten Anwendungen verwendeten Default-Interceptor-Stack deklarieren.

Listing 3.3 Das Paket `struts-default` deklariert viele häufig verwendete Komponenten.

```
<package name="struts-default">   ❶ Paketelement
...
    <interceptor-stack name="defaultStack">   ❷ Deklariert defaultStack Interceptor-Stack
        <interceptor-ref name="exception"/>
        <interceptor-ref name="alias"/>
        <interceptor-ref name="servlet-config"/>
        <interceptor-ref name="prepare"/>
        <interceptor-ref name="i18n"/>
        <interceptor-ref name="chain"/>
        <interceptor-ref name="debugging"/>
        <interceptor-ref name="profiling"/>
```

```
            <interceptor-ref name="scoped-model-driven"/>
            <interceptor-ref name="model-driven"/>
            <interceptor-ref name="fileUpload"/>
            <interceptor-ref name="checkbox"/>
            <interceptor-ref name="static-params"/>
            <interceptor-ref name="params">
                <param name="excludeParams">dojo\..*</param>
            </interceptor-ref>
            <interceptor-ref name="conversionError"/>
            <interceptor-ref name="validation">
                <param name="excludeMethods">input,back,cancel,browse</param>
            </interceptor-ref>
            <interceptor-ref name="workflow">
                <param name="excludeMethods">input,back,cancel,browse</param>
            </interceptor-ref>
        </interceptor-stack>
...
    <default-interceptor-ref name="defaultStack"/>   ❸ ║ Setzt defaultStack als Default-Stack

...
</package>
```

Innerhalb des `package`-Elements ❶ wird ein Interceptor-Stack deklariert ❷. Er bekommt den sinnigen Namen `defaultStack`. Gegen Ende des Listings sehen Sie, dass dieser Stack so wie alle anderen von `struts-default` erweiterten Pakete als Default-Interceptor-Stack ❸ für dieses Paket deklariert wird. Im weiteren Verlauf dieses Buches werden wir alle Interceptors aus dem Default-Stack vorstellen. (Auch das Schreiben eigener Interceptors wird Thema sein, und zwar in Kapitel 4.) Für jetzt werden wir einfach auf einen verweisen, den Sie zu schätzen wissen sollten. Beachten Sie den Interceptor namens `params`. Falls Sie sich darüber gewundert haben, dass Daten aus der Anfrage geheimnisvollerweise automatisch zur Action transferiert werden, bekommen Sie hier die Antwort. Dieser wichtige Interceptor `params` war derjenige, der Daten aus den Anfrageparametern in die JavaBeans-Properties unserer Action verschoben hat. Das ist kein Zaubertrick, sondern wird mit guten, alten Codezeilen erledigt. Für die Neugierigen: Diese Codezeilen bieten einen guten Einblick in die inneren Vorgänge von Struts 2; schauen Sie sich den Quellcode für `com.opensymphony.xwork2.interceptor.ParametersInterceptor` an, wenn Sie es sich nicht denken können.

Wie Sie aus dem Interceptor `params` ersehen, wird ein Großteil der Kernfunktionalität des Frameworks über Interceptors implementiert. Es ist nicht *absolut* notwendig, das `struts-default`-Paket zu erweitern, wenn Sie eigene Pakete erstellen. Wenn man diese Vererbung aber weglässt, führt das letzten Endes dazu, dass der Kern des Frameworks verworfen wird. Schauen Sie sich die Paketvererbung in Abbildung 3.1 an.

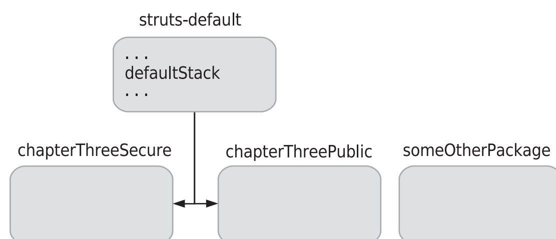

Abbildung 3.1
Ein Großteil der Frameworkfunktionalität bekommen Sie durch Erweiterung des `struts-default`-Pakets.

In diesem Paket sehen wir, dass das insgesamt wichtige `defaultStack` für unsere Pakete `chapterThreeSecure` und `chapterThreePublic` verfügbar gemacht wird. Abbildung 3.1 zeigt auch `someOtherPackage`, das `struts-default` nicht erweitert. Dieses Paket fängt bei Null an. Ohne die wichtigen, im `defaultStack` des `struts-default`-Pakets definierten Interceptors gehen die meisten Features des Frameworks verloren. Dazu gehört u.a. auch der bereits angesprochene automatische Datentransfer. Ohne diese Features ist das Framework nackt und bloß, um es mal so zu sagen. Sie können sich immer noch die Zeit nehmen, um alle Komponenten, die vom Framework in der Datei struts-default.xml deklariert werden, erneut zu deklarieren, doch das wäre lästig und witzlos. Wenn Sie nicht gerade überzeugende Gründe dagegen haben, sollten Sie immer das `struts-default`-Paket erweitern. Und (wie Sie auch später sehen werden, wenn es in Kapitel 4 ausführlich um Interceptors gehen wird) das `struts-default`-Paket deklariert seine Komponenten so, dass sie sowohl flexibel als auch wiederverwendbar werden. Alles in allem sollten Sie es sich zweimal überlegen, auf die Erweiterung von `struts-default` zu verzichten.

Nun haben wir eine recht gute Vorstellung vom Mechanismus der Organisation unserer Anwendung in Pakete und auch, wie man Pakete wie das eingebaute `struts-default`-Paket erweitert. Wir haben sogar `chapterThreeSecure` gesehen, eines der Pakete aus der Beispielanwendung Struts 2 Portfolio. Am Ende werden Pakete zu etwas, über das Sie nicht viel nachdenken. Tatsächlich werden Sie damit sogar recht wenig zu tun haben, wenn die Paketstruktur Ihrer Anwendung an Ort und Stelle ist. Nun können wir endlich Actions für Struts 2 Portfolio erstellen.

3.3 Implementierung von Actions

Jetzt machen wir uns an die Entwicklung einiger Actions. In diesem Abschnitt stellen wir die Grundlagen vor, wie Actions für Struts-2-Anwendungen geschrieben werden. Als Beispiele zeigen wir Ihnen einige der Actions aus der Kapitel-3-Version von Struts 2 Portfolio. Wir stellen diese Actions zwar gründlich vor, doch man kann noch mehr erfahren als nur das in diesem Buch Aufgeführte, wenn man sich die Beispielanwendung vornimmt. Schauen Sie einfach in den Quellcode, er ist gut kommentiert!

Die Implementierung von Struts-2-Actions ist einfach. Weiter oben in diesem Kapitel haben wir gesehen, dass der Kontrakt zwischen Framework und den Klassen, die hinter den Actions stehen, eine sehr hohe Flexibilität erlaubt. Im Grund kann jede Klasse eine Action sein, wenn das gewünscht wird. Sie muss nur eine Einstiegsmethode bieten, die das Framework aufrufen kann, wenn die Action ausgeführt wird. Schauen wir uns an, wie die Implementierung der Actions durch das Framework sogar noch leichter wird. Als Fallstudie werden wir unseren Blick auf Struts 2 Portfolio fortführen, indem wir einige seiner Action-Klassen auseinandernehmen.

Anmerkung

Struts-2-Actions müssen das `Action`-Interface nicht implementieren. Jedes Objekt kann den Kontrakt mit dem Framework formlos befolgen, indem es einfache eine `execute()`-Methode implementiert, die einen Control-String zurückgibt.

3.3.1 Das optionale Action-Interface

Obwohl das Framework Ihren Actions in formaler Hinsicht wenig aufzwingt, bietet es doch ein optional zu implementierendes Interface. Die Implementierung des `Action`-Interfaces ist nicht aufwendig und bietet einige praktische Vorteile. Schauen wir uns an, warum die meisten Entwickler das `Action`-Interface implementieren, wenn sie ihre Actions entwickeln, obwohl sie das nicht müssen.

Warnung

Bei Struts 2 bekommen Entwickler einen schnellen Entwicklungspfad, der auf intelligenten Defaults aufbaut, und ein extrem hohes Maß an Flexibilität, um auch die obskursten Anwendungsfälle elegant zu lösen. Wenn man das Framework kennenlernt, kann es helfen, sich auf die unkomplizierten Lösungen zu konzentrieren, die von den intelligenten Defaults unterstützt werden. Wenn man erst einmal mit dem normalen Weg vertraut ist, die Dinge anzugehen, wird die Flexibilität des Frameworks selbstverständlich und leistungsfähig. Ohne ein gutes Verständnis für das korrekte Verhalten kann die Flexibilität des Frameworks einem zugegebenermaßen Kopfzerbrechen bei der Frage bereiten, welchem Pfad man folgen soll.

Die meisten Actions werden das `com.opensymphony.xwork2.Action`-Interface implementieren. Es definiert nur eine Methode:

```
String execute() throws Exception
```

Weil das Framework keine Typanforderungen stellt, könnten Sie einfach die Methode in die Klasse schreiben, ohne dass Ihre Klasse dieses Interface implementieren muss. Das ist in Ordnung so, doch das `Action`-Interface bietet ebenfalls ein paar nützliche `String`-Konstanten, die als Rückgabewerte für die Auswahl des passenden Results verwendet werden können. Die vom `Action`-Interface definierten Konstanten sind

```
public static final String  ERROR   "error"
public static final String  INPUT   "input"
public static final String  LOGIN   "login"
public static final String  NONE    "none"
public static final String  SUCCESS "success"
```

Diese Konstanten kann man sehr gut als Werte für den Control-String nutzen, den Ihre `execute()`-Methode zurückgibt. Der echte Vorteil ist, dass diese Konstanten auch intern vom Framework verwendet werden können. Das bedeutet, dass man mit diesen vordefinierten Control-Strings auch weitere intelligente Default-Verhaltensweisen anzapfen kann.

Als Beispiel nehmen wir Pass-Through-Actions. Erinnern Sie sich noch an die Pass-Through-Action, mit der wir in der HelloWorld-Anwendung gearbeitet haben? Wir sagten, dass es eine Best Practice sei, sogar die einfachsten Anfragen durch Actions zu routen. In der HelloWorld-Anwendung haben wir mit einer dieser leeren Actions auf die JSP-Seite

zugegriffen, die das Formular zum Eintragen des Benutzernamens präsentierte. Dies ist die Deklaration dieser Action:

```
<action name="Name">
   <result>/chapterOne/NameCollector.jsp</result>
</action>
```

Die `Name`-Action gibt für die Actionimplementierung keine Klasse an, weil es nichts zu tun gibt. Wir wollen einfach nur zur JSP-Seite. Praktischerweise bieten die intelligenten Defaults von Struts 2 eine Default-Actionimplementierung, die wir erben können, wenn keine spezifiziert wurde. Diese Default-Action hat eine leere `execute()`-Methode, die nichts anderes macht, als automatisch die `SUCCESS`-Konstante des `Action`-Interface als Control-String zurückzugeben. Das Framework muss diesen String verwenden, um ein Result zu wählen. Glücklicherweise (oder nicht so glücklich) ist das Default-`name`-Attribut für das Result-Element auch die `SUCCESS`-Konstante. Weil unser einziges Result darauf verzichtet, den eigenen Namen zu definieren, erbt es diesen Standard und wird automatisch von unserer Action ausgewählt. Dies ist das allgemeine Muster, nach dem viele der intelligenten Defaults operieren.

Doch Moment mal – wir müssen das `Action`-Interface nicht selbst implementieren, weil das Framework eine akzeptable Implementierung bietet. Als Nächstes schauen wir uns eine Convenience-Klasse an, die dieses und andere hilfreiche Interfaces implementiert, mit denen Sie bereits im Framework enthaltene Features noch weitgehender nutzen können.

3.3.2 Die ActionSupport-Klasse

In diesem Abschnitt stellen wir die `ActionSupport`-Klasse vor. Das ist eine Convenience-Klasse, die Default-Implementierungen des `Action`-Interface und mehrere andere nützliche Interfaces enthält. So bekommen wir solch schöne Sachen wie Datenvalidierung und die Lokalisierung von Fehlermeldungen. Diese Convenience-Klasse ist ein perfektes Beispiel für das korrekte Verhalten von Struts 2, von dem wir gerade gesprochen haben. Sie werden vom Framework nicht dazu gezwungen, doch wenn man das Framework gerade neu lernt, ist das eine gute Idee. Tatsächlich ist es eigentlich immer eine gute Idee, es zu verwenden, außer es sprechen gute Gründe dagegen.

Der Tradition von „Support"-Klassen folgend bietet `ActionSupport` Default-Implementierungen für mehrere wichtige Interfaces. Wenn Ihre Actions diese Klasse erweitern, haben sie gleich automatisch den Vorteil, diese Implementierungen nutzen zu können. Allein dadurch lohnt es sich schon, diese Klasse verstanden zu haben. Jedoch bieten die von dieser Klasse gebotenen Implementierungen auch eine ausgezeichnete Fallstudie darüber, wie man eine Action zur Kooperation mit Interceptors bringt, um leistungsfähige und wiederverwendbare Lösungen für häufige Aufgaben zu bekommen. In diesem Fall bekommt man Validierungs- und Textlokalisierungsdienste über eine Kombination aus Interceptors und Interfaces. Die Interceptors steuern die Ausführung der Dienste, während die Actions Interfaces mit von den Interceptors aufgerufenen Methoden implementieren. Dieses wichtige Muster wird klarer, wenn wir uns in die Details von `ActionSupport` einarbeiten, indem wir dessen Verwendung beim Struts 2 Portfolio untersuchen (dazu gleich mehr).

3.3.2.1 Einfache Validierung

Während Struts 2 ein umfassend und weitgehend konfigurierbares Validierungsframework bietet, das wir in Kapitel 10 eingehend besprechen werden, enthält `ActionSupport` eine schnelle Form der Basisvalidierung, die in vielen Fällen gute Dienste leistet. Überdies ist es ein ausgezeichnetes Fallbeispiel, wie eine querschnittliche Aufgabe wie Validierung durch die Nutzung von Interceptors und Interfaces aus der Ausführungslogik der Action herausgenommen werden kann. Das übliche Muster ist, dass der Interceptor bei der Steuerung der Ausführung einer bestimmten Aufgabe mit der Action zusammenarbeitet, indem er Methoden aufruft, die er zur Verfügung stellt. Gewöhnlich sind diese Methoden Teil eines speziellen Interfaces, das von dieser Action implementiert wird. In unserem Fall implementiert `ActionSupport` zwei Interfaces, die mit einem der Interceptors vom Default-Stack `DefaultWorkflowInterceptor` zusammenarbeiten, um eine einfache Validierung zu ermöglichen. Wenn Ihr Paket `struts-default` erweitert (und somit den Default-Interceptor-Stack erbt) und Ihre Action `ActionSupport` erweitert (und dadurch die Implementierung der beiden notwendigen Interfaces erbt), dann haben Sie bereits alles Nötige für eine saubere Validierung der Daten.

Um den Hintergrund all dieser eingebauten Funktionalität zu verdeutlichen, zeigen wir Ihnen, wo diese Default-Interceptors definiert werden, und wie Sie sichergehen, dass sie auch vererbt werden. Listing 3.4 zeigt die Deklaration des `workflow`-Interceptors, so wie er in der Datei struts-default.xml vorkommt.

Listing 3.4 Deklaration von `DefaultWorkflowInterceptor` aus struts-default.xml

```
...
<interceptor name="workflow"  ❶
class="com.opensymphony.xwork2.interceptor.DefaultWorkflowInterceptor"/>
...
 <interceptor-stack name="defaultStack">  ❷
   ...
   <interceptor-ref name="params"/>  ❸
   ...
   <interceptor-ref name="workflow">
      <param name="excludeMethods">input,back,cancel,browse</param>   ❹
   </interceptor-ref>
   . . .
<interceptor-stack name="defaultStack">
...
```

In Listing 3.4 sehen wir zuerst das Deklarationselement für den `workflow`-Interceptor ❶, der einen Namen und eine Implementierungsklasse spezifiziert. Beachten Sie, dass der Name Workflow-Interceptor daher stammt, dass er bei einem etwaigen Validierungsfehler den Workflow der Anfrage wieder zurück zur Eingabeseite lenkt. Als Nächstes sehen wir die Deklaration des Default-Interceptor-Stacks ❷. Wir haben in diesem Listing nicht alle Interceptors aufgenommen. Stattdessen konzentrieren wir uns auf die Interceptors, die am Validierungsprozess beteiligt sind. Beachten Sie, dass der `params`-Interceptor ❸ vor dem `workflow`-Interceptor dran ist ❹. Der `params`-Interceptor wird die Anfragedaten auf unser Actionobjekt verschieben. Dann hilft uns der `workflow`-Interceptor dabei, diese Daten zu validieren, bevor unser Model sie akzeptiert. Er muss feuern, nachdem der `params`-Inter-

ceptor die Daten auf das Actionobjekt verschieben konnte. Wie bei den meisten Interceptors ist die Reihenfolge wichtig.

Nun schauen wir uns an, wie diese Validierung tatsächlich funktioniert. Wir werden zur Veranschaulichung eine unserer Actions aus der Struts 2 Portfolio-Version dieses Kapitels nehmen: die `Register`-Action. Wie beim `params`-Interceptor will der `workflow`-Interceptor die Logik einer querschnittlichen Aufgabe (in diesem Fall der Validierung) aus der Ausführungslogik der Action entfernen. Wenn der `workflow`-Interceptor feuert, wird er bei der Action zuerst nach einer `validate()`-Methode suchen, die er aufrufen kann. Sie platzieren die Validierungslogik in `validate()`. Diese Methode wird anhand des Interface `com.opensymphony.xwork2.Validateable` zur Verfügung gestellt. Technisch ausgedrückt implementiert `ActionSupport` die `validate()`-Methode, doch wir müssen deren leere Implementierung mit unserer eigenen spezifischen Validierungslogik überschreiben.

Wie bereits erwähnt demonstrieren wir die Konzepte und Strategien dieses Buches durch die Entwicklung von Struts 2 Portfolio. In diesem Abschnitt untersuchen wir die `Register`-Action aus der Version von Kapitel 3. Schauen wir uns den gesamten Quellcode dieser Action-Klasse an. Listing 3.5 zeigt den ganzen Quellcode von `Register` aus dem Verzeichnis der Beispielanwendung in *manning/chapterThree/Register.java*.

Listing 3.5 Die Action `Register` enthält in der `validate()`-Methode die Validierungslogik.

```
public class Register extends ActionSupport {   ❶

    public String execute(){

        User user = new User();
        user.setPassword( getPassword() );
        user.setPortfolioName( getPortfolioName() );
        user.setUsername( getUsername() );
                                                        ❷
        getPortfolioService().createAccount( user );
        return SUCCESS;
    }

    private String username;
    private String password;
    private String portfolioName;
    public String getPortfolioName() {
        return portfolioName;
    }
    public void setPortfolioName(String portfolioName)    {
        this.portfolioName = portfolioName;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;                             ❸
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
```

```
public void validate(){
    PortfolioService ps = getPortfolioService();❺
    if ( getPassword().length() == 0 ){
        addFieldError( "password", "Password is required.") );❼   ❻
    }
    if ( getUsername().length() == 0 ){
        addFieldError( "username", "Username is required." );          ❹
    }
    if ( getPortfolioName().length() == 0  ){
        addFieldError( "portfolioName", "Portfolio name is required.");
    }
    if ( ps.userExists(getUsername() ) ){
        addFieldError("username", "This user already exists."); ❽
    }
}

public PortfolioService getPortfolioService( ) {
    return new PortfolioService();
}
```

Wir zeigen hier zwar den gesamten Quellcode, doch Sie sollten im Hinterkopf behalten, dass wir uns gewissermaßen auf den über `ActionSupport` möglichen Validierungsmechanismus konzentrieren. Deswegen überfliegen wir schnell die Teile, die nicht zur Validierung gehören, und konzentrieren uns dann auf die Validierung. Wir kommen auf den nächsten Seiten auf die restlichen Details zurück. Zuerst beachten Sie bitte, dass unsere Action tatsächlich `ActionSupport` erweitert ❶. Achten Sie auch darauf, dass wir eine `execute()`-Methode anbieten ❷, in der die Businesslogik enthalten ist (die in diesem Fall den Benutzer registriert). Danach sehen wir eine Gruppe JavaBeans-Properties ❸. Dies sind häufig vorkommende Features von Actions; sie empfangen die Daten aus dem automatischen Transfer des Frameworks und transportieren sie dann durch die Verarbeitung des Frameworks.

Doch nun konzentrieren wir uns auf die Untersuchung des einfachen Validierungsmechanismus, den `ActionSupport` bietet. Unsere Action enthält ja eine `validate()`-Methode ❹, in der die gesamte Logik der Prüfung auf valide, von unseren JavaBeans-Properties empfangene Daten enthalten ist. Somit konzentriert sich die `execute()`-Methode dieser Action weiter auf die Businesslogik. Unsere Validierungslogik hier ist simpel: Wir überprüfen bei allen drei Feldern, dass sie nicht leer sind, indem wir die Länge aller `String`-Properties checken ❻. Wenn bestimmte Daten nicht validiert werden, erstellen und speichern wir einen Fehler ❼ über Methoden aus der `ActionSupport`-Superklasse, z.B. `addFieldError()`.

Wir prüfen auch, ob der Benutzer nicht bereits im System vorhanden ist ❽. An diesem Punkt des Buches arbeitet Struts 2 Portfolio mit einer einfachen Kapselung der Businesslogik und der Datenpersistenz. Das Objekt `PortfolioService` kann in dieser Phase unsere einfachen Businessanforderungen umsetzen. Falls es Sie interessiert: Es enthält in seinen einfachen Methoden alle Businessregeln und persistiert Daten nur im Speicher. Auch unsere aktuellen Verwaltungstechniken sind recht grob; die Action instanziiert nur ein `PortfolioService`-Objekt ❺, wenn es eines braucht. Im weiteren Verlauf des Buches lernen wir, wie man anspruchsvollere Technologien integriert, um solche wichtigen Res-

sourcen zu verwalten. Doch momentan bleibt so unsere Untersuchung der Actionkomponente klarer.

Was passiert, wenn die Validierung fehlschlägt? Wenn irgendeines der Felder leer ist oder der Benutzername im System bereits vorhanden ist, rufen wir eine Methode auf, die eine Fehlermeldung hinzufügt. Nachdem alle Validierungslogik ausgeführt wurde, kehrt die Steuerung zum `workflow`-Interceptor zurück. Beachten Sie, dass es bei der `validate()`-Methode keinen Rückgabewert gibt. Das Geheimnis liegt – wie wir gleich sehen werden – in den Fehlermeldungen, die unsere Validierung generiert.

Auch wenn die Steuerung an den `workflow`-Interceptor zurückgekehrt ist, ist die Verarbeitung noch nicht abgeschlossen. Was macht er jetzt? Nun kann er sich den Namen „Workflow“ verdienen: Nach Aufruf der `validate()`-Methode, damit die Validierungslogik der Action ausgeführt werden kann, prüft der `workflow`-Interceptor, ob diese irgendwelche Fehlermeldungen generiert hat. Falls es welche gibt, wird der `workflow`-Interceptor den Workflow der Anfrage verändern. Er wird sofort die Verarbeitung der Anfrage abbrechen und den Benutzer zurück zum Eingabeformular bringen, wo im Formular die entsprechenden Fehlermeldungen erscheinen. Probieren Sie es aus! Starten Sie die Anwendung und öffnen Sie die Kapitel-3-Version von Struts 2 Portfolio unter

http://localhost:8080/Struts2InAction/chapterThree/PortfolioHomePage.action

Wählen Sie die Erstellung eines Kontos, füllen Sie das Formular aus, aber lassen Sie dabei ein paar Daten weg. Wir haben beispielsweise das Passwort ausgelassen. Wenn wir das Formular übermitteln, schlägt die Validierung fehl und leitet den Workflow wieder zurück zum Eingabeformular (siehe Abbildung 3.2.).

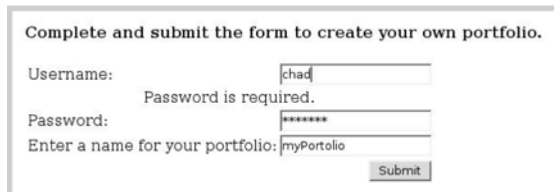

Abbildung 3.2
Der Default-Workflow-Interceptor leitet uns zum Eingabeformular zurück, auf dem Infos zu Validierungsfehlern an den entsprechenden Feldern dargestellt werden.

Zwei naheliegende Fragen bleiben offen. Wo waren diese Fehlermeldungen gespeichert und wie hat der Workflow-Interceptor gemerkt, ob irgendwelche Fehlermeldungen erstellt wurden? Das Interface `com.opensymphony.xwork2.ValidationAware` definiert Methoden für das Speichern und Auslesen von Fehlermeldungen. Eine Klasse, die dieses wichtige Interface implementiert, muss eine Sammlung von Fehlermeldungen für jedes Feld vorhalten, das validiert werden kann, sowie eine Sammlung allgemeiner Fehlermeldungen, die nur für die Action als Ganzes gelten. Zum Glück für uns gibt es all diese Methoden und Sammlungen, die sie unterstützen, bereits in der Klasse `ActionSupport`. Um damit zu arbeiten, rufen wir die folgenden Methoden auf:

```
addFieldError ( String fieldName, String errorMessage )
addActionError ( String errorMessage )
```

Um einen Feldfehler hinzuzufügen, müssen wir zusammen mit der Nachricht, die der Benutzer lesen soll, auch den Feldnamen übergeben (erfolgt in Listing 3.5). Einen für die ganze Action geltende Fehlermeldung einzufügen, ist sogar noch leichter, weil Sie nichts anderes als die Nachricht spezifizieren müssen.

Das Interface `ValidationAware` spezifiziert auch Methoden, um nach dem Vorkommen irgendwelcher Fehler zu suchen. Der `workflow`-Interceptor wird mit diesen dann bestimmen, ob er den Workflow zurück zur Eingabeseite umleiten soll. Wenn er auf einen Fehler trifft, wird er nach einem Result namens `input` suchen. Das folgende Snippet aus chapterThree.xml zeigt, dass die Action `Register` ein solches Result deklariert hat:

```
<action name="Register" class="manning.chapterThree.Register">
    <result>/chapterThree/RegistrationSuccess.jsp</result>
    <result name="input">/chapterThree/Registration.jsp</result>
</action>
```

In diesem Fall wird der `workflow`-Interceptor, falls er Fehler findet, automatisch an das Result weiterleiten, das auf die Seite Registration.jsp zeigt, weil sein Name `input` lautet. Und natürlich ist diese JSP-Seite unser Eingabeformular.

Nun wissen wir, wie Interceptors in der Ausführungslogik der Action aufräumen. Doch vielleicht sind einige von Ihnen noch nicht ganz überzeugt. Wenn Sie nun beanstanden: „Aber die Validierungsmethode ist doch immer noch beim Actionobjekt!“, dann haben Sie recht. Doch das ist kein Makel für die wichtigste Trennung der Aufgabenbereiche: Die Validierungslogik ist eindeutig von der Ausführungslogik der Action selbst getrennt. Somit bleibt unsere Action auf ihren ureigensten Unit of Work konzentriert: einen neuen Benutzer zu registrieren. Schauen Sie sich die prägnante Formulierung der Businesslogik bei der `execute()`-Methode dieser Aufgabe an:

```
public String execute(){
    User user = new User();
    user.setPassword( getPassword() );
    user.setPortfolioName( getPortfolioName() );
    user.setUsername( getUsername() );
    getPortfolioService().createAccount( user );
    return SUCCESS;
}
```

Wir erstellen einfach das Benutzerobjekt und das Konto. Kein Problem! Falls es dort eine Ausnahme gäbe, die über den Prozess der Kontoerstellung aus unserem Businessobjekt heraus erstellt würde, hätten wir ein wenig mehr Komplexität bei der Frage, welches Result wir darstellen sollten. Für unsere Zwecke gehen wir einfach von Erfolg aus.

Doch es geht nicht nur um sauber aussehenden Code. Ein subtilerer Punkt ist, dass der Steuerungsfluss des Validierungsprozesses auch von der Action separiert ist. Hierbei geht es nicht nur darum, die Validierungslogik aus der `execute()`-Methode herauszunehmen und in eine leichter lesbare Hilfsmethode zu fakturieren. Der Validierungsworkflow selbst wird vom Workflow der Action getrennt gehalten, weil die Validierungslogik vom `workflow`-Interceptor aufgerufen wird. Anders gesagt ist der `workflow`-Interceptor wirklich derjenige, der die Ausführung der Validierungslogik steuert. Die Interceptors feuern alle, bevor die Action selbst die Chance zur Ausführung bekommt. Diese Separation des

Steuerungsflusses ist es, die dem `workflow`-Interceptor erlaubt, die ganze Anfrageverarbeitung abzubrechen und wieder auf die Eingabeseite umzuleiten, ohne jemals die `execute()`-Methode der Action zu betreten. Dies ist genau die Art von Separation, die Interceptors bieten sollen.

Bevor es weitergeht, fragen sich vielleicht manche unter Ihnen, wie die Fehlermeldung es in das Registrierungsformular geschafft hat, als wir den Benutzer zu einem neuen Versuch zurückgeschickt haben. All das erledigen die Komponententags der Benutzerschnittstelle von Struts 2 für Sie. Darum wird es jetzt nicht gehen, weil wir bei der Action bleiben wollen, doch darüber erfahren Sie alles in Kapitel 7.

Nun fühlt es sich so langsam an, als würden wir wirklich etwas lernen. Wir können Actions schreiben, die Daten automatisch annehmen und validieren. Das ist cool, aber wir wollen uns nicht vom anstehenden Thema ablenken lassen. Bei der eigentlichen Lektion dieses Abschnitts geht es darum, wie Actions mit Interceptors zusammenarbeiten, um häufig vorkommende Aufgaben zu erledigen, ohne die zentrale Logik der Action zu verschmutzen. Wenn Sie dieses Teamwork von Action und Interceptor verstehen, wird der Rest des Buches für Sie kaum mehr als weitere Ausführungen dieses einen Themas sein. Natürlich warten in den verbleibenden Kapiteln auch noch eine Menge weiterer cooler Sachen auf Sie, doch dies ist des Pudels Kern, wie das Framework die Problemlösung an sich angeht.

Bevor es weitergeht, müssen wir uns noch ein anderes Problem anschauen, das `ActionSupport` für Sie löst: den lokalisierten Nachrichtentext.

3.3.2.2 Ressourcenbündel für Nachrichtentext

Bei der Validierungslogik unserer `Register`-Action haben wir eingestellt, dass bei Fehlermeldungen `String`-Literale verwendet werden sollen. Wenn Sie sich Listing 3.5 noch einmal anschauen, sehen Sie, dass wir das `String`-Literal `Username is required` der `addFieldError()`-Methode übergeben haben. Wenn man `String`-Literale auf diese Weise verwendet, beschwört das einen wohlbekannten Wartungsalptraum herauf. Überdies ist es praktisch unmöglich, die Sprachen für unterschiedliche Ländereinstellungen (*locales*) zu wechseln, ohne eine trennende Schicht zwischen dem Quellcode und den Nachrichten selbst einzufügen. Die wohletablierte Best Practice ist, diese Nachrichten in externen und leicht wartbaren Ressourcenbündeln zusammenzufassen, die üblicherweise mit einfachen Properties-Dateien implementiert werden. Bei `ActionSupport` ist genau dafür eine Funktionalität eingebaut.

`ActionSupport` implementiert zwei Interfaces, die zusammen diese lokalisierte Funktionalität für Textnachrichten bieten. Das erste Interface `com.opensymphony.xwork2.TextProvider` ermöglicht den Zugriff auf die Nachrichten selbst. Dieses Interface stellt einen flexiblen Satz Methoden bereit, über den Sie einen Nachrichtentext aus einem Ressourcenbündel auslesen können. `ActionSupport` implementiert diese Methoden, um die Nachrichten aus einer Properties-Dateiressource auszulesen. Egal welche Methode Sie verwenden, um Ihre Nachricht auszulesen, Sie beziehen sich über einen Schlüssel auf die Nach-

richten. Die TextProvider-Methoden werden die mit diesem Schlüssel verknüpften Nachrichten aus der Properties-Datei, die mit Ihrer Action-Klasse verknüpft ist, zurückgeben.

Man kann ganz leicht mit einem Properties-Dateiressourcenbündel arbeiten. Zuerst muss die Properties-Datei erstellt werden und bekommt einen Namen, der die Action-Klasse spiegelt, für die sie die Nachrichten enthält. Das folgende Code-Snippet zeigt die Inhalte aus der mit unserer Register-Action verknüpften Properties-Datei Register.properties:

```
user.exists=This user already exists.
username.required=Username is required.
password.required=Password is required.
portfolioName.required=Portfolio Name is required.
```

Falls Ihnen Properties-Dateien nicht vertraut sind: Das sind einfache Textdateien. Jede Zeile enthält einen Schlüssel und dessen Wert. Damit die ActionSupport-Implementierung des TextProvider-Interface diese Properties-Datei findet, müssen wir sie nur dem Java-Paket hinzufügen, das unsere Register-Klasse enthält. In diesem Fall finden Sie diese Datei in der Paketstruktur unter manning.chapterThree.

Wenn die Properties-Datei an Ort und Stelle ist, können wir eine der getText()-Methoden von TextProvider nutzen, um die Nachrichten auszulesen. Listing 3.6 zeigt die neue Version der Validierungslogik von Register-Action.

Listing 3.6 Mit ActionSupport die Fehlermeldung für die Validierung holen

```
public void validate(){
   PortfolioService ps = getPortfolioService();
   if ( getPassword().length() == 0 ){
      addFieldError( "password", getText("password.required") ); ❶
   }
   if ( getUsername().length() == 0 ){
      addFieldError( "username", getText("username.required") );
   }
   if ( getPortfolioName().length() == 0  ){
      addFieldError( "portfolioName", getText( "portfolioName.required" ));
   }
   if ( ps.userExists(getUsername() ) ){
      addFieldError("username", getText( "user.exists"));
   }
}
```

Wie Sie sehen können, entnehmen wir unseren Nachrichtentext statt aus String-Literalen aus der ActionSupport-Implementierung von TextProvider. Wir nutzen nun die getText()-Methode ❶, um Nachrichten basierend auf einem Schlüssel aus Properties-Dateien auszulesen. Durch diese Separationsschicht kann unser Nachrichtentext deutlich einfacher verwaltet werden. Für die Änderungen der Nachrichten muss nur die Properties-Datei verändert werden; die semantischen Schlüssel des Quellcodes brauchen nie geändert zu werden.

Mit ActionSupport kann auch der Text der Nachrichten für die Internationalisierung leicht lokalisiert werden. Das Interface com.opensymphony.xwork2.LocaleProvider stellt die Methode getLocale() bereit. ActionSupport implementiert dieses Interface, um die Standorteinstellungen des Benutzers anhand der vom Browser übermittelten Re-

gions- und Sprachoptionen auszulesen. Sie können auch Ihre eigene Version dieses Interfaces implementieren, um woanders nach den Standorteinstellung zu suchen, beispielsweise in der Datenbank. Doch wenn Ihnen die Browsereinstellungen reichen, brauchen Sie für eine minimale Internationalisierung nicht allzu viel anzustellen.

Sie lesen Ihre Nachrichtentexte immer noch genau so aus wie vorher. Auch wenn wir das nicht nutzen, prüft die `TextProvider`-Implementierung von `ActionSupport` die Standorteinstellungen jedes Mal, wenn sie einen Nachrichtentext für uns ausliest. Das macht sie per Aufruf der `getLocale()`-Methode des `ActionSupport`-Interfaces. Anhand der Standorteinstellung versucht `TextProvider` alias `ActionSupport`, eine Properties-Datei für diese Standorteinstellungen zu lokalisieren. Natürlich müssen Sie die Properties-Datei für die fragliche Standorteinstellung bereithalten oder es gibt als Standard nur Englisch. Doch es ist ganz einfach, für alle gewünschten Standorteinstellungen Properties-Dateien einzurichten. Struts 2 Portfolio ergänzen wir um eine spanische Properties-Datei. Der schwere Teil ist, dafür einen Übersetzer zu finden. Um das in Aktion zu erleben, stellen Sie die Sprache des Browsers auf Spanisch ein und übermitteln erneut das Registrierungsformular. Dabei lassen Sie eines der Felder leer, um die dann erscheinende Fehlermeldung zu sehen.

Wie bei der Validierung ist die über `ActionSupport` mögliche Internationalisierung relativ primitiv. Wenn es für Ihre Anwendung ausreicht, ist das ausgezeichnet. Wenn Sie mehr brauchen, werden Sie in Kapitel 11 erfahren, wie man mit dem Struts-2-Framework eine topaktuelle Internationalisierung hinbekommt.

Als Nächstes schauen wir uns einige alternative (manche würden es fortgeschrittene nennen) Methoden an, wie man unseren Datentransfer mit komplexen Objekten statt einfachen JavaBeans-Properties implementiert.

3.4 Daten auf Objekte transferieren

Bisher haben alle unsere Actions die Daten aus der Anfrage auf einfachen JavaBeans-Properties erhalten. Das ist zwar leistungsfähig und elegant, es geht aber noch besser. Anstatt jedes Datenfitzelchen einzeln anzunehmen und dann ein Objekt zu erstellen, auf dem diese Daten dann platziert werden können, können wir das komplexe Objekt selbst den Datentransfermechanismen der Plattform zur Verfügung stellen. Das spart nicht nur Zeit, weil man das Objekt, das die einzelnen Datenteile ansammelt, nicht mehr selbst erstellen und füllen muss. Auch Arbeit wird eingespart, weil wir dem Datentransfer direkt ein bereits existierendes Domänenobjekt zur Verfügung stellen können. Zwar gibt es einige Dinge zu beachten, doch mit der Verwendung dieser komplexen Objekte als direkte Datentransferobjekte bekommt der Entwickler eine leistungsfähige Option an die Hand.

Anmerkung

Aus Sicht der Migration von Struts 1 zu Struts 2 – Falls Sie schon danach Sehnsucht hatten, müssen wir den Wegfall der vertrauten `ActionForm` von Struts 1 verkünden. `ActionSupport` spielte für das Struts-1-Framework bei der Datenvalidierung und der Typkonvertie-

rung eine wichtige Rolle, doch die Kosten waren auch sehr hoch. Für jedes Domänenobjekt mussten Sie normalerweise eine entsprechende Formular-Bean erstellen. Um das Ganze noch schlimmer zu machen, bekamen Sie dann den Auftrag, einen zusätzlichen manuellen Datentransfer durchzuführen, wenn Sie schließlich die validen Daten aus der Formular-Bean auf das Domänenobjekt verschieben wollten. Für viele Entwickler ist es eines der überzeugendsten Aspekte von Struts 2, dass das Framework die Daten transferiert, validiert und direkt an Domänenobjekte der Anwendung bindet, wo sie dann bleiben können!

Falls Sie lieber mit komplexen Objekten arbeiten als mit einfachen JavaBeans-Properties, um Daten zu empfangen, haben Sie eine Reihe verschiedener Möglichkeiten, um solche Transfers zu implementieren. Die erste Option basiert auch auf JavaBeans. Wir können ein komplexes Objekt selbst als JavaBeans-Property zur Verfügung stellen und die Daten direkt auf das Objekt verschieben lassen. Eine andere Alternative ist, die sogenannte `ModelDriven`-Action einzusetzen. Zu dieser Option gehören ein einfaches Interface und einer der Default-Interceptors. Wie die mit Objekten gepufferte JavaBeans-Property können wir auch mit einer `ModelDriven`-Action ein komplexes Java-Objekt zum Datenempfang nutzen. Die Unterschiede zwischen diesen beiden Methoden sind gering, und die Wahl zieht keine funktionalen Konsequenzen nach sich. Sie entscheiden sich da je nach Projektanforderung. Sie werden beide Techniken anhand von Beispielen aus Struts 2 Portfolio erlernen.

3.4.1 Objekt-gepufferte JavaBeans-Properties

Wir haben bereits gesehen, wie der `params`-Interceptor aus dem `defaultStack` automatisch Daten aus der Anfrage in unsere Actionobjekte transferiert. Um diesen Transfer zu aktivieren muss der Entwickler nur JavaBeans-Properties für seine Actions bereitstellen und dabei die gleichen Namen wie die Formularfelder verwenden, die übermittelt werden. Das ist einfach, aber trotzdem belastet uns häufig noch eine andere lästige Aufgabe. Diese besteht aus der Sammlung der einzeln transferierten Datenelemente und dem Transfer in ein Anwendungsdomänenobjekt, das wir selbst instanziieren müssen. Listing 3.7 zeigt die vorige Version der `execute()`-Methode von `Register`-Action.

Listing 3.7 Daten per Hand sammeln und das Domänenobjekt erstellen

```
public String execute(){

    User user = new User();
    user.setPassword( getPassword() );                    ❶
    user.setPortfolioName( getPortfolioName() );
    user.setUsername( getUsername() );

    getPortfolioService().createAccount( user );

    return SUCCESS;
}
```

Auch wenn wir vor ein paar Seiten beeindruckt waren, wie prägnant und qualitativ hochwertig diese Methode ist, können wir nun aber sehen, dass fünf der sieben Zeilen nichts anderes machen als die einzelnen Daten ❶ einzusammeln, die das Framework in unsere

einfachen JavaBeans-Properties transferiert hat. Wir sind zwar immer noch ganz baff, dass die Daten automatisch transferiert und an unsere Java-Datentypen gebunden werden, doch warum sollte uns das reichen?

Warum kann das Framework nicht die Daten direkt zu unserem `User`-Objekt transferieren? Warum kann das Framework nicht für uns das `User`-Objekt instanziieren? Weil Struts 2 leistungsfähige Möglichkeiten für den Datentransfer und die Typkonvertierung enthält, deren wahre Stärken wir später im Buch noch entdecken werden, fragen wir nach diesen Dingen und bekommen sie auch. In diesem Falle geht das ganz einfach: Wir schreiben unsere `Register`-Action um, damit sie die einzelnen JavaBeans-Properties durch eine einzige Property ersetzt, die auf dem `User`-Objekt selbst beruht. Listing 3.8 zeigt die neue Version unserer neuen Action, wie sie in der Klasse `manning.chapterThree.objectBacked.ObjectBackedRegister` implementiert wurde.

Listing 3.8 Eine mit Objekten gepufferte Property empfängt Datentransfers.

```
public String execute(){
    getPortfolioService().createAccount( user );   ❶
    return SUCCESS;
}

private User user;

public User getUser() {                          ❷
    return user;
}
public void setUser(User user) {
    this.user = user;
}
public void validate(){

    ...

    if ( getUser().getPassword().length() == 0 ){    ❸
        addFieldError( "user.password", getText("password.required") );
    }

    ...
}
```

Listing 3.8 hat unsere Businesslogik nun auf einen Einzeiler reduziert. Wir übergeben ein bereits instanziiertes und gefülltes `User`-Objekt an die Kontoerstellungsmethode des Serviceobjekts ❶. Das war’s schon. Diese Logik ist deutlich sauberer, weil wir dem Framework die Instanziierung unseres `User`-Objekts und das Befüllen seiner Attribute mit Daten aus der Anfrage überlassen. Vorher haben wir das selbst erledigt. Damit das Framework diese lästige Arbeit übernimmt, haben wir einfach die einzelnen JavaBeans-Properties durch eine einzige Property ersetzt, die auf dem `User`-Objekt selbst beruht ❷. Wir müssen nicht einmal das `User`-Objekt erstellen, auf dem die Property beruht, weil der Datentransfer des Frameworks dies für uns erledigt, wenn es damit beginnt, die Daten zu verschieben. Beachten Sie, dass unser Validierungscode nun in der Notation bei der `User`-Property starten muss um an die Datenelemente der einzelnen Felder zu kommen.

Entsprechend müssen wir auch ein paar Änderungen an der Art vornehmen, wie wir unsere Daten aus unseren Results referenzieren – in diesem Fall in den JSPs. Zuerst müssen wir die Feldnamen im Formular ändern, das an unsere neue Action übermittelt wird. Es läuft darauf hinaus, dass wir nun eine weitere Schicht in unserer Notation der JavaBeans-Properties haben. Das folgende Code-Snippet zeigt die kleine Änderung am `textfield`-Namen unseres Formulars, das sich auf der Seite Registration_OB.jsp findet:

```
<s:textfield name="user.username" label="Username"/>
```

Wie Sie sehen können, startet die Referenz nun beim `user`, um der Property in der Action zu entsprechen. Vorher war bei unserer Referenz der Teil mit dem `user` in dieser Referenz nicht erforderlich, als wir alle einzelnen Daten als individuelle JavaBeans-Properties zur Verfügung gestellt hatten. Die Namen der anderen Felder in diesem Formular werden entsprechend geändert.

Wir überarbeiten das andere Ende der Anfrage auf ähnliche Weise. Wenn wir die resultierende Erfolgsseite rendern, müssen wir die ausführlichere Property-Notation verwenden, um auf die Daten zuzugreifen. Das folgende Code-Snippet aus RegistrationSuccess_OB.jsp zeigt die neue Notation:

```
<h5>Congratulations! You have created </h5>
<h3>The <s:property value="user.portfolioName" /> Portfolio</h3>
```

Wie Sie sehen, lässt sich dem Framework noch mehr Arbeit zuschanzen, wenn wir ein Anwendungsdomänenobjekt direkt als JavaBeans-Property nutzen. Die geringfügige Konsequenz ist, dass wir einen Punkt mehr verwenden müssen, wenn wir unsere Daten aus den JSP-Seiten referenzieren. Nun schauen wir uns eine weitere Methode an, um Objekte für die Transfermöglichkeiten des Frameworks bereitzustellen. So bekommen wir die gleiche, sauberere `execute()`-Methode wie bei der mit Objekten gepufferten JavaBeans-Property, doch dabei ist es nicht nötig, dass wir diesen zusätzlichen Punkt mehr beim Datenzugriff unserer View-Schicht verwenden müssen.

3.4.2 ModelDriven-Actions

Durch `ModelDriven`-Actions nehmen wir Abstand von der Verwendung von JavaBeans-Properties für die Bereitstellung von Domänendaten. Stattdessen stellen sie über die `getModel()`-Methode, die vom `com.opensymphony.xwork2.ModelDriven`-Interface deklariert wird, ein Anwendungsdomänenobjekt zur Verfügung. Diese Methode führt ein neues Interface und auch einen neuen Interceptor ein und ist einfach zu verwenden. Der Interceptor befindet sich bereits im Default-Stack; der Datentransfer erfolgt immer noch automatisch, und man kann mit ihm sogar noch leichter arbeiten als mit früheren Techniken. Schauen wir uns an, wie das funktioniert.

Die Interface-Implementierung setzt praktisch nichts voraus. Wir müssen deklarieren, dass unsere Action das Interface implementiert, doch über die `ModelDriven`-Action wird einzig die `getModel()`-Methode zur Verfügung gestellt. Mit *Model* meinen wir das Model im Sinne von MVC. In diesem Fall sind es die Daten, die aus der Anfrage eintreffen und

durch die Ausführung der Businesslogik verändert werden. Diese Daten gehen dann zum View, im Fall unseres Struts 2 Portfolios sind das JSP-Seiten. Listing 3.9 zeigt den neuen Action-Code aus der Klasse `manning.chapterThree.modelDriven.ModelDrivenRegister`.

Listing 3.9 Automatischer Transfer von Anfragedaten an Anwendungsdomänenobjekte

```
public class ModelDrivenRegister extends ActionSupport
                  implements ModelDriven { ❶

    public String execute(){
        getPortfolioService().createAccount( user );
        return SUCCESS;
    }

    private User user = new User(); ❷
    public Object getModel() {
        return user;
    }

    public void validate(){

    ...

        if ( user.getPassword().length() == 0 ){
            addFieldError( "password", getText("password.required") ); ❸
        }

    ...

    }

    ...
}
```

Zuerst sehen wir, dass unsere neue Action das Interface implementiert ❶. Die einzige bei diesem Interface erforderliche Methode ist `getModel()`, die unser Model-Objekt zurückgibt: das vertraute `User`-Objekt. Beachten Sie, dass wir mit der `ModelDriven`-Methode das `User`-Objekt selbst initialisieren müssen ❷. Warum, sehen wir in Kapitel 5, wenn wir die Details des Datentransfermechanismus untersuchen, doch im Moment brauchen Sie nur auf dieses kleine, aber wichtige Detail zu achten.

Ein Fallstrick soll hier erwähnt werden: Wenn die `execute()`-Methode Ihrer `ModelDriven`-Action aufgerufen wurde, hat das Framework sich eine Referenz für Ihr Model-Objekt besorgt, die während der gesamten Anfrage verwendet wird. Weil das Framework seine Referenz von Ihrem Getter bekommt, wird es nicht merken, wenn Sie intern in Ihrer Action das Model-Feld ändern. Das kann zu Problemen mit der Datenkonsistenz führen. Wenn Sie während der Codeausführung das Objekt ändern, auf das Ihre Model-Feld-Referenz zeigt, wird das Model Ihrer Action nicht mehr mit dem des Frameworks übereinstimmen. Das folgende Code-Snippet demonstriert dieses Problem:

```
public String execute(){
    user = new User();
    user.setSomething();
    getPortfolioService().createAccount( user );
    return SUCCESS;
}
```

```
private User user = new User();
public Object getModel() {
    return user;
}
```

In der `execute()`-Methode dieser Action hat der Entwickler aus irgendeinem Grund die `user`-Referenz auf ein neues Objekt gesetzt. Doch das Framework hat immer noch eine Referenz auf das Originalobjekt, wie es in der Instanzfelddeklaration für `user` initialisiert wurde. Wenn das Framework das Result aufruft, wird der Datenzugriff Ihrer JSP-Seite anhand des alten Objekts aufgelöst. Was immer auch dieser fehlerhafte Code gesetzt hat, es ist nun nicht verfügbar. Sie können dieses ursprüngliche Model-Objekt natürlich ganz nach Belieben manipulieren. Machen Sie halt nur kein neues oder lassen Sie die vorhandene Referenz auf ein anderes zeigen!

Wie bei der vorigen, auf einem Objekt beruhenden JavaBeans-Property-Methode haben wir durch die Verwendung eines Domänenobjekts zum Empfang aller Daten den Luxus einer sauberen `execute()`-Methode. Hier nehmen wir wieder einen kleinen Nachteil in Kauf, der mit der Tiefe unserer Referenzen zu tun hat. Wie Sie im Validierungscode von Listing 3.9 sehen können, greifen wir nun das Passwort auf dem Weg zum Passwortfeld durch eine Referenz über das Model-Objekt zu ❸.

Doch die Referenztiefe in der View-Schicht hat für uns keine weiteren Auswirkungen. Alle Referenzen in den JSP-Seiten werden wieder so einfach wie die ursprüngliche `Register`-Action, die für den Datentransfer die einfachen, individuellen JavaBeans-Properties verwendet hat. Die folgenden Code-Snippets aus Registration_MD.jsp

```
<s:textfield name="username" label="Username"/>
```

und RegistrationSuccess.jsp

```
<h5>Congratulations! You have created </h5>
<h3>The <s:property value="portfolioName" /> Portfolio</h3>
```

zeigen wieder die Einfachheit der View-Schicht-Referenzen auf Daten, die in der `ModelDriven`-Action enthalten sind. Dies wird als einer der wichtigsten Gründe für die Nutzung der `ModelDriven`-Methode betrachtet, wenn man sie mit der Methode der mit Objekten gepufferten JavaBeans-Property vergleicht, die dem Datentransfer die Domänenobjekte zur Verfügung stellt.

Wenn man Domänenobjekte für den Datentransfer nimmt, ist das eine prima Sache, doch dabei muss eine bestimmte Sache beachtet werden. Wir werden diese potenzielle Gefahr als Nächstes untersuchen.

3.4.3 Domänenobjekte für Datentransfer – Eine Abschlussbemerkung

Zuerst wollen wir auf eine potenzielle Gefahr hinweisen, wenn man Domänenobjekte für den Datentransfer einsetzt. Das Problem entsteht, wenn die Daten automatisch in das Objekt transferiert werden. Wie bereits gesehen, werden die Daten auf diese Attribute verschoben, wenn die Anfrage Parameter aufweist, die zu den Attributen des Domänenobjekts

passen. Nun stellen Sie sich einmal den Fall vor, dass Ihre Domänenobjekte sensible Datenattribute enthalten, die Sie beim automatischen Datentransfer eben nicht übermitteln wollen, vielleicht eine ID. Ein böswilliger User könnte einen entsprechend benannten Querystring-Parameter an die Anfrage hängen, sodass der Wert dieses Parameters automatisch auf das Attribut Ihres bereitgestellten Objekts geschrieben wird. Natürlich können Sie diese Attribute aus dem Objekt entfernen, doch dann geht der Vorteil verlustig, vorhandene Objekte erneut verwenden zu können, statt neue schreiben zu müssen. Leider gibt es für dieses Problem noch keine gute Lösung. Normalerweise brauchen Sie sich darüber keine Sorgen zu machen, doch es ist etwas, das Sie beim Schreiben Ihrer Actions im Hinterkopf behalten sollten.

Letzten Endes bleibt es Ihnen überlassen, welche Methode Sie für den Empfang der Daten aus dem Framework nehmen. Jede dient ihrem Zweck, und wir gehen davon aus, dass schließlich Ihre Projektanforderungen über den besten Ansatz bestimmen. Im weiteren Verlauf des Buches werden wir viele Beispiele von Best Practices und der Integration mit anderen Technologien sehen, in denen einige Fälle genauer ausgeführt werden, wo die eine oder andere Methode am dienlichsten ist. Manchmal passt es auch, ein wenig von allen zu nehmen. Haben wir schon erwähnt, dass Sie bei Bedarf auch alle gleichzeitig nehmen können? Die Plattform ist wirklich sehr flexibel. Schauen wir uns nun eine Fallstudie an.

3.5 Dateiupload – Eine Fallstudie

Nun haben Sie mittlerweile die Tools, die Sie zum Schreiben der Actionkomponenten Ihrer Anwendung brauchen, und wissen, wie man sie mit einer rudimentären Struts-2-Anwendung verschaltet. Wir vermuten sogar, dass Sie schon soviel ableiten konnten, um mit der Implementierung einer View-Schicht mit JSP-Results beginnen zu können. Später im Buch, wenn es in Teil 3 um Results, Tags, Benutzerschnittstellenkomponenten und Ajax-Integration geht, werden Sie sehen, wie viel mehr das Framework Ihnen für die View-Schicht bieten kann. Nun runden wir unsere Ausführungen über die Actionkomponente mit einer praktischen Fallstudie ab. Dabei werden Sie nicht nur selbst aktiv, sondern können auch noch einmal wiederholen, wie Actions und Interceptors zusammenarbeiten, um die üblichen Probleme bei Webanwendungen zu lösen.

Irgendwann muss wohl jeder mal einen Dateiupload implementieren. Bei unserer Beispielanwendung müssen Bilddateien hochgeladen werden, weil Struts 2 Portfolio ansonsten recht trist wäre. Wir nehmen uns den Dateiupload deshalb gerade an dieser Stelle vor, weil man unserer Meinung nach leichter das persistente Muster des Frameworks demonstrieren kann, anhand von Interceptors die Logik häufig vorkommender Aufgaben aus der Action selbst herauszunehmen. Schauen wir uns also an, wie unsere Actions mit einem Interceptor aus dem Default-Stack arbeiten können, um einen supersauberen und komplett wiederverwertbaren Dateiupload hinzukriegen.

3.5.1 Eingebauter Support über das Paket struts-default

Wie bei den meisten Routineaufgaben finden Sie auch für den Dateiupload eine eingebaute Hilfe bei Struts 2. In diesem Fall enthält der Default-Interceptor-Stack den `FileUpload-Interceptor`. Wie Sie sich vielleicht erinnern, ist struts-default.xml die Systemdatei, über die alle eingebauten Komponenten definiert werden. Listing 3.10 zeigt die Elemente aus der Datei, die den `fileUpload`-Interceptor deklarieren und ihn zum Bestandteil des Default-Interceptor-Stacks machen.

Listing 3.10 Deklaration des `FileUploadInterceptors` und Einfügen in den Stack

```
<package name="struts-default">
    <interceptors>

    ...

    <interceptor name="fileUpload"
      class="org.apache.struts2.interceptor.FileUploadInterceptor"/>

    ...

    </interceptors>

    ...

    <interceptor-stack name="defaultStack">

        ...

        <interceptor-ref name="model-driven"/>
        <interceptor-ref name="fileUpload"/>
        <interceptor-ref name="params"/>

        ...

    </interceptor-stack>
</package>
```

Wie Sie sehen können, enthält das Paket `struts-default` eine Deklaration des `file-Upload`-Interceptors, der auf der Implementierungsklasse `org.apache.struts2.interceptor.FileUploadInterceptor` beruht. Dieser Interceptor wird dann dem `default-Stack` hinzugefügt, sodass bei allen Paketen, die das `struts-default`-Paket erweitern, automatisch dieser Interceptor in den Actions eingesetzt werden kann. Wir erweitern die Pakete von Struts 2 Portfolio nun dieses Paket, um diese eingebauten Komponenten nutzen zu können.

3.5.2 Was macht der fileUpload-Interceptor?

Der `fileUpload`-Interceptor erstellt eine spezielle Version des bereits bekannten automatischen Datentransfermechanismus. Bei den früheren Datentransfers hatten wir es mit dem Transfer von Formularfelddaten aus der Anfrage zu den passenden JavaBeans-Properties unserer Actionobjekte zu tun. Der `params`-Interceptor (ebenfalls Teil des `defaultStacks`) war verantwortlich dafür, alle Anfrageparameter auf das Actionobjekt zu verschieben, sobald die Action einer JavaBeans-Property zum Namen des Anfrageparameters passte. In

Listing 3.10 sehen Sie, dass der `defaultStack` den `fileUpload`-Interceptor direkt vor den `params`-Interceptor platziert. Wenn der `fileUpload`-Interceptor ausgeführt wird, verarbeitet er eine mehrteilige Anfrage und transformiert die Datei und auch einige Metadaten in Anfrageparameter, indem er einen Wrapper um die Servlet-Anfrage legt. Tabelle 3.2 zeigt die Anfrageparameter, die von diesem `fileUpload`-Interceptor hinzugefügt werden.

Tabelle 3.2 Anfrageparameter, die vom FileUpload-Interceptor bereitgestellt werden

Parametername	Parametertyp und Wert
[Dateiname aus Formular]	`File` – die hochgeladene Datei selbst
[Dateiname aus Formular]`ContentType`	`String` – der Inhaltstyp der Datei
[Dateiname aus Formular]`FileName`	`String` – der Name der hochgeladenen Datei, wie sie auf dem Server gespeichert wird

Nachdem der `fileUpload`-Interceptor die Teile der mehrteiligen Anfrage als Anfrageparameter zur Verfügung gestellt hat, wird es Zeit, dass der nächste Interceptor im Stack seine Arbeit macht. Praktischerweise ist der nächste der `params`-Interceptor. Wenn dieser feuert, verschiebt er alle Anfrageparameter einschließlich derjenigen, die in Tabelle 3.2 aufgeführt sind, auf das Actionobjekt. Somit braucht ein Entwickler nur seinem Actionobjekt die JavaBeans-Properties hinzuzufügen, die zu den Namen in Tabelle 3.2 passen, um ganz bequem zu seinem Dateiupload zu kommen.

Zu Ihrer Info

Die elegante Verwendung von Interceptors wie z.B. durch den `fileUpload`-Interceptor sollte hier noch einmal besonders betont werden. Wie bereits erwähnt, versucht das Struts-2-Framework alles, um seine Actionkomponenten so sauber wie möglich zu halten. Das macht es vor allem durch den Einsatz von Interceptors, um querschnittliche Aufgaben aus den zentralen Aufgaben der Verarbeitung der Action selbst auszulagern. Der `fileUpload`-Interceptor zeigt dies durch seine Kapselung der Verarbeitung mehrteiliger Anfragen und die Injektion der verarbeiteten Uploaddaten in die JavaBeans-Setter-Methoden des Actionobjekts.

Wir haben auch auf die Rolle der Interceptors hinsichtlich der Vor- und Nachverarbeitung verwiesen. Im Falle des `fileUpload`-Interceptors besteht die Vorverarbeitung aus der Transformation der mehrteiligen Anfrage in Anfrageparameter, die der `params`-Interceptor automatisch in unsere Action verschieben wird. Die Nachverarbeitung findet statt, wenn der Interceptor nach der Action erneut feuert, um die temporäre Version der hochgeladenen Datei zu beseitigen.

Wie sieht das alles im Code aus? Das Struts 2 Portfolio verwendet das Hochladen von Dateien, also schauen wir uns das an.

3.5.3 Der Beispielcode von Struts 2 Portfolio

Das Struts 2 Portfolio lädt über diesen Dateiupload-Mechanismus neue Bilder in das Portfolio. Bei einer solchen Aufgabe muss als Erstes ein Formular präsentiert werden, mit dem

ein Benutzer Dateien hochladen kann. Sie können die Seite für den Bilder-Upload besuchen, indem Sie zuerst ein Konto mit der Kapitel-3-Version von Struts 2 Portfolio erstellen. Der Enduser-Workflow ist momentan unvollständig, doch das beheben wir in den nächsten Kapiteln. Nachdem Sie ein Konto erstellt haben, wählen Sie, dass Sie mit dem Portfolio arbeiten und ein neues Bild hinzufügen wollen. Sie sehen eine Seite, die Ihnen ein einfaches Formular zum Upload eines Bildes präsentiert. Das folgende Code-Snippet aus chapterThree/ImageUploadForm.jsp zeigt das Markup für dieses Formular:

```
<h4>Complete and submit the form to create your own portfolio.</h4>
<s:form action="ImageUpload" method="post" enctype="multipart/form-data">
    <s:file name="pic" label="Picture"/>
    <s:submit/>
</s:form>
```

Wenn wir dieses Formular erstellen, müssen wir ein paar Punkte beachten: Erstens wird das Formular anhand von Struts-2-Tags erstellt. Wir behandeln die Struts-2-Tag-Library in den Kapiteln 6 und 7. Jetzt nehmen Sie bitte einfach hin, dass dieses Tag das HTML-Markup eines Formulars generiert, mit dem der Benutzer eine Datei hochladen kann. Dann ist der Encoding-Typ des Formulars auf `multipart/form-data` gesetzt worden. Dieses wichtige Attribut teilt dem Framework mit, dass die Anfrage als Upload zu behandeln ist. Ohne diese Einstellung wird es nicht funktionieren. Schließlich beachten Sie bitte, dass die Datei durch das Formular unter dem `name`-Attribut übermittelt wird, das wir beim `file`-Tag ergänzt haben. Dieses Detail ist wichtig, weil Sie diesen Namen verwenden werden, um die JavaBeans-Properties zu erstellen, die die Upload-Daten erhalten.

Da unser JSP für die Formularseite nun fertig ist, schauen wir uns die Action an, die den Upload erhalten und verarbeiten wird. Zuerst achten Sie bitte darauf, dass das Paket, zu dem Ihre Action gehört, das `struts-default`-Paket erweitert, damit es den `fileUpload`-Interceptor und den Default-Interceptor-Stack erbt. Das Listing 3.11 zeigt in einem Snippet aus der Datei manning/chapterThree/chapterThree.xml, dass wir das gemacht haben.

Listing 3.11 Erweitern des `struts-default`-Pakets, um die Upload-Verarbeitung zu vererben

```
<package name="chapterThreeSecure" namespace="/chapterThree/secure"
   extends="struts-default">

  ...

    <action name="AddImage" >
        <result>/chapterThree/ImageUploadForm.jsp</result>
    </action>

    <action name="ImageUpload" class="manning.chapterThree.ImageUpload">
        <result>/chapterThree/ImageAdded.jsp</result>
        <result name="input">/chapterThree/ImageUploadForm.jsp</result>
    </action>

  ...

</package>
```

Dies ist unser Paket der sicheren Actions für Struts 2 Portfolio. Wir haben noch keine Sicherheitsmaßnahmen eingebaut, aber wir wissen, dass diese Actions bestimmte Sicherheitsvorkehrungen erfordern, also haben wir sie schon einmal in ein separates Paket gelegt. In Kapitel 4 werden wir die Sicherheit mit einem angepassten Interceptor hinzufügen.

Nach dem `defaultStack` und dessen Dateiupload-Interceptor müssen wir unserem Actionobjekt nur Properties hinzufügen, die zu den Parameternamen aus Tabelle 3.2 passen.

Wir haben bereits gesehen, dass unsere Datei unter dem Namen `pic` übermittelt wird. Anhand der Namenskonventionen in Tabelle 3.2 können wir die Namen der zu implementierenden JavaBeans-Properties ableiten. Listing 3.12 zeigt die JavaBeans-Properties, die von der Klasse `manning.chapterThree.ImageUpload` implementiert werden. (Wie immer erfahren Sie weiteres aus dem Quellcode von Struts 2 Portfolio, wenn Sie noch mehr wissen wollen.)

Listing 3.12 Die JavaBeans-Properties, die die hochgeladene Datei und die Metadaten erhalten

```
File pic;
String picContentType;
String picFileName;

public File getPic() {
    return pic;
}

public void setPic(File pic) {
    this.pic = pic;
}

public String getPicContentType() {
    return picContentType;
}

void setPicContentType(String picContentType) {
    this.picContentType = picContentType;
}

public void setPicFileName(String picFileName) {
    this.picFileName = picFileName;
}

public String getPicFileName() {
    return picFileName;
}
```

Sie müssen die nicht unbedingt alle implementieren. Wenn Sie einige nicht implementieren, dann empfangen Sie halt einfach die Daten nicht. Kein Problem! Jedenfalls ist die Verwendung des `fileUpload`-Interceptors beinahe so einfach wie diese JavaBeans-Properties zu schreiben. Dank der Trennung von der Upload-Logik ist die Arbeit der Action selbst einfach. Wie im folgenden Code-Snippet aus der `ImageUpload`-Action ersichtlich, kann die Action sich ganz auf die anstehende Aufgabe konzentrieren:

```
public String execute(){
    getPortfolioService().addImage( getPic() );
    return SUCCESS;
}
```

Hier gibt es nur den Aufruf der Businesslogik. Die Bilddatei ist praktischerweise nur einen Getter weit weg, so wie das bei allen automatisch transferierten Daten der Fall ist. Dank des Teamworks von `fileUpload`- und `params`-Interceptor ist ein Dateiupload so einfach wie der Umgang mit Primitiven. Übrigens können Sie in chapterThree.xml den Pfad mit einem Parameter für das Bildupload-Actionelement auf das Verzeichnis setzen, in dem die Action die Bilddatei speichern wird.

Nun schauen wir uns ein paar Nachbesserungen an, die Sie beim `fileUpload`-Interceptor machen können, damit auch so etwas wie mehrere Dateiuploads gleichzeitig möglich sind.

3.5.3.1 Mehrere Dateien und andere Einstellungen

Auch der Upload mehrerer Dateien mit den gleichen Parameternamen wird unterstützt. Sie müssen nur die JavaBeans-Properties Ihrer Action zu Arrays ändern, d.h. `File` wird zu `File[]` und die beiden Strings zu String-Arrays. Die drei Arrays haben immer die gleiche Länge, und ihre Reihenfolge ist die gleiche, was bedeutet, dass Index 0 für alle drei Arrays die gleiche Datei und Dateimetadaten repräsentiert. Es gibt für den `fileUpload`-Interceptor auch viele andere konfigurierbare Parameter. Das reicht von der maximalen Dateigröße bis zur Implementierung des mehrteiligen Anfrage-Parsers, mit dem die Anfrage bearbeitet wird. Insgesamt können durch die extreme Flexibilität des Struts-2-Frameworks in einem Buch wie dem vorliegenden unmöglich alle Details angesprochen werden. In diesem Buch sollen so viele irrelevante Details wie möglich ausgesiebt werden, damit die Konzepte des Frameworks klarer werden. Für weitere Details und Zusatzinfos dient die Website von Struts 2 als gute Referenz.

3.6 Zusammenfassung

In diesem Kapitel lernten wir eine Menge über das Erstellen von Struts-2-Actions. Wir begannen unsere Tour durch diese wichtige Komponente mit der Untersuchung der Rolle, die Actions innerhalb des Frameworks spielen. Actions machen drei Sachen: Erstens und am wichtigsten ist, dass Sie die Interaktion des Frameworks mit dem Model kapseln. Das bedeutet letzten Endes, dass die Aufrufe der Businesslogik und der Datenschicht in der `execute()`-Methode der Action-Klasse zu finden sind. Der zweite Job der Action ist, als Datentransferobjekt für die Anfrageverarbeitung zu dienen. Wir gehen davon aus, diesen Punkt mittlerweile überdeutlich klargemacht zu haben. Schließlich ist die Action auch verantwortlich dafür, einen Control-String zurückzugeben, der vom Framework verwendet werden kann, um die entsprechende Result-Komponente auszuwählen, die die View beim User rendert.

Wir haben auch gesehen, wie man die Actionkomponenten in Struts-2-Pakete verpackt. Mit diesen Paketen bekommt man eine logische Organisation der Frameworkkomponenten, vor allem der Actions. Mittels der Paketstruktur können wir verschiedene wichtige Dinge machen. Wir können URL-Namensräume auf Gruppen von Actions mappen. Wir können uns auch die Vererbungsmechanismen der Pakete zunutze machen, um wiederverwendbare Gruppen von Frameworkkomponenten zu definieren. Wir haben dieses Feature bereits verwendet, indem wir unsere Struts 2 Portfolio-Pakete das eingebaute `struts-default`-Paket erweitern ließen, um u.a. dessen Default-Interceptor-Stack nutzen zu können. Dies soll als Modell für die Erstellung eigener Pakethierarchien dienen.

Dann stellten wir verschiedene Schlüsselfiguren des Frameworks vor, die einem die Arbeit erleichtern. Zuerst war dies das `Action`-Interface, das einige wichtige Definitionen von

Konstanten enthält, die das Framework für häufig verwendete Control-Strings benutzt. Danach beschäftigten wir uns ausführlich mit der Funktionalität der `ActionSupport`-Klasse. Diese nützliche Klasse implementiert mehrere wichtige Interfaces und kooperiert mit zentralen Interceptors des `defaultStack`s, um eingebaute Implementierungen solch wichtiger Domänenaufgaben wie Validierung und grundlegende Internationalisierung zu bieten. Wir haben dies alles anhand der Beispielanwendung Struts 2 Portfolio demonstriert.

Eine der wichtigeren Überlegungen, wenn es um die Implementierung eigener Struts-2-Actions geht, wird die verwendete Methode für den Datentransfer sein. Dafür stehen verschiedene Optionen bereit. Wir führten zwei Methoden an, die beide JavaBeans-Properties beim Actionobjekt selbst implementieren. Bei der ersten entsprechen einfache Properties jeweils den individuellen Parametern der eingehenden Anfrage. Die andere Methode mit JavaBeans-Properties stellt Properties bereit, die von komplexen Domänenobjekten gepuffert sind. Schließlich sahen wir, dass man auch eine völlig andere Methode nehmen kann, um komplexe Domänenobjekte durch Implementierung von `ModelDriven`-Actions zur Verfügung zu stellen. Die Wahl der Methode hängt weitgehend von den Anforderungen des Projekts und der vorliegenden Action ab. Flexibilität ist ein immer wiederkehrendes Thema des Struts-2-Frameworks.

Das Kapitel schloss mit einer Fallstudie ab. Wir schauten uns die Arbeit mit einem der eingebauten Interceptors des Frameworks an, mit dem man die Beispielanwendung um eine Dateiupload-Action erweitern kann. Zwar ist der Upload von Dateien mit Struts 2 recht leicht, doch wir wiesen auch auf einige wichtige Lektionen hin, die dieses Beispiel über das Framework selbst demonstrierte. Insbesondere zeigten wir, was wir mit der Feststellung meinen, dass das Framework eine saubere Implementierung von MVC zu bieten versucht. Speziell das Beispiel mit dem Dateiupload zeigt, wie eine saubere Kooperation zwischen Interceptors und Actions zu einer Webanwendung führt, die wiederverwertbare und flexible Kapselungen von querschnittlichen Aufgaben sowie supersaubere Actions ermöglicht.

Nun wissen Sie also über Actions Bescheid. Als Nächstes steht ein detaillierter Blick auf Interceptors an, und wir erweitern das Struts 2 Portfolio, indem wir diese nutzbringend einsetzen.