Diplomarbeit

Herstellung von Konsistenz und Validität in Web-Anwendungen

Robert F. Beeger

September 2003

Erstbetreuung: Prof. Dr. Heinz Züllighoven
Zweitbetreuung: Prof. Dr. Norbert Ritter

Fachbereich Informatik
Arbeitsbereich Softwaretechnik (SWT)
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Diese Diplomarbeit wurde am Fachbereich Informatik der Universität Hamburg zur teilweisen Erfüllung der Anforderungen zur Erlangung des Titels Diplom-Informatiker vorgelegt

Erklärung
Hiermit versichere ich, diese Arbeit selbstständig und unter ausschließlicher Zuhilfenahme der in der Arbeit aufgeführten Hilfsmittel erstellt zu haben.

Hamburg, den 01.09.2003

Robert F. Beeger
Schottweg 14
22087 Hamburg

Matrikelnummer: 4824555

Betreuung
Erstbetreuer: Prof. Dr. Heinz Züllighoven
Zweitbetreuer: Prof. Dr. Norbert Ritter
Reviewer: Dipl. Inform. Henning Wolf

Prof. Dr. Heinz Züllighoven
Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Prof. Dr. Norbert Ritter
Fachbereich Informatik
Arbeitsbereich Verteilte Systeme und Informationssysteme
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Inhalt

Abbildungen

Danksagung

Ich danke

- Prof. Heinz Züllighoven für die Erstbetreuung und das richtungsweisende Feedback
- Prof. Norbert Ritter für die Zweitbetreuung
- Henning Wolf für das Review und das wertvolle Feedback

Ebenfalls bedanken möchte ich mich bei meinen Kollegen bei der it-wps für die gute Atmosphäre, die dazu beigetragen hat, dass ich immer wieder mit neuer Energie zu dieser Arbeit zurückkehren konnte.
Nicht zuletzt gilt mein Dank auch meiner Familie für die Unterstützung und den in mich gesetzten Glauben.

1 Einleitung

> *Born in the ice-blue waters of the festooned Norwegian coast; amplified (by an aberration of world currents, for which marine geographers have yet to find a suitable explanation) along the much grayer range of the Californian Pacific; viewed by some as a typhoon; by some as a tsunami, and by some as a storm in a teacup – a tidal wave is hitting the shores of the computing world.*
>
> — Bertrand Meyer : *Object Oriented Software Construction – Second Edition*

1.1 Motivation

Web-Anwendungen, die heutzutage konstruiert werden, benutzen meistens eine durch HTML-Seiten[1] spezifizierte Oberfläche, die von einem Browser[2] angezeigt wird. Selten werden noch Applets[3] benutzt. Nach dem anfänglichen Applet-Boom stellte sich schnell eine Art von Ernüchterung ein. Obwohl Applets annähernd die gleichen Möglichkeiten bieten, die in Desktop-Anwendungen existieren, haben sie entscheidende Nachteile. Applets brauchen lange, ehe sie auf den Computer des Benutzers heruntergeladen sind, und dann dauert es auch eine gewisse Zeit bis sie gestartet sind. Diese Verzögerungen wurden schnell als unzumutbar empfunden, weshalb es eine Rückkehr zu HTML-basierten Web-Anwendungen gab.

HTML-Seiten bieten dem Benutzer zwei Arten von Interaktion an. Einerseits kann der Anwender Verweisen auf andere Seiten folgen, indem er auf bestimmte Textstellen oder Bilder klickt. Andererseits kann er über das Ausfüllen von HTML-Formularen dem Servercomputer, von dem er die HTML-Seiten anfordert, mehr Informationen übermitteln als dies durch einen Klick auf einen Verweis möglich ist.

Auf den ersten Blick, scheinen **HTML-Formulare** für einen mit JWAM[4] vertrauten Entwickler dieselben Möglichkeiten zu bieten wie **JWAM-Formulare**[5] in Desktop-Anwendungen. Es wird definiert, welche Felder ein Formular hat, und ein generisches Werkzeug kümmert sich darum, dass das Formular in einer passenden Form angezeigt und ausgefüllt werden kann.

Aber schon ein zweiter Blick deckt hier gravierende Unterschiede auf.

In einem JWAM-Formular wird definiert, welche Felder es gibt und welchen Typ diese Felder haben. Der Typ eines Feldes ist der Typ der **Fachwerte**, die in dieses eingefüllt werden dürfen. Ein Fachwert ist ein unveränderliches Objekt, das von einer **Fachwert-Fabrik**[6] erzeugt wird. Eine solche Fabrik kann einen Fachwert meistens aus einer Zeichenkette erzeugen. Eine wichtige Eigenschaft einer Fachwert-Fabrik ist, dass sie entscheiden kann, ob eine Zeichenkette eine valide Repräsentation eines Fachwertes des Typs ist, von dem die Fabrik neue Exemplare erzeugen kann. So wird eine Fabrik, die Wochentag-Fachwerte erzeugen kann, „Freitag“ als eine valide Zeichenkettenrepräsentation akzeptieren, nicht aber „Blablub“. Durch die Verwendung von Fachwerten wird in JWAM-Forumularen Validität auf Feldebene[7] hergestellt.

---

[1] Abschnitt 3.1 gibt eine kurze Einführung in HTML

[2] Ein Browser ist ein Programm, das zur Anzeige von HTML-Dateien benutzt wird. Browser besitzen typischerweise auch die Fähigkeit, die anzuzeigenden HTML-Dateien über das Internet von entfernten Rechnern abzurufen

[3] Ein Applet ist ein meistens kleines Java-Programm, das in einer HTML-Seite eingebettet ausgeführt wird. Applets werden verwendet, um in HTML-Seiten dynamischen Inhalt zu präsentieren. Das reicht von Nachrichtentickern, die die neuesten Schlagzeilen präsentieren, bis zu Computerspielen

[4] JWAM ist ein Anwendungsrahmenwerk für die Entwicklung nach dem Werkzeug- und Material-Ansatz (Siehe hierzu auch [Züllighoven 98] und [JWAM-website])

[5] JWAM-Formulare sind eine Implementierung von Formularen in Software. Siehe Abschnitt 2.4 für eine tiefer gehende Betrachtung von Formularen, ihren Software-Gegenstücken im Allgemeinen und den JWAM-Formularen im Speziellen

[6] Fachwert-Fabriken implementieren das Fabrik-Muster, das in [Gamma 98] beschrieben ist

[7] Die verschiedenen Ebenen der Validierung werden in Abschnitt 2.1 beschrieben

In HTML-Formularen kann nur festgelegt werden, ob ein Feld eine Zeichenkette, ein Passwort oder eine Auswahl eines Wertes aus mehreren enthalten soll. Außerdem kann bei Feldern, die Zeichenketten oder Passwörter aufnehmen, festgelegt werden, wie lang die Eingabe sein darf. Hier sind die Möglichkeiten, mit denen Validität hergestellt und sichergestellt werden kann, um einiges beschränkter als dies bei den JWAM-Formularen der Fall ist.

In einem JWAM-Formular werden die Felder und die Typen der Fachwerte definiert, die diese Felder aufnehmen sollen. Hier erfolgt aber keine Zuordnung zu **Widgets**[8], die für die Befüllung der Felder benutzt werden sollen. Dafür gibt es eine als **Singleton**[9] realisierte Klasse `PFFactory`. An dieser Klasse kann eine Zuordnung von einem Fachwert-Typ zu einem für die Eingabe eines solchen Fachwertes zu verwendenden Widget vorgenommen werden. Damit wird auch Konsistenz hergestellt, denn für einen Fachwert-Typ wird nun immer dieselbe Art von Widget verwendet. Der Anwender wird also nicht mit immer neuen Widgets für die Eingabe von Fachwerten desselben Typs konfrontiert. Da die verwendeten Widgets speziell für die Aufgabe der Eingabe von Fachwerten erstellt werden, können sie auch die Fähigkeit der Fachwert-Fabriken, eine eingegebene Zeichenkette zu validieren, nutzen und so dem Benutzer während der Eingabe schon Feedback darüber geben, ob das, was er eingegeben hat, eine korrekte Eingabe für einen Fachwert-Typ ist.

In HTML-Formularen wird nicht nur definiert, welche Felder es gibt, sondern es wird auch definiert, mit welchen Widgets diese Felder zu befüllen sind. Hier wird also Konsistenz nicht sichergestellt. In einem Fall kann zum Beispiel für die Eingabe eines Wochentages eine Drop-Down-Liste mit den möglichen Wochentagen angeboten werden. In einem anderen Fall kann es sich um ein gewöhnliches Textfeld oder eine Reihe von Radio-Knöpfen handeln. Obwohl der Anwender also mehrmals Fachwerte desselben Typs eingeben muss, muss er sich jedes Mal auf eine neue Art der Eingabe umstellen. Ein Lerneffekt der Art ,,Den Wochentag, an dem ich geboren wurde, soll ich jetzt eingeben. Da habe ich doch bestimmt eine Drop-Down-Liste zur Verfügung, aus der ich den sechsten Eintrag für den Samstag auswählen kann.“ wird dadurch verhindert. Natürlich kann der Entwickler selbst für Konsistenz in seinen HTML-Seiten sorgen, indem er darauf achtet, dass für dieselben Typen von Fachwerten dieselben Typen von Widgets verwendet werden. Aber das ist auch einer der großen Nachteile gegenüber den JWAM-Formularen. Bei der Verwendung von HTML-Formularen muss der Entwickler selbst für diese Konsistenz sorgen, während ihm bei der Verwendung von JWAM-Formularen diese Arbeit abgenommen wird.

Ein weiterer Nachteil von HTML-Formularen gegenüber den JWAM-Formularen ist der Mangel der Möglichkeit, die eingegebenen Daten zu validieren. Da die Widgets diese Validierung nicht durchführen können, weil eine genaue Spezifizierung des Fachwert-Typs am Widget nicht möglich ist, muss diese auf dem Servercomputer erfolgen. Diese serverseitige Validierung muss aber vom Entwickler der Web-Anwendung in Gang gesetzt werden.

HTML-Formulare sind das mächtigste Mittel der Interaktion, das von HTML-Seiten angeboten wird. Trotz dieser Mächtigkeit sind sie den JWAM-Formularen, die bei Desktop-Anwendungen ihren Einsatz finden, weit unterlegen.

Diese Unterlegenheit äußert sich in folgenden Punkten:

[8] ,,Widget“ ist ein Kunstwort und eine Neuschöpfung aus den beiden Wörtern ,,Window“ und ,,Gadget“. Ein ,,Widget“ ist also ein ,,Fenster-Dings“, etwas, das in einem Fenster zu sehen ist. Die Dinge, die man in Fenstern – den Fenstern auf einem Bildschirm – sieht, sind Knöpfe, Eingabefelder, Optionsfelder und ähnliches. All diese Dinge sind Widgets.

[9] Ein weiteres Muster aus [Gamma 98]. Dieses stellt sicher, dass es von einer Klasse nur ein einziges zentral zugängliches Exemplar gibt.

- Die Möglichkeit den Typ eines Feldes zu definieren ist sehr eingeschränkt. Validität kann hier also nicht sichergestellt werden.
- Der Typ eines Feldes wird zusammen mit dem Layout definiert. Es kann nicht sichergestellt werden, dass für einen bestimmten Feldtyp immer die gleiche Art von Widgets verwendet wird. Konsistenz der Benutzungsoberfläche kann also nicht sichergestellt werden.
- Die eingegebenen Daten können nicht validiert werden. Es kann nur sichergestellt werden, dass eine Zeichenkette eine bestimmte Länge nicht überschreitet. Alle darüber hinausgehenden Validierungen müssen vom Entwickler in der Logik der Anwendung programmiert werden.

1.2 Zielsetzung

In dieser Arbeit sollen Konzepte zur Herstellung von Validität und Konsistenz in Web-Anwendungen erarbeitet und schließlich implementiert werden.

1.3 Vorgehen

Zunächst sollen die für diese Arbeit wichtigen Begriffe ,,Validität“, ,,Konsistenz“ und ,,Web-Anwendung“ geklärt werden. Anschließend soll genauer betrachtet werden, wie Validität und Konsistenz in Desktop-Anwendungen hergestellt wird.

Hiernach sollen Web-Technologien untersucht werden, die zur Konstruktion von Web-Anwendungen verwendet werden. Diese Untersuchung soll zunächst allgemein klären, wie diese Technologien funktionieren. Auf diese Untersuchung aufbauend, soll vorgestellt werden, welche Lösungen für die in dieser Arbeit betrachtete Problematik der Validität und Konsistenz von Web-Anwendungen bereits im Zusammenhang mit den einzelnen Technologien bestehen. Aus diesen Lösungen sollen die guten Konzepte extrahiert werden, um dann als Anforderungen und Basis für eigene Konzeptionen zu dienen.

Die beste der Konzeptionen soll schließlich implementiert werden, um so ihre Realisierbarkeit zu belegen

1.4 Behandlung der Thematik in der Literatur

Hassan und Holt fassen die aktuelle Situation bei der Entwicklung von Web-Anwendungen in [Hassan 02] folgendermaßen zusammen

> ,,Originally developed as a document-sharing platform, the web is still often considered as such. Consequently, the development of web applications is considered to be an authoring problem and not a software engineering problem. [...]
>
> With a very short development cycle, software engineering principles are rarely applied to the development of web applications. [...] The techniques used by web application developers are similar to the ad hoc ones used by their predecessors in the 1960s and 1970s. To aggravate matters, the web community has a high turnover rate with an average employment length that is just over one year.“

Bei der Entwicklung von Web-Anwendungen spielen heutzutage nur selten software-technische Geschichtspunkte eine Rolle. Web-Anwendungen werden schnell entwickelt. Wichtig ist, dass sie möglichst schnell verfügbar sind. Auf die Zeit nach dem Produktivgang einer Web-Anwendung,

in der sie gewartet und an neue Anforderungen angepasst werden muss, wird kein Gedanke verschwendet.

So ist es kein Wunder, dass die meisten Bücher und Artikel über Web-Anwendungen die Technologien beschreiben, die zu ihrer Entwicklung verwendet werden. Nur vereinzelt wird hier auch auf die Gesichtspunkte der Softwaretechnik eingegangen.

Da das Web zunächst nur zur Veröffentlichung von Dokumenten gedacht war, gibt es viele Bücher und Artikel, die sich damit befassen, wie diese Dokumente präsentiert werden sollen und wie zwischen ihnen am besten navigiert werden soll. Viele dieser Bücher betrachten Aspekte der Software-Ergonomie (siehe zum Beispiel [Nielsen 99] und [Nielsen 02]). Diese Art von Literatur ist auch für Entwickler von Web-Anwendungen interessant, denn sie gibt Richtlinien für die Entwicklung der Präsentation[10] einer Web-Anwendung an, deren Beachtung zu besseren Benutzungsmodellen in Web-Anwendungen führen kann

1.5 Übersicht

In Kapitel 2 „Konzeptionelle Grundlagen“ werden die Begriffe „Konsistenz“, „Validität“ und „Web-Anwendung“ eingeführt. Außerdem wird die Unterstützung zur Herstellung von Konsistenz und Validität in Desktop-Anwendungen untersucht.

In Kapitel 3 „Web-Technologien und deren Unterstützung für die Herstellung von . . . “ werden einige Web-Technologien und deren Unterstützung für die Herstellung von Konsistenz und Validität in Web-Anwendungen vorgestellt.

In Kapitel 4 „Designalternativen“ werden die in Kapitel 3 aufgestellten Anforderungen an eine Lösung der in dieser Arbeit behandelten Probleme wiederholt und danach die Designalternativen für eine Lösung vorgestellt

In Kapitel 5 „Realisierung“ wird die Realisierung der Lösung vor, die in Kapitel 4 als die tauglichste befunden wurde, vorgestellt.

In Kapitel 6 „Zusammenfassung und Ausblick“ wird eine Gesamtzusammenfassung der Arbeit und ein Ausblick in die Zukunft des in dieser Arbeit behandelten Bereichs gegeben.

[10] Die Präsentation einer Anwendung ist der Teil von ihr, den der Benutzer sieht. Häufig wird dieser Teil auch Benutzungsschnittstelle genannt. Präsentation ist jedoch treffender und auf den Punkt gebracht. Die Präsentation einer Anwendung sollte keine Funktionalität enthalten und selbst die Handhabung dieser Präsentation sollte von ihr getrennt sein(Siehe das Entwurfsmuster „Trennung von Handhabung und Präsentation in [Züllighoven 98]“)

2 Konzeptionelle Grundlagen

> *Condense fact from the vapor of nuance*
>
> — Neal Stephenson : *Snow Crash*

In diesem Kapitel werden zunächst die Begriffe „Validität“, „Konsistenz“ und „Web-Anwendung“ geklärt. Diese Begriffe nehmen eine zentrale Rolle in dieser Arbeit ein, weshalb es wichtig ist, dass der Autor und die Leser ein gemeinsames Verständnis dieser Begriffe teilen.

Dieser Begriffsklärung folgt eine Betrachtung der Thematik „Herstellung und Sicherstellung von Konsistenz und Validität“ bei Desktop-Anwendungen. Dabei wird das JWAM-Formularwesen, das bereits in der Einleitung erwähnt wurde, näher betrachtet.

2.1 Konsistenz und Validität

Der erste Begriff, der hier geklärt werden soll, ist „Konsistenz“. Dieser Begriff wird in der Literatur sehr häufig benutzt und nicht überall bedeutet er dasselbe, weshalb er einer Definition für diese Arbeit bedarf.

Das „Deutsches Wörterbuch“ von Wahrig (siehe [Wahrig 00]) definiert „Konsistenz“ folgendermaßen

> „**Konsistenz** Beschaffenheit (eines Stoffes) hinsichtlich der Struktur; Beständigkeit (eines Stoffes) hinsichtlich Formveränderungen; Widerspruchsfreiheit (von Theorien)“

Und das Adjektiv „konsistent“ wird im gleichen Werk wie folgt definiert

> „**konsistent** dicht, fest, dauerhaft, haltbar, zäh; in sich lückenlos und widerspruchsfrei; Gegensatz inkonsistent [lat. consistens ‚standhaltend‘, Partizip Präsens zu consitere ‚sich hinstellen, standhalten‘]“

Etwas ist also konsistent, wenn es sich nicht jeden Augenblick verändert, wenn es auf eine Weise beständig ist.

Konsistenz von Anwendungen bedeutet, dass Benutzer etwas, das sie an einer Stelle einer Anwendung gelernt haben, auch an anderen Stellen einer Anwendung wieder verwenden können. Es bedeutet, dass gleiche Teilaufgaben auch in der ganzen Anwendung auf die gleiche Art erledigt werden können. Man sagt auch, dass Anwendungen, die diesem Kriterium entsprechen, ein konsistentes Benutzungsmodell haben.

Die kleinste Teilaufgabe, die ein Benutzer in einer Anwendung erledigt, ist das Eingeben von Daten. Ein gutes Beispiel, das bereits in Kapitel 1 auftaucht, ist die Eingabe eines Wochentages. Wenn der Benutzer an jeder Stelle einer Anwendung, an der die Eingabe eines Wochentages notwendig ist, das gleiche Widget präsentiert bekommt, kann er diese Aufgabe sehr schnell erledigen. Nach einiger Zeit weiß er, wie das geht. Es wird zur **Routine** für ihn.

Jef Raskin schreibt dazu in seinem Buch „The Humane Interface“ (siehe [Raskin 00])

> „When you perform a task repeatedly, it tends to become easier to do. Juggling, table tennis, and playing piano are everyday examples in my life; they all seemed impossible when I first attempted them. Walking is a more widely practiced example. [...]

> Many of the problems that make products difficult and unpleasant to use are caused by human-machine design that fails to take into account the helpful and injurious properties of habit formation“

Die Bildung von **Gewohnheiten** ist also ein wichtiges Mittel zur Vergrößerung der Benutzungsqualität einer Anwendung.
Ebenfalls [Wahrig 00] definiert ,,Validität“

> ,,**Validität** Gültigkeit, Rechtskraft; (gehobene) (wissenschaftliche) Zuverlässigkeit“

Das Adjektiv ,,valid“ im gleichen Werk

> ,,**valid** kräftig; rechtskräftig [lat. validus; zu valere ‚stark sein‘]“

Bei Validität geht es nun darum, dass erfasste Daten gültig sind. Benutzer sind Menschen, und Menschen machen Fehler. Vor allem Fehler bei der Eingabe sind keine Seltenheit. Es ist zum Beispiel ein Leichtes, statt eines Kommas einen Punkt einzugeben, da die entsprechenden Tasten auf der Tastatur nebeneinander liegen. Anwendungen, die die Eingaben auf ihre Stimmigkeit, ihre Gültigkeit, prüfen und dies nicht ihren Benutzern überlassen, sind robuster. Fehler die gleich an der Fehlerquelle aufgedeckt werden, können schneller und sicherer bereinigt werden. Werden Fehler erst im Nachhinein gefunden, ist das Kind meistens schon in den Brunnen gefallen, wie ein Volksspruch sagt. Stellt eine Anwendung die Validität der Daten nicht selbst sicher, so müssen diese ,,per Hand“ im Nachhinein und wiederholt geprüft werden.
Validitätsprüfung kann auf verschiedenen Ebenen erfolgen. Abbildung 2.1 zeigt eine Einteilung der Ebenen bei der Validitätsprüfung.

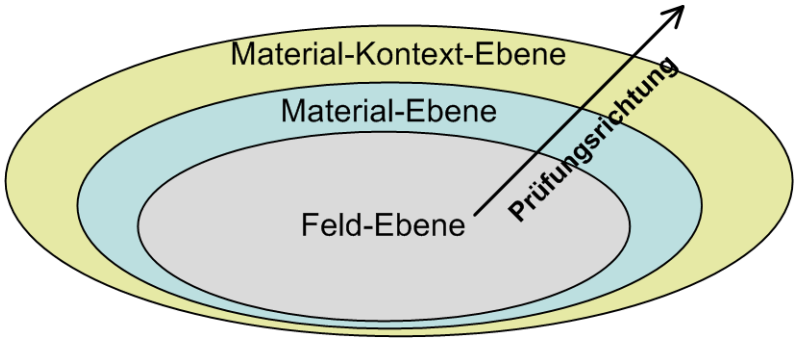

Abbildung 2.1 Validitätsebenen

Eine Prüfung fängt immer auf der **Feld-Ebene** an. Hier werden alle Daten für sich geprüft. Das ist die Ebene, auf der geprüft wird, ob die Daten, die der Benutzer eingegeben hat, gültige Fachwerte sind. Zum Beispiel wird hier geprüft, ob eine Eingabe eine valide Repräsentation eines Datums ist. Die Ebene, die nach der Feld-Ebene kommt, ist die **Material-Ebene**. Hier könnte zum Beispiel bei einer Kalender-Anwendung, in der gerade ein neuer Termin erfasst wird, geprüft werden, dass der Anfangszeitpunkt vor dem Endzeitpunkt des Termins liegt. Auf der **Material-Kontext-Ebene** schließlich wird geprüft, ob das Material in Beziehung zu anderen Materialien valide Daten enthält. Ist zum Beispiel die Bedingung für einen validen Termin, dass er sich nicht mit einem anderen überschneidet, so würde auf dieser Ebene geprüft werden, ob in der Zeit, die für diesen Termin

festgelegt wurde, schon andere Termine eingetragen sind (Siehe [Züllighoven 98][11] [12] für ein Beispiel aus dem Bankenbereich).

2.2 Web-Anwendung

Beier und Vaughan erklären den Begriff ,,Web-Anwendung“ in [Beier 03] folgendermaßen

> ,,Web applications are delivered via a browser that is dominated by different metaphors (e.g. page-centric) than desktop applications (e.g. windows and menus).“

Hassan und Holt geben in [Hassan 02] folgende Definition

> ,,A web application is a software whose functionality is delivered through the web.“

Die Funktionalität einer Web-Anwendung wird also über das Web zur Verfügung gestellt. Der Benutzer verwendet eine Anwendung – genannt Browser –, um auf Web-Anwendungen zuzugreifen. Bei einer Web-Anwendung spielt der Begriff der ,,Seite“ eine wichtige Rolle. Der Benutzer bekommt nach jeder Anfrage, die er durch einen Klick auf einen Knopf oder Verweis auslöst, eine neue Seite angezeigt. Diese Seite kann der vorigen Seite sehr ähnlich sein. Es könnte sein, dass sich im Gegensatz zu der vorigen Seite nur einige Informationen geändert haben. Trotzdem ist es immer eine neue Seite, die der Browser anzeigt. Es werden nicht nur einzelne Widgets aktualisiert.

Die obigen Definitionen gehen für diese Arbeit zu wenig in die Tiefe, weshalb nun eine etwas vertiefende Betrachtung erfolgen soll.

Das Web ist ein Teil des Internets[13]. Es liegt also nahe anzunehmen, dass nur Anwendungen, die über das Internet zugegriffen werden, Web-Anwendungen sein können. Web-Anwendungen können aber auch in einem Intranet[14] existieren.

Die Eigenschaften, die eine Web-Anwendung ausmachen sind

- Die Anwendung befindet sich auf einem entfernten Rechner.
- Zum Zugriff auf die Anwendung wird ein Browser verwendet.
- Auf den Rechner des Benutzers wird kein Teil der Anwendungslogik übertragen. Der Browser lädt jeweils nur Teile der Präsentation der Anwendung vom entfernten Rechner und zeigt diese dem Benutzer an.

[11] Hier wird der Begriff ,,Konsistenz“ benutzt. In der Literatur findet häufig eine Vermischung der Begriffe ,,Konsistenz“ und ,,Validität“ statt

[12] Diese Einteilung der Ebenen stammt von mir selbst. Sie ist von [Züllighoven 98] inspiriert, stammt jedoch nicht aus diesem Werk. Dort werden diese drei Ebenen zwar anhand eines Beispiels aus dem Bankenbereich beschrieben, aber nicht auf diese Weise explizit gemacht.

[13] Das Internet ist der Nachfolger des ARPANET, eines Netzwerks von Rechnern, das zum Austausch von Daten zwischen Wissenschaftlern vom US-amerikanischen Militär geschaffen wurde. Die Erklärung, wie das Internet funktioniert, würde den Rahmen dieser Arbeit sprengen. In der Kürze sei gesagt, dass mehrere Rechner über Glasfaserkabel, Telefonleitungen oder auf andere Art miteinander verbunden sind und unter Verwendung einiger standardisierter Protokolle miteinander kommunizieren. Arbeitet nun ein Benutzer an einem dieser Rechner, so kann er über dieses Netzwerk mit allen Benutzern der anderen Rechner kommunizieren oder auf Daten und Programme auf den anderen Rechnern im Netz zugreifen. Ein Buch, das dieses Thema und die dabei benutzten Protokolle zur Genüge behandelt, ist [Stevens 94]

[14] Ein Intranet ist eine kleine Variante des Internets. In einem Intranet sind normalerweise nur die Rechner einer Firma verbunden

Eine Frage, die sich nun stellt, ist jene, ob Java-Applets zu einer Web-Anwendung gehören können oder gar selbst Web-Anwendungen sind. Dass Java-Applets selbst Web-Anwendungen sind, kann verneint werden. Applets werden immer vollständig auf den Rechner des Benutzers geladen. Es gibt keine Server-Komponente, die auf dem entfernten Rechner verbleibt. Alle Funktionalität wird auf dem Rechner des Benutzers ausgeführt. Manche Applets greifen auf entfernte Rechner und die dort installierten Anwendungen zu. Aber genauso, wie ein Browser, der auf entfernte Web-Anwendungen zugreift, dadurch nicht selbst zu einer wird, wird auch ein Applet dadurch nicht zu einer Web-Anwendung.

Eine andere Art von Anwendungen sind die Thin-Client-Anwendungen (siehe auch [ulc-website] und [Bohlmann 00]). Diese Anwendungen werden auf dem Server ausgeführt, bieten dem Benutzer jedoch eine Präsentation, die der Präsentation einer Desktop-Anwendung gleichkommt. Als eine Möglichkeit des Zugriffs auf diese Anwendungen können relativ kleine Applets in eine Web-Seite eingebettet werden. Ein solches Applet baut dann im Browser die Präsentation der Anwendung auf und benachrichtigt die auf dem Server laufende Anwendung, wenn der Benutzer etwas mit der Präsentation macht (zum Beispiel einen Knopf anklickt oder in ein Eingabefeld etwas eingibt). Diese Anwendungen scheinen also trotz des kleinen Anteils an Anwendungslogik, der in dem Applet vorhanden sein muss, alle bisher aufgezeigten Eigenschaften einer Web-Anwendungen zu haben.

Eine Tatsache, die dagegen spricht, solche Thin-Client-Anwendungen als Web-Anwendungen anzusehen ist, dass bei ihnen der Begriff der „Seite“ keine Rolle spielt. Tatsächlich ist aber die ganze Zeit über eine einzige Seite vorhanden, nämlich die, die das Applet enthält. Der Browser ist an der Kommunikation mit der Anwendung auf dem Server-Rechner nicht beteiligt. Diese obliegt hier einzig und allein dem Applet. Für Thin-Client-Anwendungen ist nicht einmal der Browser selbst notwendig. Der Benutzer kann das, was hier in einem Applet verpackt ist auch als normale Desktop-Anwendung installieren, die dann wie gehabt mit dem Server kommuniziert.

Es soll hier nicht bestritten werden, dass solche Anwendungen sinnvolle Einsatzkontexte haben. Diese Einsatzkontexte unterscheiden sich jedoch meistens von denen der Web-Anwendungen. Web-Anwendungen sollen von überall mit minimalsten Anforderungen benutzt werden können. Solche minimalen Anforderungen sind zum Beispiel, dass der Rechner des Benutzers über ein Rechner-Netzwerk mit dem Server-Rechner verbunden ist und dass ein Browser auf diesem Rechner installiert ist. Sollten Applets in Web-Anwendungen verwendet werden, müsste auf dem Rechner des Benutzers auch eine Java-Laufzeitumgebung installiert sein und die Verbindung des Rechners des Benutzers mit dem Server-Rechner müsste eine nicht zu kleine Bandbreite haben, damit die Applets schnell heruntergeladen werden könnten.

Im Folgenden sollen Web-Anwendungen betrachtet werden, die möglichst wenige Anforderungen an den Rechner des Benutzers stellen. Diese sollen also Web-Anwendungen sein, die keine Applets oder zu Applets vergleichbare Technologien verwenden.

Bisher wurde festgestellt, dass bei Web-Anwendungen die ganze Anwendungslogik auf dem Server-Rechner verbleibt und auf den Rechner des Benutzers nur die Präsentation der Anwendung geladen wird. Das ist die Idealvorstellung von einer Web-Anwendung. Diese Idealvorstellung lässt sich häufig nicht vollständig aufrechterhalten, ohne dass die Benutzbarkeit der Anwendung darunter leidet. Benutzer müssen in Web-Anwendungen häufig Formulare ausfüllen. Wenn sie das Formular abschicken und nach einer kurzen Verzögerung vom Server-Rechner eine neue Seite angezeigt bekommen, die ihnen mitteilt, dass sie an einer Stelle einen Fehler gemacht haben, kommt der Interaktionsfluss schnell ins Stocken. Manchmal ist die Validierung der Eingabe einfach möglich. So könnte mit einem kleinen Teil von Anwendungslogik bereits auf dem Rechner des Benutzers geprüft werden, dass zum Beispiel bei einer Mengenangabe nur Zahlen eingegeben werden. Eine

solche Validierung kann nicht abschließend sein. Es könnte zum Beispiel sein, dass in dem Online-Geschäft, das diese Web-Anwendung betreibt, die eingegebene Menge des Produkts nicht verfügbar ist. Eine solche Vor-Validierung kann jedoch Tippfehler und Missverständnisse abfangen.

Wenn in dieser Arbeit im weiteren Verlauf von Web-Anwendungen gesprochen wird, so sind damit Anwendungen der Art gemeint, wie sie in Abbildung 2.2 skizziert sind.

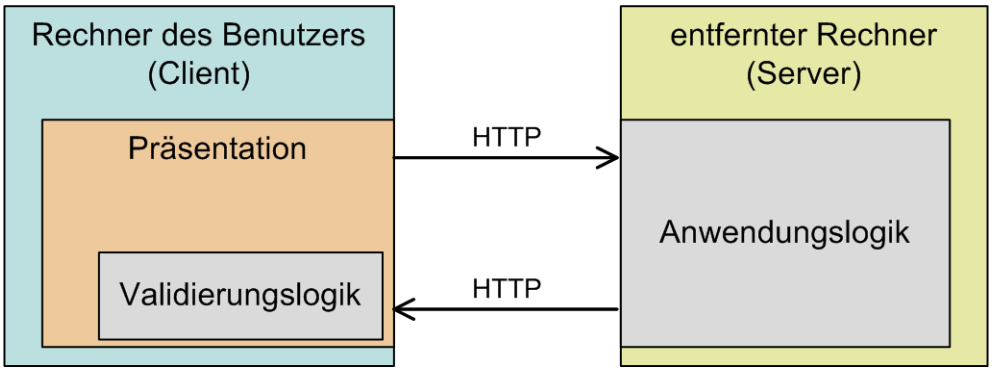

Abbildung 2.2 Schematische Darstellung einer Web-Anwendung

2.3 Benutzungsmodelle für Web-Anwendungen

Web-Anwendungen bieten andere Möglichkeiten als Desktop-Anwendungen. Es gibt hier keine Fenster, keine Menüs, keine Symbolleisten und auch keine Tastaturkürzel, die die Anwendung verwenden kann.

Es gibt zwar einige Versuche diese Dinge auch in Web-Anwendungen anzubieten, aber diese Experimente scheitern meist, weil sie zum Beispiel eine bestimmte Browserversion erfordern und somit von Benutzern, die diese nicht haben, nicht benutzt werden können. Web-Anwendungen sind Seiten-orientiert (siehe Abschnitt 2.2). Es wird immer nur eine Seite, nur ein Teil der Präsentation der Web-Anwendung, angezeigt. Das Laden neuer Seiten geschieht mit einer gewissen Verzögerung, weshalb unnötiges Laden neuer Seiten möglichst vermieden werden soll.

Da Web-Anwendungen so verschieden von Desktop-Anwendungen sind, benötigen sie auch andere Benutzungsmodelle.

Software-Ergonomen beschäftigen sich schon seit einigen Jahren damit, wie Web-Sites aussehen sollen, um ein effizientes Arbeiten mit ihnen zu ermöglichen. Ein wichtiger Punkt ist hierbei auch die Navigation zwischen den einzelnen Seiten einer Web-Site. Das Buch „Designing Web Usability : The Practice of Simplicity“ von Jakob Nielsen (siehe [Nielsen 99]) ist mittlerweile zu einem Standardwerk in diesem Bereich geworden.

Da Web-Anwendungen von außen betrachtet nicht anders aussehen als Web-Sites, die lediglich aus statischen Seiten bestehen, gilt vieles von dem, was für gute Web-Sites gilt, auch für gute Web-Anwendungen.

Es würde den Rahmen dieser Arbeit sprengen, hier diese Themen der Software-Ergonomie aufzugreifen.

Es bleibt festzustellen, dass Web-Anwendungen keine erweiterten Desktop-Anwendungen sind, die im Web laufen. Web-Anwendungen bieten sich überall dort an, wo Informationen verwaltet werden sollen[15]. Sie bieten sich fast nie für Anwendungen an, die für kreative Arbeit verwendet werden,

[15] Dies folgt aus der Betrachtung zurzeit bekannter Web-Anwendungen wie Online-Kaufhäuser, Online-Banking, Web-Mail, Wörterbücher, Zeiterfassungssysteme etc.

denn sie bieten nur einfache Interaktionen an. So wird sich ein Zeichenprogramm nur schwerlich als Web-Anwendung entwickeln lassen. Ja selbst Anwendungen, bei denen hauptsächlich Texte geschrieben und verändert werden – Office-Systeme und sogar Entwicklungsumgebungen – lassen sich nicht mit derselben Fülle an Möglichkeiten als Web-Anwendungen entwickeln. Web-Anwendungen müssen sich mit einer einfachen, meist gradlinigen, Struktur begnügen.

2.4 Konsistenz und Validität in Desktop-Anwendungen

Im Folgenden soll die Herstellung von Konsistenz und Validität in Desktop-Anwendungen betrachtet werden. Um daraus Schlüsse und Ideen für das gleiche Thema im Kontext der Web-Anwendungen zu ziehen, sollen die Desktop-Anwendungen, die hier betrachtet werden, solche sein, die sich mit der Erstellung, Ausfüllung und Verwaltung von Formularen beschäftigen.

Formulare kommen trotz der großen Möglichkeiten in Desktop-Anwendungen auch dort vor. Ein Formular als solches ist den meisten Benutzern aus seiner **Papierform** bekannt (Schließlich muss jeder arbeitende Mensch einmal im Jahr seine Einkommensteuererklärung machen und dabei zuweilen einen ganzen Berg von Formularen ausfüllen). Durch diesen hohen Bekanntheitsgrad des einfachen Konzeptes, das hinter Formularen steckt, ist eine Transition zu Formularen in einer Anwendung auf einem Rechner meistens leicht vollziehbar.

Formulare haben Felder, in die ein Ausfüller alle möglichen Arten von Daten eintragen kann – Geburtsdaten, Adressen, Geldbeträge etc. –, Kästchen, die angekreuzt werden können und schließlich auch Beschriftungen, die angeben, was all diese Felder und Kästchen bedeuten. Auch Formulare in Anwendungen auf Rechnern haben diese Elemente und erweitern diese durch spezielle auf dem Rechner verfügbare Möglichkeiten der Eingabe – Auswahllisten verschiedener Art sind hierfür ein Beispiel[16].

Der Designer eines guten Formulars achtet auf Konsistenz im Formular. Wenn es im Formular bei einer Frage, auf die mit „Ja“ oder „Nein“ geantwortet werden kann, zwei entsprechende Ankreuzkästchen gibt, wird es bei der nächsten Frage derselben Art nicht plötzlich ein Feld geben, in das dann per Hand „Ja“ oder „Nein“ reingeschrieben werden muss. Wie in Abschnitt 2.1 gesagt, ist die Bildung von **Gewohnheiten** wichtig für ein effizientes Arbeiten. Das gilt sowohl für Arbeiten, die mit einem Bleistift auf einem Papierformular erledigt werden als auch für Tätigkeiten am Rechner.

So muss auch beim Entwurf von Formularen in Anwendungen auf die Herstellung von Konsistenz geachtet werden. Beim Entwurf eines Papierformulars muss der Designer selbst darauf achten, dass es konsistent ist. Beim Entwurf von Formularen in Software kann dies erleichtert werden.

Ein Beispiel für Software-Formulare, die die Herstellung von Konsistenz unterstützen, ist das Formularwesen[17], das sich im JWAM-Rahmenwerk befindet. Während andere Formularsysteme eine eigene Sprache zur Definition der Formulare anbieten – Das in [Girgensohn 95] beschriebene System ist ein solches –, erfolgt hier die Definition eines Formulars in einer Java-Klasse, die von einer Rahmenwerksklasse `Form` erbt.

Die Definition eines Formulars wird normalerweise im Konstruktor der jeweiligen Formularklasse vorgenommen, so dass jedes Objekt dieser Klasse auch gleich weiß, welche Formularfelder es hat.

Hier nun ein Beispiel für ein einfaches Formular

[16] [Girgensohn 95] stellt ein System vor, das ausschließlich zum Entwurf und zur Ausfüllung von solchen Softwareformularen entwickelt wurde

[17] Eine vertiefende Beschreibung des Formularwesens in JWAM sowie der dahinter stehenden Konzepte ist in Olaf Thiels Diplomarbeit (siehe [Thiel 02]) zu finden

```
public class Meeting extends Form
{
  public Meeting ()
  {
    super("Meeting", "Meeting");
    addElement(new FormField(StringDV.Factory.instance(),
               "shortdesc", "Short Description"));
    addElement(new FormField(StringDV.Factory.instance(),
               "location", "Location"));
    addElement(new FormField(TextDV.Factory.instance(),
               "longdesc", "Long Description"));
    addElement(new FormField(DateDV.Factory.instance(),
               "date", "Date"));
    addElement(new FormField(TimeDV.Factory.instance(),
               "starttime", "Start Time"));
    addElement(new FormField(TimeDV.Factory.instance(),
               "endtime", "End Time"));
    addElement(new FormField(UsernameDV.Factory.instance(),
               "creator", "Creator"));
    addElement(new FormField(UsernameDV.Factory.instance(),
               "changer", "Changer"));
    addElement(new FormField(DateDV.Factory.instance(),
               "lastchange", "Last Change Date"));
    addElement(new FormField(BooleanDV.Factory.instance(),
               "tentative", "Tentative"));
    addElement(new FormField(ChangePrivilegedDV.Factory.instance(),
               "changeprivileged", "May be changed by"));
  }
}
```

Hier wird also die aus der Oberklasse geerbte Operation `addElement` verwendet, um dem Formular einige Objekte vom Typ `FormField` hinzuzufügen. Diesen werden bei ihrer Konstruktion folgende Parameter übergeben

- die Fachwertfabrik, die benutzt wird, um aus den Eingaben des Benutzers **Fachwerte** zu machen
- einen Namen, über den die Formularfelder im Formular gefunden werden können
- eine Beschreibung, die neben dem Formularfeld als Information für den Benutzer angezeigt wird

Wie bereits in der Einleitung (siehe Abschnitt 1) erwähnt, sind Fachwerte unveränderliche Objekte und simulieren somit die Eigenschaften von Werten wie Geldbeträgen, Postleitzahlen und vielen mehr (Für mehr Informationen zu Fachwerten siehe [Müller 99]).

Eine für das Formularwesen wichtige Eigenschaft des **JWAM-Fachwert-Rahmenwerkes** ist die Möglichkeit, an der jeweiligen Fabrik zu prüfen, ob zum Beispiel eine Zeichenkette eine valide Repräsentation für einen Fachwert des Typs von Fachwerten ist, die von der Fabrik hergestellt werden.

Spezielle Fachwert-Widgets, die für die Eingabe von Fachwerten entwickelt wurden, nutzen diese Eigenschaft und prüfen während der Eingabe, ob das, was bis zum Zeitpunkt der Prüfung eingegeben wurde, valide ist.

Im Formularwesen von JWAM gibt es ein generisches Werkzeug – den **FormEditor** –, mit dem Formulare aller Art – sofern sie Objekte der Klasse `Form` oder einer ihrer Subklassen sind – bearbeitet werden können. Über einen generischen Mechanismus in der Klasse `Form` holt sich dieses Werkzeug nacheinander alle Elemente des Formulars. Für jedes Formularfeld liest es die Beschreibung und zeigt diese an der Oberfläche an. Auf die Beschreibung muss nun das Widget für das Formularfeld folgen. Hierzu holt sich das Werkzeug bei einer Klasse `PFFactory` ein dort für den jeweiligen Fachwert-Typ registriertes Widget und integriert dieses in die Präsentation.

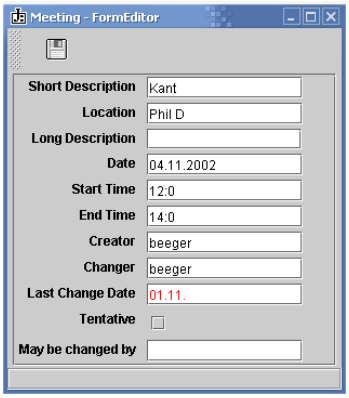

Abbildung 2.3 Der generische FormEditor

Dadurch, dass für einen Typ von Fachwerten immer dieselbe Art von Widgets verwendet wird, ist auch Konsistenz sichergestellt.

2.5 Zusammenfassung

In diesem Kapitel wurden eingangs die Begriffe „Konsistenz“ und „Validität“ in dem Sinne geklärt, wie sie in dieser Arbeit verwendet werden.

Konsistenz hat etwas mit einem konsistenten Benutzungsmodell bei einer Anwendung zu tun. Gleiche Aufgaben können auf die gleiche Art erledigt werden. So werden für die Eingabe von Daten gleicher Art auch immer Widgets gleicher Art verwendet.

Validität hat mit den **Daten** selbst zu tun. Sind diese valide, dann heißt das, dass sie fehlerfrei und gültig sind. Validität kann auf drei **Ebenen** betrachtet werden. Auf der ersten, der Feld-Ebene, wird zum Beispiel geprüft, ob das, was der Benutzer an einer Stelle eingegeben hat, ein valides Datum oder eine valide Geldbetragsangabe ist. Auf der zweiten, der Material-Ebene, wird zum Beispiel geprüft, ob der Starttermin eines Treffens vor einem Endtermin liegt. Auf der dritten Ebene, der Material-Kontext-Ebene, wird schließlich zum Beispiel geprüft, ob ein Geldbetrag, der überwiesen werden soll, auf dem zu belastenden Konto gedeckt ist.

Web-Anwendungen sind Anwendungen, die auf einem **entfernten Rechner** laufen. Der Benutzer benutzt sie über das Internet oder ein Intranet. Um auf eine Web-Anwendung zuzugreifen, verwendet er ein Programm, das **Web-Browser** genannt wird. Die Web-Anwendung beantwortet jede Anfrage des Benutzers mit der Rückgabe eines Teils der Präsentation der Web-Anwendung, der dem Benutzer weitere Möglichkeiten bietet, mit der Web-Anwendung zu interagieren. Wichtig ist hierbei, dass auf den Rechner des Benutzers nur die Präsentation der Anwendung übertragen wird. Die Anwendungslogik verbleibt auf dem entfernten Rechner.

Bei der Betrachtung von Benutzungsmodellen für Web-Anwendungen wurde festgestellt, dass diese sich von den Benutzungsmodellen bei Desktop-Anwendungen unterscheiden. Web-Anwendungen stellen nicht die gleichen Möglichkeiten zur Verfügung, wie sie bei Desktop-Anwendungen vorhangen sind. Web-Anwendungen müssen sich mit einfachen und gradlinigeren Strukturen zufrieden geben. Da Web-Anwendungen von außen nicht anders aussehen als Web-Sites gelten auch für sie die für Web-Sites aufgestellten Richtlinien.

Zum Abschluss des Kapitels wurde die Unterstützung für die Herstellung von Konsistenz und Validität auf den Desktopsystemen betrachtet. Als eine mit den Web-Anwendungen vergleichbare Klasse von Desktop-Anwendungen wurden die **Formular-basierten** herausgestellt und hier das JWAM-Formularwesen vorgestellt, das einige Unterstützung im erwähnten Bereich bietet.

3 Web-Technologien und deren Unterstützung für die Herstellung von Konsistenz und Validität

> *When Ronald Reagan was a radio announcer, he used to call baseball games that he did not physically attend by reading the terse descriptions that trickled in over the telegraph wire and were printed out on a paper tape. He would sit there, all by himself in a padded room with a microphone, and the paper tape would creep out of the machine and crawl over the palm of his hand printed with cryptic abbreviations. If the count went to three and two, Reagan would describe the scene as he saw it in his mind's eye: 'The brawne left hander steps out of the batter's box to wipe the sweat from his brow. The umpire steps forward to sweep the dirt from the home plate,' and so on. When the cryptogram on the paper tape announced a base hit, he would whack the edge of the table with a pencil, creating a little sound effect, and describe the arc of the ball as if he could actually see it. His listeners, many of whom presumably thought that Reagan was actually at the ballpark watching the game, would reconstruct the scene in their minds according to his description.*
> *This is exactly how the World Wide Web works: the HTML files are the pithy description on the paper tape, and your web browser is Ronald Reagan. The same is true of graphical user interfaces in general.*
>
> — Neal Stephenson : *In the beginning . . . was the command line*

In diesem Kapitel werden einige Web-Technologien und deren Unterstützung für die Herstellung von Konsistenz und Validität in Web-Anwendungen vorstellt.

3.1 HTML

HTML[18] ist eine Dokumentenbeschreibungssprache. Sie wurde ursprünglich zum Publizieren **wissenschaftlicher Dokumente** im Internet entwickelt. Mittlerweile wird HTML auch zum Publizieren von Werbung, Romanen, Selbstdarstellungen und vielem mehr benutzt. Ein immer stärker anwachsendes Einsatzgebiet für HTML ist die Beschreibung der Präsentation von Web-Anwendungen.

HTML hat ihren Ursprung in der **SGML**[19], einer mächtigeren aber auch komplizierteren Dokumentenbeschreibungssprache.

HTML wird zur Beschreibung von Dokumenten verwendet. Diese Beschreibung enthält sowohl den Inhalt des Dokuments, also den tatsächlichen Text und Verweise auf weitere anzuzeigende oder abzuspielende Medien wie Bilder und Animationen, als auch eine Beschreibung, wie dieser Inhalt anzuordnen ist.

Da HTML zunächst nur zum Publizieren von wissenschaftlichen Dokumenten gedacht war, genügte es, dass die ersten Versionen von HTML nur die logische Anordnung der Teile eines Dokumentes erlaubten. So war es in den Anfängen nur möglichen zu definieren, was als normaler Text und was als Überschriften angezeigt werden sollte. Es war nicht möglich zu definieren, welche Schriftart verwendet werden sollte und in welcher Größe.

Mit der Zunahme der Einsatzgebiete für HTML nahmen auch die Möglichkeiten, das tatsächliche Aussehen eines Dokumentes vorherzubestimmen, zu.

Bei HTML werden in den Text so genannte **Tags** eingefügt, die die logischen Strukturmerkmale und in zunehmendem Maße auch Layoutmerkmale definieren.

Ein einfaches HTML-Dokument sieht folgendermaßen aus

[18] HTML = Hypertext Markup Language
[19] SGML = Standard Generalized Markup Language

```
<html>
  <head>
    <title>Einfaches Dokument</title>
  </head>
  <body>
    <h1>Große Überschrift</h1>
    <h6>Ziemlich kleine Überschrift</h6>

    Ganz normaler Text. Bla bla bla bla bla bla
    bla bla bla 42 42 42 42 42 42 42 42 42 42 42
    42 42.

    <table>
      <tr>
        <td>Zeile 1 / Spalte 1</td>
        <td>Zeile 1 / Spalte 2</td>
        <td>Zeile 1 / Spalte 3</td>
      </tr>
      <tr>
        <td>Zeile 2 / Spalte 1</td>
        <td>Zeile 2 / Spalte 2</td>
        <td>Zeile 2 / Spalte 3</td>
      </tr>
    </table>
  </body>
</html>
```

Die Tags sind hierbei jene Textstellen, die zwischen spitzen Klammern stehen, also `<html>`, `<table>` und so weiter. Fast jedes öffnende Tag – zum Beispiel `<table>` – hat auch ein dazugehörendes schließendes Tag – zum Beispiel `</table>`. Tags können geschachtelt werden. So gibt es in dem Tag `<table>`[20] des Beispiel-HTML-Dokumentes zwei Tags `<tr>`[21]. Diese bestehen jeweils aus drei Tags `<td>`[22]. HTML ist nicht sehr strikt. An vielen Stellen kann das schließende Tag weggelassen werden, und die Seite wird vom Browser immer noch korrekt angezeigt. Da der Code eines HTML-Dokumentes, das ein schließendes Tag für jedes öffnende Tag enthält, besser zu lesen ist, und die nächste Version von HTML – **XHTML** – diese Freiheiten nicht mehr zulässt, gehört es aber zum guten Ton, auch immer ein schließendes Tag einzufügen. Wie so ein einfaches HTML-Dokument in einem Browser angezeigt wird, ist in Abbildung 3.1 zu sehen.

Wie in Abbildung 3.1 zu sehen ist, werden die Zeilenumbrüche, die im Text eingefügt wurden, ignoriert. Zeilenumbrüche werden wie eine Form von Leerzeichen interpretiert und markieren nur das Ende eines Wortes und den Beginn eines anderen. In der Darstellung werden Zeilenumbrüche an den Stellen eingefügt, an denen sie gebraucht werden. Dieser Umstand wird in Abbildung 3.2 verdeutlicht. Hier ist das gleiche Dokument in einem kleineren Browser zu sehen. Weil der Platz hier geringer ist, hat der Browser an den nötigen Stellen Zeilenumbrüche in der Darstellung eingefügt.

Dies ist nun ein kleines HTML-Dokument, das einige Grundelemente der Dokumentenbeschreibungssprache und deren Verwendung zeigt. Es würde den Rahmen dieser Arbeit sprengen, alle

[20] engl. table = Tabelle
[21] tr = table row. engl. table row = Tabellenzeile
[22] td = table data. engl table data = Tabellendaten. In diesen Tag kommen schließlich die Daten der Tabelle. Ein td stellt also eine Zelle einer Tabellenzeile dar.

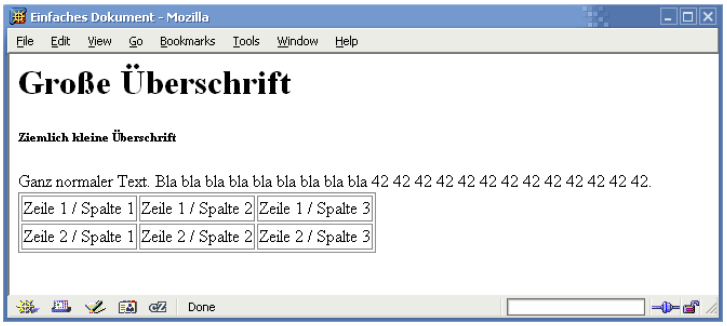

Abbildung 3.1 Das Aussehen des einfachen HTML-Dokumentes in einem Browser

Abbildung 3.2 Das Aussehen des einfachen HTML-Dokumentes in einem kleineren Browser

möglichen Elemente eines HTML-Dokumentes und deren Verwendung zu zeigen. Eine vollständige Übersicht zu HTML ist in ,,SelfHTML“ von Stefan Münz (siehe [Münz 01]) zu finden.

Auch die Website des World Wide Web Consortiums (siehe [w3c-website]), das HTML und andere Web-Technologien weiterentwickelt, ist eine gute Referenz für HTML.

3.2 HTML-Formulare

Von besonderem Interesse für diese Arbeit sind **HTML-Formulare**, weswegen dieses Thema hier näher beleuchtet wird.

Während die bisher vorgestellten Bestandteile HTMLs nur die Möglichkeit bieten, Text strukturiert anzuzeigen, stellen HTML-Formulare eine Möglichkeit des Datenaustauschs zwischen dem Leser der Texte und dem Server-Rechner, von dem die Dokumente angefordert werden, dar. Dies wird zum Beispiel genutzt, um dem Benutzer zu ermöglichen, in einem Archiv von Dokumenten, die für ihn relevanten zu suchen. Ein HTML-Formular hat ein für die Benutzer von **Werkzeugen** mit graphischer Benutzungsoberfläche vertrautes Bild. Auch hier gibt es Eingabefelder, Listen, Knöpfe und anderes von den Werkzeugen bekanntes. Die Interaktion mit HTML-Formularen ist jedoch um einiges eingeschränkter als die mit Werkzeugen. Hier können nur Daten eingetragen und an den Server verschickt werden, der dann als Reaktion eine neues HTML-Dokument zur Anzeige stellt. Die neue Seite kann auch wieder Formulare enthalten.

Eine einfache HTML-Seite mit Formular sieht wie folgt aus

```
<html>
  <head>
   <title>Einfaches Dokument mit Formular</title>
  </head>
  <body>
    <form action="servlet/LoginServlet" method="get">
      <table align="center">
        <tr>
          <td>Benutzername</td>
          <td><input name="username" type="text"/></td>
        </tr>
        <tr>
          <td>Passwort</td>
          <td><input name="password" type="password"/></td>
        </tr>
        <tr>
          <td>Domäne</td>
          <td>
            <select name="domain">
              <option>FBISWT</option>
              <option>DOM-WPS</option>
            </select>
          </td>
        </tr>
        <tr>
          <td colspan="2" align="middle">
            <input name="LoginAction" type="submit" value="Login"/>
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

Diese wird in einem Browser wie in Abbildung 3.3 zu sehen, angezeigt.

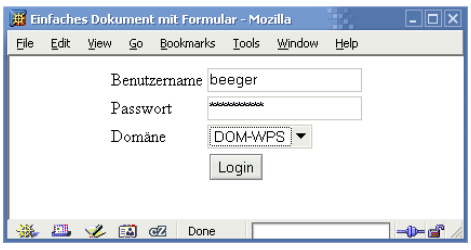

Abbildung 3.3 Das Aussehen der einfachen HTML-Seite mit Formular in einem Browser

Ein Formular wird mit einem `form`-Tag eingeleitet. Dieses enthält ein Attribut `action`, das angibt, wohin das Formular geschickt werden soll, wenn der Benutzer auf den in diesem Beispiel mit „Login“ beschrifteten Knopf klickt. Dieses Verschicken kann auf unterschiedliche Arten erfolgen. In diesem Beispiel wird „get“ hinter dem Attribut `method` angegeben.

Die `input`-Tags sowie das `select`-Tag stehen für Widgets, die im Browser an deren Stelle angezeigt werden. An Stelle eines `input`-Tags, bei dem das Attribut `type` auf `text` gesetzt ist, wird der Browser ein Texteingabefeld anzeigen. Für ein `select`-Tag wird er eine ausklappbare Auswahlliste anzeigen.

3.3 Bereits bestehende Validierungsmöglichkeiten für HTML-Formulare

Da HTML-Formulare hauptsächlich zur **Datenerfassung** benutzt werden, stellt sich die Frage nach der **Validierung** der eingegebenen Daten. Eingabefehler müssen möglichst frühzeitig abgefangen werden, damit sie nicht zu einem inkonsistenten Datenbestand führen. Der Benutzer muss eine Möglichkeit haben, seine Fehleingaben zu korrigieren (Siehe hierzu auch Abschnitt 2.1).

Hier gibt es zwei Varianten der Validierung[23].

3.3.1 Verzögerte Validierung

Bei der **verzögerten Validierung** werden die Daten an den Server geschickt, der sie dann validiert. Wie diese Validierung realisiert wird, ist von Fall zu Fall unterschiedlich. Oft werden in C oder Perl geschriebene **CGI[24]-Programme** verwendet, die dies erledigen. Wenn nun die Eingabe fehlerhaft ist, gibt es zwei verbreitete Varianten der Reaktion. In vielen Fällen wird als Reaktion auf einen solchen Fehler, eine neue HTML-Seite angezeigt, auf der dem Benutzer mitgeteilt wird, dass er einen Fehler bei der Eingabe gemacht hat. Verbreitet, aber nicht unbedingt selbstverständlich, ist die Angabe der Eingabefelder, die falsch befüllt wurden.

Ein Mangel dieser Art von Reaktion ist, dass der Benutzer den „Zurück“-Knopf seines Web-Browsers benutzen muss, um zurück zum Formular zu gelangen, wo er die Eingabe verbessern kann.

Ein zweiter Mangel wird durch ein Zitat aus Jef Raskins Buch „The Humane Interface“ (siehe [Raskin 00]) deutlich

> „Perceptions do not become automatically memories. Most perceptions are lost after they decay. One implication for interface design of the rapid decay of sense perceptions is that you cannot assume that, because someone has seen or heard a particular message 5 seconds earlier, that person will remember its wording. If that particular wording is important or if there is an important detail – for example, if the message is, ‚Report error type 39-152,‘ with the critical detail being the particular number – either you must keep the message displayed until it is no longer needed (the best strategy), or the user must be able to apply the information immediately – that is, before memory of it decays.“

Der zweite Mangel ist, dass der Benutzer sich die Auflistung der fehlerhaften Einträge merken muss, da er diese Information nach dem Betätigen des „Zurück“-Knopfes nicht mehr sieht. Hat er nur

[23] Die hier gezeigte Aufteilung der Varianten der Validierung beruht auf den Erfahrungen und Beobachtungen des Autors in diesem Bereich

[24] CGI = Common Gateway Interface

ein Feld fehlerhaft eingetragen, kann er diese Information noch im Kopf haben, wenn das Formular erneut angezeigt wird. Bei mehr als einer Fehleingabe ist der Erfolg eher unwahrscheinlich.

Eine bessere Variante ist es, als Reaktion auf eine Fehleingabe das Formular erneut anzuzeigen und die Information über die fehlerhaften Eingaben auf derselben Seite anzuzeigen, auf der sich auch das Formular befindet. Das kann entweder so realisiert werden, dass dieselbe Meldung, die bei der vorigen Variante verwendet wurde, auch hier verwendet wird, oder dass neben jedem fehlerhaften Feld eine entsprechende Meldung erscheint.

Die letzte ist die beste Variante, da der Benutzer alle Felder mit angehängter Meldung nacheinander durchgehen kann und nicht für jedes Feld eine lange Meldung von Neuem lesen muss.

3.3.2 Sofortige Validierung

Bei der **sofortigen Validierung** werden proprietäre Erweiterungen von HTML benutzt, die nicht zum Standard gehören. Eine solche, viel benutzte Erweiterung, ist **JavaScript**[25]. JavaScript bietet Möglichkeiten, die sonst statischen HTML-Seiten zu dynamisieren. So können in JavaScript Reaktionen auf das Betreten und Verlassen eines Eingabefeldes mit dem Cursor programmiert werden. Dies wird zu Validierungszwecken so eingesetzt, dass beim Verlassen eines Eingabefeldes eine Validierungsfunktion aufgerufen wird. Findet diese Funktion heraus, dass die Eingabe falsch ist, wird oft ein neues Fenster geöffnet, in dem eine entsprechende Meldung angezeigt wird. Ein besserer, wenn auch komplizierterer, Einsatz dieser Technologie ist es, dynamisch den Bereich neben der Fehlangabe mit einer Meldung zu versehen. Ein Beispiel zur sofortigen Validierung mit JavaScript ist im Anhang A.1 zu finden.

Eine sofortige Validierung kann unter Umständen sehr komplex werden, wenn alle Daten sehr akkurat validiert werden sollen. Damit wird aber auch die Validierungsfunktion länger und mit ihr die Seite, die länger braucht, um auf den Rechner des Benutzers geladen zu werden. Zu der Zeit, die eine Seite braucht, um heruntergeladen zu werden schreiben Jakob Nielsen und Marie Tahir in ihrem Buch „Hompage Usability“ (siehe [Nielsen 02])

> „**Download time** : At most 10 seconds at the prevalent connection speed for your customers. For modem users, this means a file size of less than 50 KB. Faster is better.“

Es muss also immer zwischen der Genauigkeit der Validierung und der Größe einer HTML-Seite – und damit der benötigten Ladezeit für die Seite – ausbalanciert werden. Ist die Validierungsfunktion sehr genau, so ist das Arbeiten mit dem Formular benutzerfreundlich. Wenn das Laden der Seite jedoch zu lange dauert, kann es sein, dass die Benutzer ungeduldig werden und erst gar nicht auf das Formular warten und zu anderen Web-Sites wechseln, die ihnen denselben Service mit einem möglicherweise weniger komfortablen, aber dafür schneller geladenem Formular anbieten.

3.3.3 Bewertung

Die Validierung der Daten, die ein Benutzer eingibt sollte möglichst früh geschehen, so dass der Benutzer etwaige Fehler korrigieren kann. Die sofortige Validierung ermöglicht die früheste Validierung, die für HTML-Formulare möglich ist. Sie hat jedoch auch ihre Nachteile. Einer der Nachteile ist, dass komplexe Validierungsfunktionen die Seitengröße ansteigen lassen und damit

[25] Es gibt eine von der ECMA (European Computer Manufacturers Association) standardisierte Version von JavaScript, die ECMAScript heißt. Die Unterstützung dieses Standards seitens der Browserhersteller fällt eher spärlich aus. Man konzentriert sich hier lieber auf die eigenen Erweiterungen zu JavaScript

auch die Zeit länger wird, um eine solche Seite auf den Rechner des Benutzers zu laden. Ein weiterer Nachteil ist die Gefahr bei komplexen Validierungsfunktionen spezielle JavaScript-Features zu verwenden, die nur in bestimmten Browser-Versionen verfügbar sind. Hinzu kommt noch, dass komplexe Validierungsfunktionen, die eine 100-prozentige Validierung ermöglichen immer speziell für einen Datentyp neu geschrieben werden müssen.

Hier bietet es sich an, den Teil der sofortigen Validierung durch die Verwendung von regulären Ausdrücken generisch zu halten. Dann ist nur eine Validierungsfunktion nötig, der neben dem Inhalt des Feldes auch der für dieses Feld zu verwendende reguläre Ausdruck übergeben werden muss. Da es mitunter schwierig oder sogar nicht möglich sein kann, einen regulären Ausdruck anzugeben, der garantiert, dass Daten, die zu ihm passend sind wirklich gültig sind, bietet sich eine zweistufige Validierung an. Auf dem Rechner des Benutzers wird eine Vor-Validierung mittels sofortiger Validierung vorgenommen. Bei dieser Validierung werden reguläre Ausdrücke verwendet. Werden Daten bei dieser Validierung als valide erkannt, so besteht eine hohe Chance, dass sie es auch tatsächlich sind. Hiermit sollten die meisten Tippfehler und Fehleingaben abzufangen sein. Auf dem Server erfolgt dann die endgültige Validierung, die dann auch komplexer sein kann als die sofortige Validierung. Es kann dann durchaus vorkommen, dass Daten, die in der Vor-Validierung als valide erkannt wurden, jetzt als invalide erkannt werden. In dem Fall, dass die Daten nun invalide sind, kann die Seite dem Benutzer erneut angezeigt werden, wobei für ihn erkennbar sein muss, welche Eingaben fehlerhaft waren und warum.

Eine Kombination beider Validierungsarten bietet also die Möglichkeit, die Anwendung benutzerfreundlicher zu machen und dabei die Validität der Daten sicherzustellen.

Als eine erste Anforderungen an eine Lösung der Probleme, die Thema dieser Arbeit sind, kann also aufgestellt werden

> Web-Anwendungen sollen bei der Validierung von Daten ein zweistufiges Verfahren einsetzen, das sich aus einer Vor-Validierung auf dem Rechner des Benutzers und einer Haupt-Validierung auf dem Server zusammensetzt.

3.4 Servlet-API und JavaServer Pages

Die Programmierung von CGI-Programmen und -Skripten ist ein aufwendiger Prozess, bei dem ein großes Wissen über **HTTP**[26] notwendig ist.

Schnell wurden andere Möglichkeiten gefunden, Web-Anwendungen zu konstruieren. Auf der Java-Seite ist das **Servlet-API**[27] das Mittel der Wahl zur Programmierung von Web-Anwendungen

Ein **Servlet** ist eine Klasse, die auf dem Web-Server ausgeführt wird und HTTP-Anfragen bearbeitet. Der Name Servlet ist hier in Anlehnung an Applet gewählt. Während ein Applet eine meistens kleine Anwendung ist, die im Browser des Benutzers in einer HTML-Seite eingebettet ausgeführt wird, ist ein Servlet, als Gegensatz dazu, ein Java-Programm, das auf dem Server ausgeführt wird. Ein Entwickler muss sich bei der Programmierung eines Servlets nicht mit dem HTTP auseinandersetzen. Diese technischen Details werden von dem Servlet-API weggekapselt und hinter einer objektorientierten Fassade verborgen (Für ein Beispiel zur Verwendung des Servlet-API siehe Anhang A.2).

[26] HTTP steht für „Hypertext Transfer Protocol“ und ist das Protokoll, über das HTML-Seiten von Server-Rechnern abgerufen werden

[27] API steht für „Application Programming Interface“. Ein API ist im Fall von Java-APIs eine Sammlung von Klassen, die durch Vererbung und Benutzung zur Programmierung von Anwendungen eingesetzt werden kann

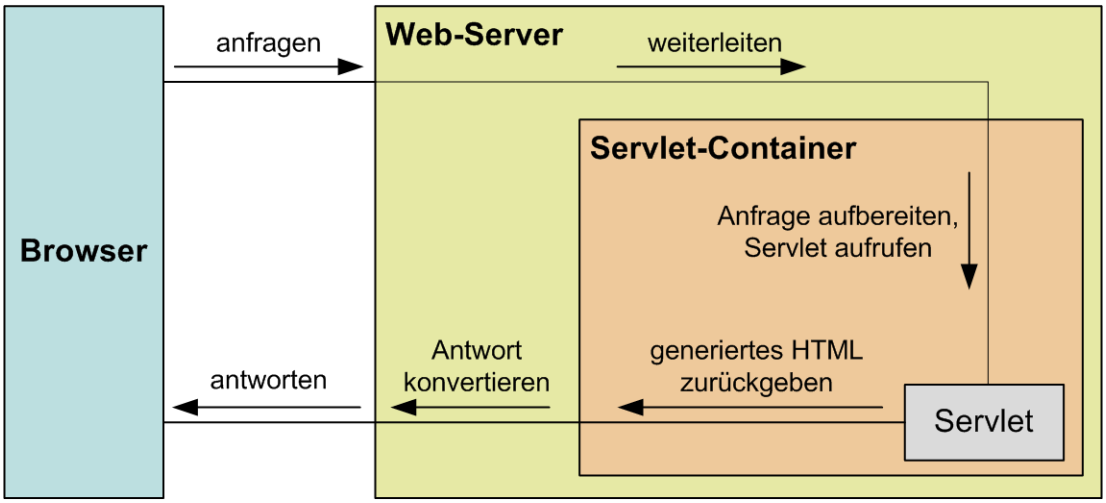

Abbildung 3.4 Interaktion mit einem Servlet

Abbildung 3.4 zeigt wie sich die Interaktion mit einem Servlet gestaltet. Ein Benutzer veranlasst seinen Browser dazu, eine Anfrage zu starten. Diese Anfrage wird vom Web-Server empfangen. Ein Web-Server kann von sich aus nur Anfragen nach statischen Inhalten (HTML-Dateien, Bilder etc.) beantworten. Deshalb leitet der Web-Server die Anfrage an den Servlet-Container weiter. Der Servlet-Container dient als ein Vermittler zwischen dem Web-Server und den Servlets. Der Servlet-Container bereitet die Anfrage für das Servlet auf und ruft das Servlet dann auf. Das Servlet kann nun die Parameter der Anfrage analysieren und als Reaktion eine neue HTML-Seite generieren, die es dann an den Servlet-Container zurückgibt. Der Servlet-Container leitet diese Seite an den Web-Server weiter, der sie an den anfragenden Browser als Antwort sendet.

Obwohl dieses einfache dynamische Erzeugen von HTML-Seiten einige Vorteile bietet und eine Erleichterung gegenüber der Programmierung con CGI-Programmen darstellt, hat es auch einen Nachteil.

Der Nachteil bei der Programmierung von Servlets ist, dass obwohl vielleicht nur ein kleiner Teil der Seite dynamische Inhalte enthält, die sich von Anfrage zu Anfrage ändern, trotzdem die ganze Seite vom Servlet generiert werden muss. Die Programmierung der Generierung ist sehr umständlich und fehlerträchtig. So kann hier kein HTML-Editor für den statischen Teil verwendet werden, der frühzeitig Syntaxfehler erkennbar macht, denn das statische HTML ist in dem Servlet eingebettet.

Ein weit größeres Problem, das mit dem ersten zusammenhängt, ist, dass man die Gestaltung des statischen Teils der Seite nicht einem Web-Designer überlassen kann, was jedoch in großen Projekten häufig gemacht wird. Web-Designer können nur selten in Java programmieren. Sie verwenden visuelle und textuelle Werkzeuge zur Konstruktion von HTML-Seiten. Sie wären also mit der Anpassung der statischen Teile der Seite in einem Servlet überfordert.

Aus diesen Gründen wurde das Servlet-API um das Konzept der **JavaServer Page** – kurz JSP genannt – ergänzt. JSPs sind das genaue Gegenstück zu Servlets. Während es sich bei Servlets um Java-Klassen mit eingestreutem HTML handelt, sind JSPs HTML-Seiten mit eingestreutem Java. Zum Erstellen von JSPs können HTML-Editoren verwendet werden, und Web-Designer können hier auch die statische Struktur der JSP bearbeiten. Java-Programmierer können dann in die JSPs Java-Code einfügen und die Seite somit dynamisieren (Für ein Beispiel zum Thema ,,JavaServer Pages“ siehe Anhang A.3).

Sobald eine JSP zum ersten Mal angesprochen wird, wandelt die JSP-Engine die JSP in ein Servlet um, wobei der HTML-Teil unverändert übernommen wird, und nur der Code, der sich in speziellen JSP-Tags befindet, im Servlet ausgeführt wird.

„Core Servlets and JavaServer Pages“ von Marty Hall ([Hall 00]) sei hier als ein umfassendes Werk zum Thema Servlets und JavaServer Pages genannt.

3.5 Architekturen für Web-Anwendungen mit Servlets und JSPs

Bei der Entwicklung von Web-Anwendungen mit Servlets und JSPs kommt meistens eine von zwei bereits in den ersten JSP-Spezifikationen erwähnten Architekturen zum Einsatz (siehe [Geary 01]).

Die **Model 1 Architektur** ist hierbei die einfachere von beiden. Dabei greifen die JSPs lesend und verändernd über vorgeschaltete Objekte, die Beans genannt werden, weil sie meistens der JavaBeans-Spezifikation[28] genügen, auf die Geschäftsdaten zu. Die Model 1 Architektur ist in Abbildung 3.5 skizziert.

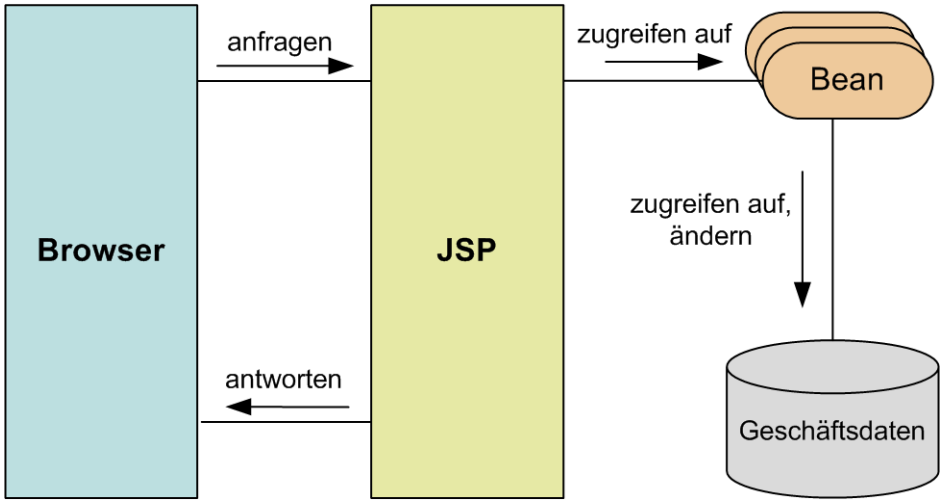

Abbildung 3.5 Model 1 Architektur (nach [Geary 01])

Die Beans schotten die JSPs gewissermaßen von den Geschäftsobjekten ab. Da die JSPs aber die Operationen an den Beans aufrufen, die im Endeffekt Veränderungen an den Geschäftsobjekten vornehmen, enthalten die JSPs einen Teil der Anwendungslogik.

Die **Model 2 Architektur** orientiert sich am **MVC-Muster**.

Das MVC-Muster beschreibt eine Art, Anwendungen mit graphischer Benutzungsoberfläche zu konstruieren und aufzuteilen. Die erste Implementierung in einem Rahmenwerk wurde für die Sprache Smalltalk unternommen.

Steve Burbeck beschreibt in [Burbeck 92] die Grundkonzepte von MVC wie folgt

> „In the MVC paradigm the user input, the modeling of the external world, and the visual feedback to the user are explicitely separated by three types of object, each specialized for its task. The **view** manages the graphical and/or textual output to the portion of the bitmapped display that is allocated to its application. The **controller** interprets the mouse

[28] Objekte, die der JavaBeans-Spezifikation genügen haben für jedes Attribut eine lesende Operation, die mit „get“ anfängt, und eine schreibende Operation, die mit „set“ anfängt

> and keyboard inputs from the user, commanding the model and/or the view to change as appropriate. Finally , the **model** manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view) and responds to instructions to change state (usually from the controller).“

Hier erfolgt also eine Aufteilung einer Anwendung in drei Bereiche, die sich jeweils um spezifische Aufgaben innerhalb einer Anwendung kümmern. Das Model hält die Daten der Anwendung. Das View zeigt die Daten des Models an. Der Controller reagiert auf Ereignisse der graphischen Benutzungsoberfläche wie die Bewegung der Maus oder das Drücken einer Taste und weist das Model oder das View an sich den Wünschen des Benutzers entsprechend zu verändern.

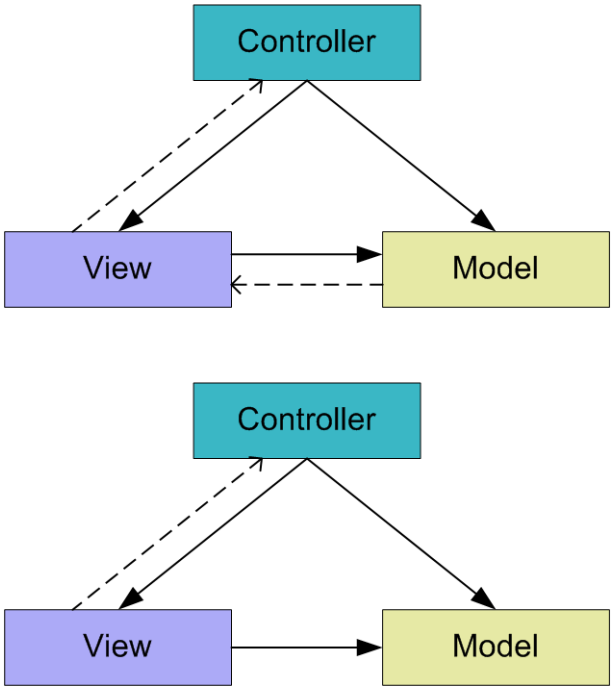

Abbildung 3.6 MVC: oben aktive Variante; unten passive Variante

Burbeck beschreibt zwei Arten von MVC, die in Abbildung 3.6 skizziert sind.

Bei der aktiven Variante weiß das Model, dass es das Model in einem MVC-Dreiergespann ist. Da es davon weiß, informiert es das View über den Ereignismechanismus (siehe dazu das Beobachter-Muster in [Gamma 98]), wenn sich etwas an ihm geändert hat.

Bei der passiven Variante weiß das Model nichts von seiner Rolle. Es kann ein beliebiges fachliches Objekt sein (oder in der WAM-Welt genauer: ein Material. Siehe hierzu auch das Muster ,,Materialentwurf“ in [Züllighoven 98]). Es löst keine Ereignisse aus, wenn sich etwas an ihm ändert. Der Controller weist bei dieser Variante das View direkt an, sich zu aktualisieren, wenn das Model geändert wurde. Das geht natürlich nur dann, wenn der Controller über alle Änderungen Bescheid weiß, was wiederum nur dann möglich ist, wenn es der einzige Controller ist, der mit dem Model arbeitet.

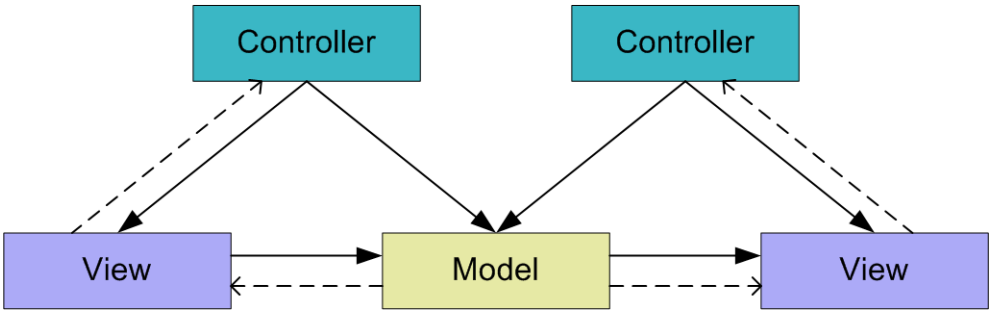

Abbildung 3.7 Mehrere Sichten auf dasselben Model

Die passive Variante hat den Vorteil, dass fachliche Klassen für MVC nicht angepasst werden müssen und ihren Material-Charakter nicht verlieren. Die aktive Variante hat den Vorteil, dass sie es ermöglicht, zur gleichen Zeit verschiedene Sichten auf die gleichen Daten zu haben.

Abbildung 3.7 zeigt, wie es aussieht, wenn zwei Sichten auf das gleiche Model angezeigt werden. Es gibt hier zwei Views, die mit dem selben Model arbeiten. Diese Views werden beide über Änderungen am Model von dem Model benachrichtigt. Für jeden der beiden Views gibt es einen Controller, der die Ereignisse des jeweiligen Views interpretiert und in Änderungen am Model oder dem jeweiligen View übersetzt. Zu beachten ist hier, dass sich weder die Views noch die Controller untereinander kennen. Jedes Controller/View-Paar verhält sich so als wäre es das einzige, das mit dem Model arbeitet. Einzig das Model bildet hier eine Schnittstelle zwischen den beiden.

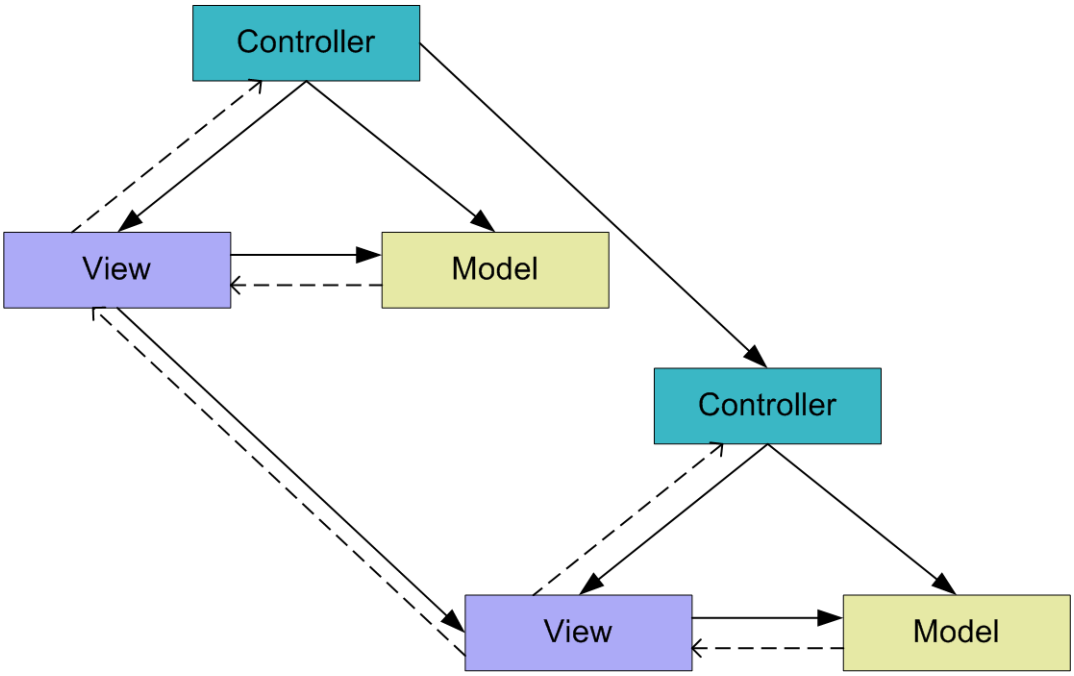

Abbildung 3.8 MVC-Hierarchien

Wie in Abbildung 3.8 zu sehen können in MVC auch Hierarchien gebildet werden. So können komplexe Views aus kleineren zusammengesetzt werden. Zu jedem View gehört ein eigener Controller und ein eigenes Model. Dies ermöglicht die Modularisierung einer Anwendung, in der sich jedes

Modul mit einem kleinen Teilaspekt der ganzen Anwendung befasst. Die Verschachtelung der Views führt zu einer ähnlichen Verschachtelung unter den Controllern. Bewegt der Benutzer die Maus über den Bildschirm, so fragt der oberste Controller alle seine Sub-Controller, ob sie die Verarbeitung dieser Aktion übernehmen wollen. Nur der Controller, in dessen View sich die Maus befindet wird diese Frage bejahen. Dieser Controller wird dann alle seine Sub-Controller fragen, womit das ganze von vorne anfängt, bis ein Controller die Aktion verarbeitet, weil es keinen Sub-Controller hat, der dies für sich beanspruchen würde.

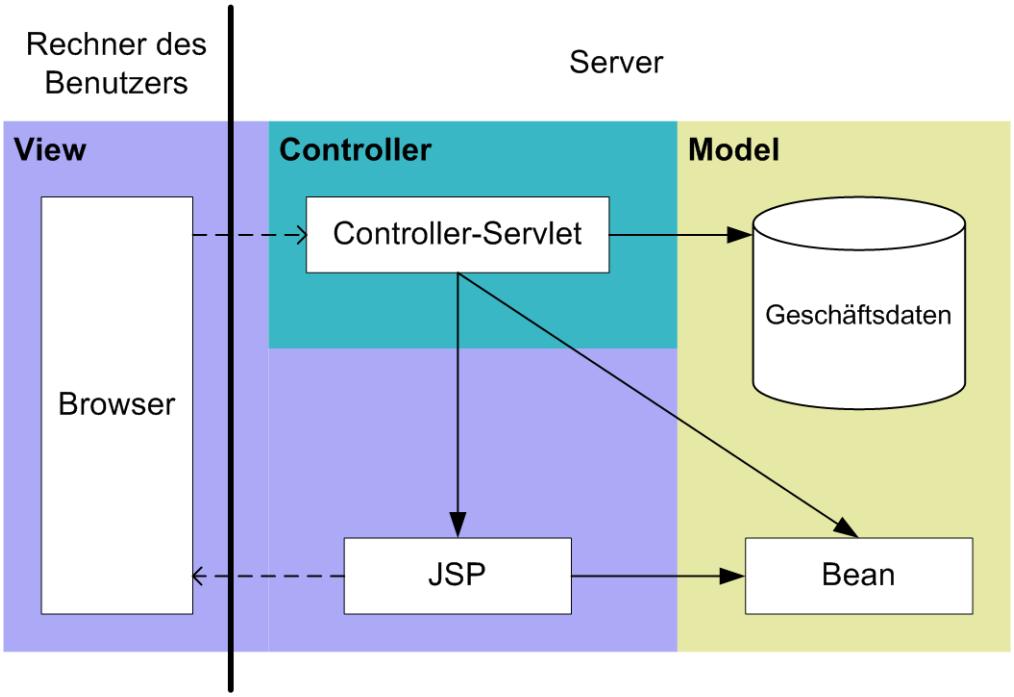

Abbildung 3.9 Verteilung der Rollen Model, View und Controller in der Model 2-Architektur

Abbildung 3.9 zeigt wie in einer Model 2-Architektur die Rollen Model, View und Controller besetzt werden.

Als Controller wird ein Servlet eingesetzt, das Controller-Servlet genannt wird. Es gibt im Gegensatz zu MVC, wo es beliebig viele Controller geben kann, nur ein Exemplar dieses Servlets pro Web-Anwendung. Wie auch in einer MVC-Anwendung, reagiert der Controller auf Ereignisse des Views und verändert das Model und das View.

Das Model ist zweigeteilt. Der eine Teil des Models sind die Geschäftsdaten, die in Datenbanken enthalten sind. Den zweiten Teil bilden die Beans. Eine Bean wird vom Controller mit gerade den Daten befüllt, die von der nächsten anzuzeigenden JSP benötigt werden. Das Model löst nach einer Veränderung keine Ereignisse aus.

Das View ist ebenfalls zweigeteilt. Zum einen gehört der Browser zum View, denn er zeigt die einzelnen Seiten einer Web-Anwendung an. Zum anderen gehören die JSPs zum View. Sie lesen die Daten aus den für sie bereitgestellten Beans und generieren HTML-Seiten, die an den Browser zum Zweck der Anzeige geschickt werden. Der Browser schickt neue Anfragen des Benutzers – wenn dieser auf einen Knopf oder Link klickt – an das Controller-Servlet zurück.

Mehrere Dinge fallen hier auf.

Da das Model nicht mit dem Verschicken eines Ereignisses auf Veränderungen an ihm reagiert, ist die Model 2-Architektur mit der passiven MVC-Variante vergleichbar. Ereignisse würden hier auch keinen Sinn machen, weil es niemanden gäbe, der sie empfangen könnte. Das View ist, wenn die JSP die Seite erzeugt und verschickt hat, durch den Server-Teil der Anwendung nicht erreichbar. Erst, wenn der Benutzer auf einen Knopf oder Link klickt, hat der Server-Teil eine Möglichkeit, das View zu ändern.

Es können keine View-Hierarchien mit eigenen Controllern konstruiert werden. Es ist immer eine JSP zuständig dafür, was der Benutzer sehen wird. Es ist eigentlich auch nicht möglich mehrere Ansichten auf das gleiche Model zu erhalten. Eine Web-Anwendung kann keine voneinander unabhängigen Views haben. Für einen Benutzer gibt es immer einen Browser mit einer aktuellen Seite der Anwendung. Der Benutzer kann dies in manchen Web-Anwendungen umgehen, indem er ein neues Browser-Fenster öffnet und eine neue Sitzung in derselben Web-Anwendung anfängt. In diesem Fall kann es passieren, dass er zwei Views auf dieselben Daten enthält. Das kann jedoch zu Problemen in der Web-Anwendung führen.

Wenn der Benutzer zum Beispiel in beiden Sitzungen seine Kundendaten betrachtet und in beiden Ansichten etwas ändert, wird es problematisch, wenn er zuerst in der einen Ansicht die Änderungen abschickt und dann in der anderen. Das zweite Abschicken wird möglicherweise die Änderungen, die zuerst abgeschickt wurden, rückgängig machen. Hier fehlt also, dass das Model bei jeder Änderung eine Nachricht an alle Views schickt, wenn es verändert wurde.

Eine weitere Eigenart von Web-Anwendungen, die bei der Betrachtung der Model 2-Architektur beachtet werden muss, ist, dass Ereignisse vom View viel seltener sind als in einer MVC-Anwendung. Während das View in einer MVC-Anwendung bei jeder Mausbewegung und jedem Tastendruck ein Ereignis auslöst, das vom Controller interpretiert werden kann, löst nur das Klicken auf einen Link oder einen Knopf in einer Model 2-Architektur ein Ereignis des Views aus.

Auch mit einer Model 2-Architektur bleiben Web-Anwendungen Seiten-basiert.

3.5.1 Bewertung

Eine Model 1-Architektur eignet sich nur für Prototypen und sehr kleine Web-Anwendungen. Hier sind Präsentation und Anwendungslogik größtenteils in den JSPs vermischt. Eine solche Strukturierung einer Web-Anwendung führt in komplexeren Anwendungen zur Doppelung von Funktionalitäten in verschiedenen JSPs und erschwert die Einarbeitung in den Code einer derartigen Anwendung. Da Prototypen irgendwann zu Produkten heranreifen und kleine Web-Anwendungen im Laufe der Zeit anwachsen können, ist die Model 1-Architektur für keine Web-Anwendung zu empfehlen – auch nicht für Prototypen und anfangs kleine Web-Anwendungen.

Eine Migration von der Model 1-Architekur zur Model 2-Architektur dürfte sich aufgrund der Probleme mit der Model 1-Architektur als schwierig erweisen.

Web-Anwendungen sollten von Anfang an mit einer Model 2-Architektur entwickelt werden. Diese Architektur bietet die Trennung zwischen Präsentation und Anwendungslogik, die eine Anwendung lesbarer und damit auch wartbarer machen. Außerdem ermöglicht diese Aufteilung die Einbeziehung eines Web-Designers, der das Layout der Seiten machen kann ohne sich auch um Aspekte der Anwendungslogik kümmern zu müssen. Durch die Bereitstellung von speziellen Beans für die jeweiligen JSPs wird die Präsentation einer Anwendung von der Struktur der Geschäftsdaten entkoppelt. Diese Trennung macht klar, dass die JSPs nichts am Model ändern, und welche Daten sie wirklich brauchen, um eine Seite zu generieren.

Dies führt zu einer weiteren Anforderung an eine Lösung der in dieser Arbeit behandelten Probleme.

Web-Anwendungen sollen eine Model 2-Architektur verwenden, um eine Trennung zwischen Präsentation und Anwendungslogik zu erreichen.

3.6 Bestehende Validierungsmöglichkeiten auf Basis von JavaServer Pages

Es gibt einige Lösungen auf Basis der Technologie JSP für das Problem, dass HTML-Formulare nicht genügend Möglichkeiten bieten, festzulegen, welchen Regeln die Eingaben entsprechen müssen, um als valide Eingaben zu gelten.

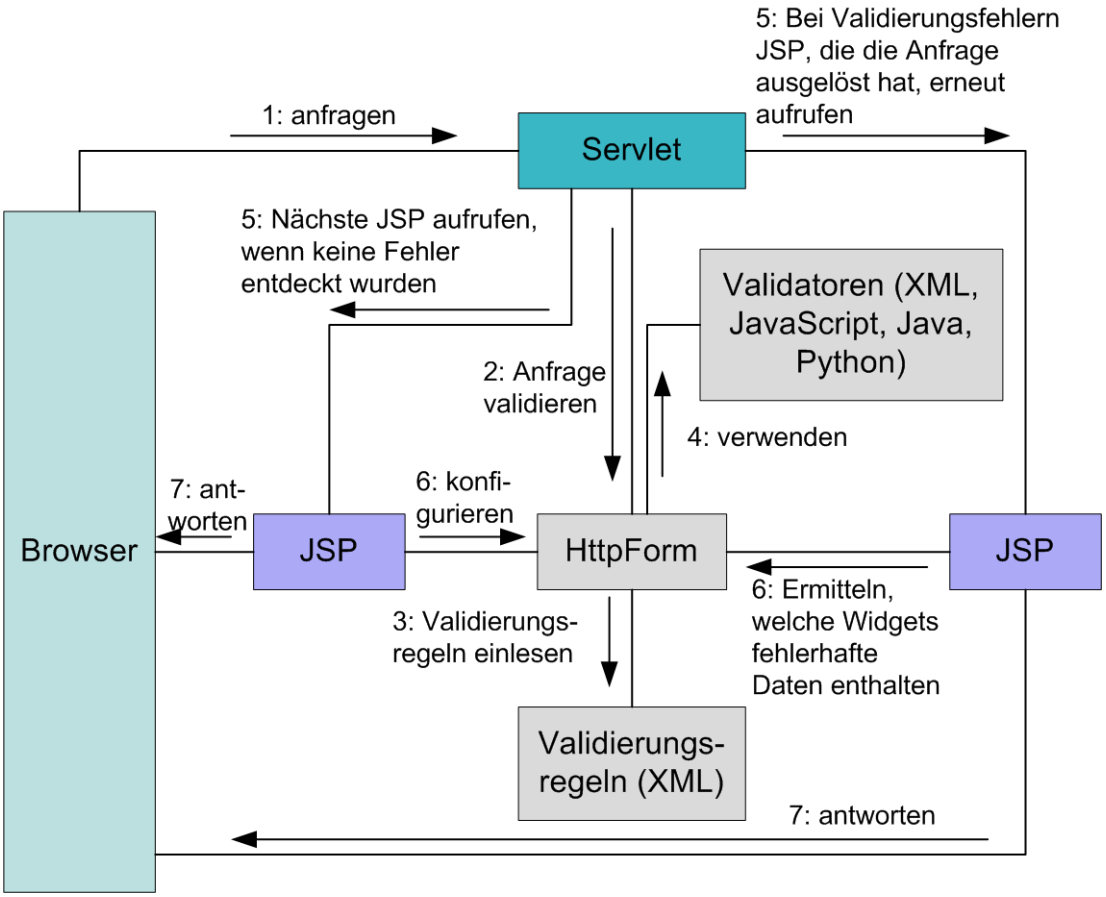

Abbildung 3.10 Verwendung von FormProc

Eine dieser Lösungen ist FormProc (siehe [FormProc-website]), eine API, die zu genau diesem Zweck entwickelt wurde. Die Verwendung FormProcs in einer Web-Anwendung ist in Abbildung 3.10 skizziert.

Nachdem ein Benutzer ein Formular ausgefüllt und an den Server abgeschickt hat, wird die Anfrage von einem Servlet empfangen. FormProc setzt keine bestimmte Architektur bei einer Web-Anwendung voraus. Das Servlet ist Teil der Anwendung und nicht FormProcs. FormProc bietet also eine Dienstleistung für Web-Anwendungen an und tritt nicht selbst in den Vordergrund. Das Servlet ruft nun ein `HttpForm`-Objekt auf, das dann die Anfragedaten validiert.

`HttpForm` liest mehrere XML-Dateien ein. Eine davon ist die Definition der Validierungsregeln für das Formular, das validiert werden soll. Hier ist für jedes Dateneingabewidget festgelegt mit welchem Validator der Inhalt des Widgets validiert werden soll.

FormProc ermöglicht es, Validatoren lokal für ein einziges Formular zu definieren oder auch global für alle Formulare einer Anwendung. Über die globale Definition lassen sich Validatoren für häufiger auftretende Datentypen an einer zentralen Stelle definieren und wiederverwenden.

FormProc enthält grundlegende Validatoren, die zum Beispiel die Existenz einer Eingabe oder deren Länge überprüfen können. Reichen diese Validatoren nicht aus, kann ein Entwickler selbst neue Validatoren definieren.

Validatoren können auf unterschiedliche Arten definiert werden. Die einfachste Möglichkeit ist die Verknüpfung von bereits eingebauten Validatoren zu neuen Validatoren. Solche Validatoren können in einer XML-Datei und ohne zusätzlichen Programmieraufwand definiert werden.

Validatoren können aber auch in JavaScript, Python oder Java definiert werden, wobei die Java Validatoren vor dem Einsatz compiliert werden müssen. Java-Validatoren können aber auch alle Möglichkeiten ausschöpfen, die Java-Programmen zur Verfügung stehen.

Nachdem das `HttpForm` das Wissen darüber erlangt hat, welche Validatoren verwendet werden sollen, werden diese der Reihe nach aufgerufen.

Hat die Validierung ergeben, dass eine oder mehrere Eingaben fehlerhaft waren, ruft das Servlet die JSP erneut auf, die die Anfrage ausgelöst hat. Die JSP zeigt dann das Formular mit den gemachten Eingaben an und kennzeichnet jedes Widget, das fehlerhafte Eingaben enthält. Bei der nächsten Anfrage fängt der Zyklus von vorne an.

Hat die Validierung jedoch keine Fehler aufgedeckt, ruft das Servlet das nächste JSP auf. Dieses JSP konfiguriert das einzige `HttpForm`-Objekt, dass in einer Sitzung vorhanden ist mit dem Namen des Formulars und generiert das HTML. das dem Benutzer angezeigt wird. Die Konfiguration des `HttpForm` sorgt dafür, dass dieses bei der nächsten Anfrage weiß, welches Formular validiert werden muss (Siehe Anhang A.4 für ein Beispiel zu FormProc und eine genauere Betrachtung der technischen Details).

FormProc bietet keine eigene TagLib an. Die Konfiguration des Formulars in der JSP und die Aufrufe an `HttpForm`, die dazu nötig sind, um feststellen zu können, welche Widgets fehlerhafte Daten enthalten, müssen über eingestreutes Java geschehen, was die JSPs schnell unübersichtlich und überfrachtet aussehen lässt.

3.6.1 Bewertung

FormProc wiegt eines der großen Nachteile bei den HTML-Formularen auf. Mit FormProc können komplexe Validierungsregeln definiert werden und der Programmier-Aufwand, der für die Validierung mit FormProc nötig ist, ist viel geringer als wenn der Entwickler jede Validierung selbst programmieren müsste. Einfache Validatoren sind bereits in FormProc enthalten und lassen sich leicht zu komplexeren Validatoren zusammensetzen. Genügen auch diese Möglichkeiten nicht, so können Validatoren in Python oder Java programmiert werden. Mit FormProc lassen sich also auch komplexe Validatoren konstruieren

Ein Nachteil bei FormProc ist, dass die Validierung auf Widget-Basis durchgeführt wird. Wenn es für eine Datumsangabe drei Eingabefelder für Tag, Monat und Jahr gibt, so kann mit FormProc jedes dieser 3 Elemente für sich einzeln validiert werden (zum Beispiel, dass der Tag zwischen 1 und 31 liegt), aber es kann nicht validiert werden, dass die drei Eingabefelder zusammen ein valides Datum enthalten.

Daraus ergibt sich eine Anforderung an die Lösung der in dieser Arbeit behandelten Probleme.

> Ein Formular besteht aus Feldern, die Daten aufnehmen können. Wird ein Formular angezeigt, so kann es sein, dass mehrere Widgets für ein Feld verwendet werden. Die Validierung eines Formulars muss die Felder des Formulars validieren und nicht die einzelnen Widgets.

Dass FormProc keine TagLib anbietet und deshalb Java-Code in die JSPs eingestreut werden muss, ist ein weiterer Nachteil von FormProc. Die Einstreuung von Java-Code in die JSPs macht diese unleserlicher und Web-Designern fällt es schwieriger solche JSPs zu bearbeiten.

Eine weitere Anforderung erwächst hieraus

> Eine Lösung soll eine JSP-TagLib anbieten, um zu verhindern, dass Java-Code für die Verwendung der Lösung in JSPs eingestreut werden muss.

3.7 Struts

Struts ist ein Rahmenwerk für die Entwicklung von Web-Anwendungen in Java. Struts-Anwendungen haben eine Model 2-Architektur.

Wie die Hauptkomponenten einer mit Struts entwickelten Web-Anwendung miteinander kollaborieren ist in Abbildung 3.11 skizziert.

In einer Struts-Anwendung gibt es nur ein Servlet – das `ActionServlet`. Das `ActionServlet` empfängt die Anfragen, die an die Anwendung gerichtet sind. Um zu erfahren, wie mit der Anfrage zu verfahren ist, lädt es eine XML-Konfigurationsdatei.

Bei jeder Anfrage, die von einem Formular in einer JSP der Anwendung ausgelöst wird, wird der Name der Anfrage mitgeschickt. Über diesen Namen kann das `ActionServlet` mit Hilfe der Konfiguration herausfinden, welches `ActionForm` mit den Anfrageparametern befüllt werden muss.

Eine `ActionForm` ist im einfachsten Fall eine JavaBean. Ist das der Fall, wird ermittelt welche Attribute dieses JavaBean hat und diese werden, falls sie in den Anfrageparametern vorhanden sind, an der `ActionForm` gesetzt.

Soll nicht für jede Anfrage eine eigene `ActionForm` implementiert werden, kann auch eine `DynaActionForm` verwendet werden, die ein generischer Daten-Container ist. Wird eine `DynaActionForm` verwendet, muss in der Konfigurationsdatei spezifiziert werden, welche Attribute sie annehmen soll.

Nach der Befüllung wird die `ActionForm` angewiesen, sich selbst zu validieren. Schlägt die Validierung fehl, wird die JSP aufgerufen, die die Anfrage ausgelöst hat und der Benutzer hat die Möglichkeit, die Eingabefehler zu korrigieren[29].

Verläuft die Validierung erfolgreich, so ermittelt das `ActionServlet` danach anhand der Konfiguration, welche `Action` für die weitere Bearbeitung der Anfrage zuständig ist.

Typischerweise gibt es für jede mögliche Anfrage eine `Action`. Diese holt, wenn sie aufgerufen wird, die Daten aus der `ActionForm` und verarbeitet sie anwendungsspezifisch. Meistens benutzt eine `Action` eine Datenbank, in der die Daten der Anwendung verwahrt werden. Die `Action` füllt darauf eine neue `ActionForm` mit den Daten und gibt den Namen einer Weiterleitung an das `ActionServlet` zurück.

[29] Abbildung 3.11 zeigt nur den Fall, in dem die Validierung erfolgreich verläuft. Für eine nähere Betrachtung der Validierung mit Struts und der Behandlung von fehlgeschlagenen Validierungen siehe Abschnitt 3.8

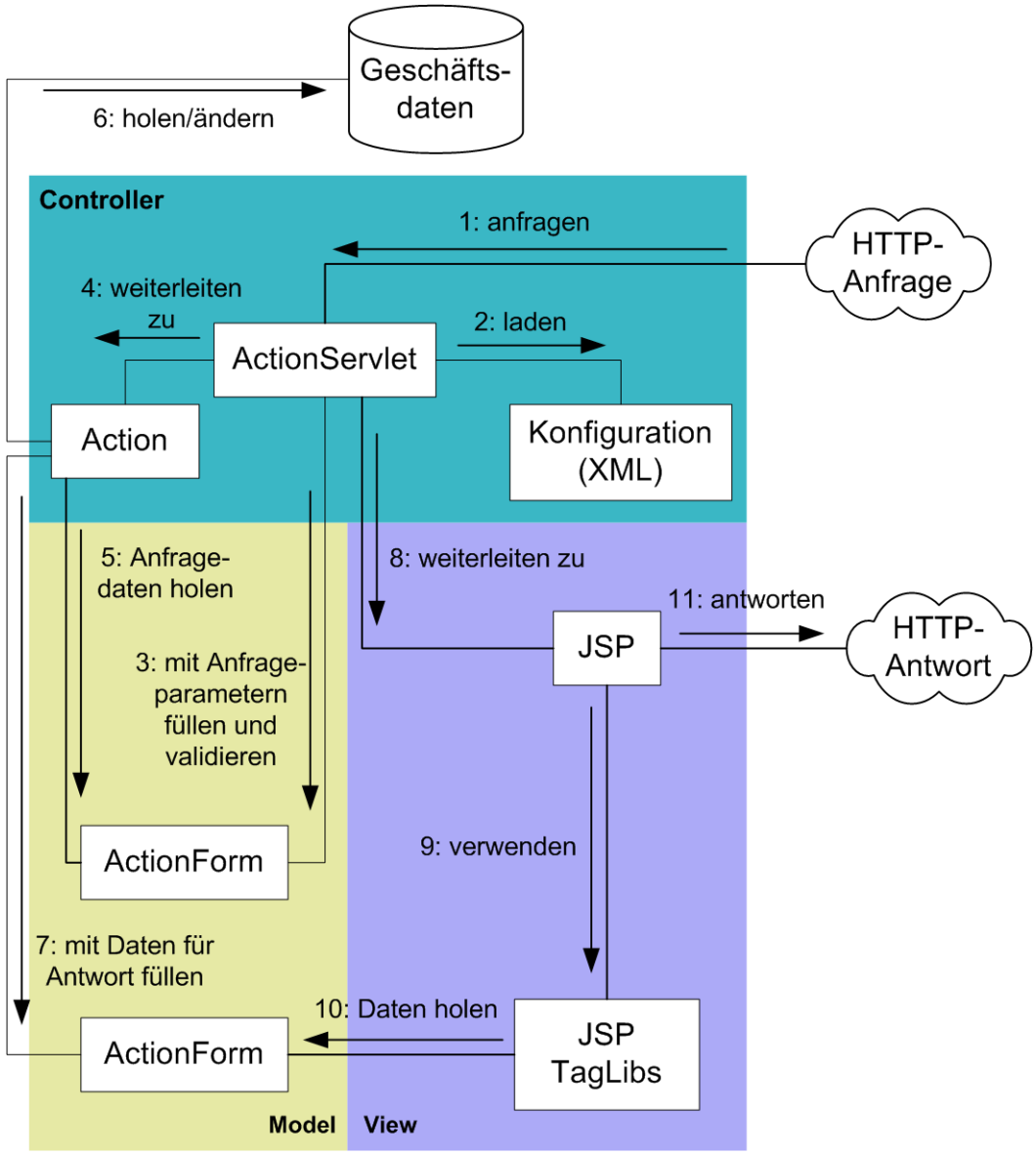

Abbildung 3.11 Kollaborationsdiagramm einer mit Struts entwickelten Web-Anwendung

Das `ActionServlet` löst diesen Namen über die Konfiguration auf und ruft die damit identifizierte JSP auf. Die JSP verwendet in Struts enthaltene TagLibs, um auf die Daten der Antwort-`ActionForm` zuzugreifen und diese darzustellen.

3.7.1 Bewertung

Struts erleichtert die Konstruktion von Model 2-Web-Anwendungen, da es das Gerüst für solche Anwendungen liefert. Der Entwickler muss sich nur noch um die für seine Anwendung spezifischen Dinge kümmern.

Struts verhält sich dem Entwickler gegenüber wie eine Black-Box mit wohldefinierten Steckplätzen, in die der Entwickler die Komponenten seiner Anwendung stecken muss.

Die Konfiguration in einer XML-Datei stellt hier den Plan dar, wie die einzelnen Teile der Anwendung während der Laufzeit zusammenarbeiten sollen.

Ein weiterer positiver Punkt ist die Bereitstellung von Tag-Libs, die mithelfen, die JSPs einer Anwendung frei von Java-Code zu halten.

Ein Nachteil von Struts sind die `Action`s. Für jede JSP, die eine Anfrage auslösen kann, muss es eine eigene Action geben, die diese Anfrage bearbeiten kann. Es ist nicht vorgesehen, Anfragen aus dem selben Bereich, wie zum Beispiel der Benutzerverwaltung, zu gruppieren und von einer `Action` verarbeiten zu lassen. Die Anwendungslogik verteilt sich bei einer großen Anwendung feingranular über viele `Actions`. Das kann schnell zu Unübersichtlichkeiten führen.

Dem kann jedoch abgeholfen werden, wenn die Rolle der Actions etwas umdefiniert wird. Wenn man die Actions als Vermittler zwischen dem mit Struts entwickelten Web-Klienten und fachlichen Dienstleistern[30] ansieht, die die eigentliche Anwendungslogik enthalten, so wiegt dieser Nachteil nicht mehr so schwer, denn nun sind Actions leichtgewichtige Objekte, die die Anfragen interpretieren und in Anfragen an fachliche Dienstleister umwandeln. Diese fachlichen Dienstleister wissen nichts darüber, wie der Benutzer auf die Anwendung zugreift. Er kann sie über eine Web-Schnittstelle oder auch über eine vollwertige Desktop-Anwendung verwenden. Fachliche Dienstleister sorgen für eine gute Kapselung der fachlichen Logik und bereiten den Weg für Multi-Channeling-Anwendungen[31] vor.

Hieraus läst sich eine weitere Anforderung an die Lösung ableiten

> Web-Anwendungen sollen fachliche Dienstleister verwenden und somit eine Trennung zwischen der fachlichen Logik und der Logik erreichen, die sich ausschließlich mit den speziellen Anforderungen der Web-Anwendung befasst.

3.8 Bestehende Validierungsmöglichkeiten auf Basis von Struts

Struts bietet ein optional einsetzbares Validierungsrahmenwerk, das von der Herangehensweise an FormProc (siehe dazu Abschnitt 3.6) erinnert.

Abbildung 3.12 skizziert die Verwendung des Struts-Validators.

Wenn der Struts-Validator verwendet wird, kommt eine Spezialisierung der `ActionForm` zum Einsatz, die ein Teil des Validator-Rahmenwerkes ist. Die `ValidatorActionForm` wendet sich an das zentrale `Resources`-Objekt des Validator-Rahmenwerkes, wenn sie angewiesen wird, sich zu validieren.

`Resources` verwaltet unter anderem die Konfiguration der Validatoren, die aus einigen XML-Dateien besteht, und kann mit dieser Validator-Objekte erzeugen, die eine `ValidatorActionForm` validieren können.

Ein Validator ist sehr generisch und nicht darauf beschränkt `ValidatorActionForm`s zu validieren. Die `ValidatorActionForm`, auf die er in der hier beschriebenen Anwendung Zugriff nimmt, kennt er

[30] Zum Thema der fachlichen Dienstleister siehe auch [Otto 00]

[31] Als Multi-Channeling-Anwendungen werden Anwendungen bezeichnet, die über mindestens Benutzungsschnittstellentypen zugänglich sind. Heutige Multi-Channeling-Anwendungen bieten meistens einen Zugang über eine Desktop-Anwendung (Diese wird dann auch oft Rich-Client-Anwendung genannt, weil sie über die breiten Möglichkeiten von Desktop verfügen) und einen über ein Web-Interface, das von jedem Rechner mit Internet-Zugang und Web-Browser aufgerufen werden und benutzt werden kann

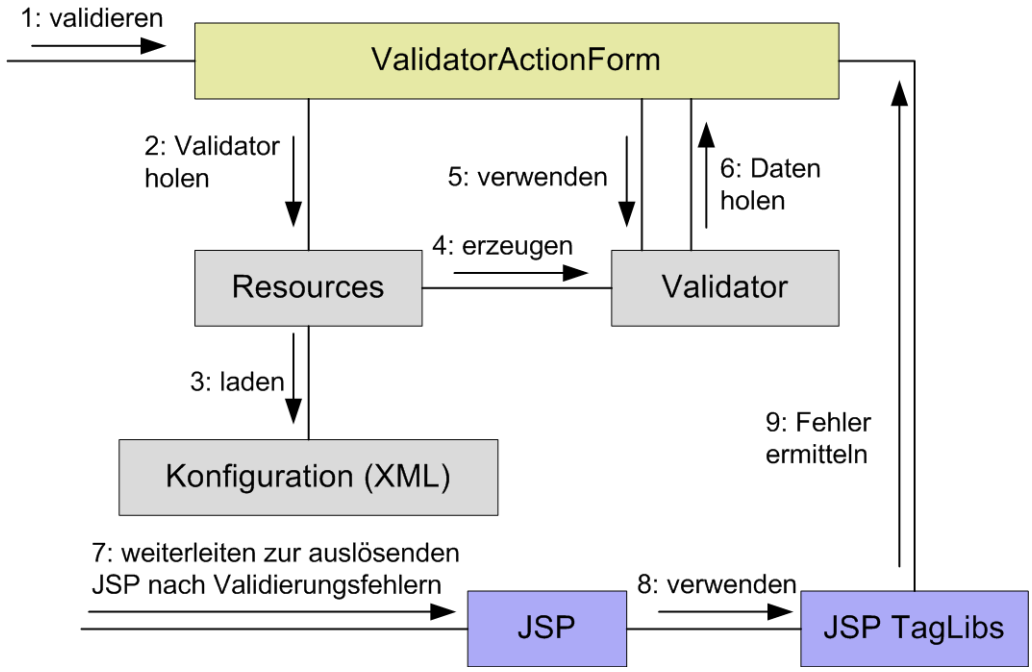

Abbildung 3.12 Verwendung des Struts-Validator-Rahmenwerkes

nur als JavaBean. Es wäre also durchaus möglich den Validator auch in einer Desktop-Anwendung zu verwenden.

Der Validator geht alle Felder in seiner Konfiguration durch, holt sich für jedes Feld die Daten aus der Bean und validiert sie entsprechend der Konfiguration.

Das Validator-Rahmenwerk bietet von sich aus bereits eine Anzahl von Widget-Validatoren[32]. Es können aber auch eigenen Widget-Validatoren in Java programmiert werden.

Alle Validatoren werden in den Konfigurationsdateien des Rahmenwerks konfiguriert und können dort auch für einzelne Felder einer JavaBean kombiniert werden.

falls bei der Validierung Fehler auftreten, ruft Struts die JSP erneut auf, die die Anfrage ausgelöst hat. Über eine Struts-TagLib ermittelt die JSP die Fehler und zeigt die Seite erneut mit den eingegebenen Daten an.

Anhang A.6 zeigt anhand eines Beispiels einige Details der Konfiguration des Validator-Rahmenwerkes

3.8.1 Bewertung

Gegenüber FormProc hat das Validierungs-Rahmenwerk den Vorteil, das es in Struts integriert ist und die Verwendung in Struts-Anwendungen unproblematisch ist. Wie FormProc verwendet es XML-Konfigurationsdateien, bietet eine Auswahl an vordefinierten Widget-Validatoren an und ermöglicht die Programmierung eigener Widget-Validatoren. Das Validierungs-Rahmenwerk bietet zwar nicht wie FormProc die Möglichkeit an, Widget-Validatoren in Python zu definieren, aber

[32] Das Rahmenwerk verwendet für das Objekt, das durch alle Widgets eines Formulars iteriert und für die einzelnen Widget-Validatoren den gleichen Namen „Validator“. Aus Gründen der Klarheit werden hier die einzelnen Validatoren, die die Widgets validieren Widget-Validatoren genannt.

dieser Nachteil wiegt nicht sehr schwer, denn die Validatoren, die in Java entwickelt werden können, genügen hier jeder Validierungsanforderung.

Ein Vorteil gegenüber FormProc ist auch, dass eine eigene TagLib vorhanden ist, mit der auf die Fehlermeldungen der Validierung zugegriffen werden kann.

Ein Nachteil, der auch bereits auf FormProc zutrifft, ist, dass nur die Inhalte von Widgets und nicht wirklich Formularfelder validiert werden können. Wie auch FormProc ist das Validierungs-Rahmenwerk nur in Anwendungen verwendbar, die keine Formularfelder verwenden, die von mehreren Widgets gemeinsam befüllt werden.

Auch scheint es etwas merkwürdig, dass die Daten erst dann validiert werden, wenn sie bereits in die `ActionForm` gefüllt wurden. Damit wird klar, dass die `ActionForm` nur ein generischer Daten-Container ist, der keine fachliche Bedeutung haben kann.

3.9 wingS

wingS ist ein Open Source-Projekt, das von einigen Leuten der Mercatis GmbH und anderen Freiwilligen entwickelt wird (siehe auch [wingS-website]).

Der Name, der den Eindruck macht, falsch geschrieben zu sein, birgt auch schon die wichtigsten Informationen. Nimmt man das große „S“ vom Ende des Namens und setzt es an die erste Stelle, so wird daraus „Swing“. Swing (siehe auch [Eckstein 98]) wiederum ist ein GUI-Toolkit[33] für in Java geschriebene Desktop-Anwendungen. Swing hat sich in den letzten Jahren zum Standard-GUI-Toolkit für die Java-Welt entwickelt. Viele Entwickler wissen, wie man Oberflächen mit Swing-Widgets zusammenbaut.

Die Einsicht, dass es mehr Entwickler gibt, die mit Swing umgehen können, als solche, die eine Web-Anwendung entwickeln können, führte zur Entstehung von wingS. wingS ist ein **Swing für Web-Anwendungen**. Die Idee ist, dass Entwickler Web-Anwendungen genauso entwickeln können sollen wie Desktop-Anwendungen. Dazu enthält wingS Gegenstücke zu beinahe allen Swing-Widgets, die größtenteils die gleichen Schnittstellen haben. Layout-Manager, mit denen die Widgets in einem Fenster angeordnet werden können, stehen auch hier zur Verfügung.

Abbildung 3.13 zeigt die Verwendung von wingS in einer Anwendung.

In einer wingS-Anwendung gibt es ein einziges Servlet – das `WingsServlet` –, das alle Anfragen annimmt. Für eine Anfrage ermittelt dieses Servlet die Session des Benutzers und ruft den in dieser Session vorhandenen Dispatcher auf.

Der Dispatcher erzeugt aus den Anfrageparametern Low-Level-Events, die er an die GUI-Elemente schickt, aus denen die Präsentation der Anwendung besteht.

Diese Low-Level-Events führen dazu, dass die Veränderungen, die der Benutzer an den Elementen in der HTML-Seite vorgenommen hat, nun auch in den GUI-Elementen nachvollzogen werden, die ihre Gegenstücke in der HTML-Seite in der Anwendung vertreten. Nachdem ein GUI-Element seinen Status aktualisiert hat, schickt es, falls notwendig High-Level-Events an Listener, die die Anwendungslogik an ihnen angemeldet hat. Ein `SButton` würde zum Beispiel, wie sein Gegenstück bei Swing, ein `ActionEvent` an etwaige an ihm angemeldete `ActionListener` schicken.

Die Listener rufen als Reaktion auf ihren Aufruf verschiedene Teile der Anwendungslogik auf, die wiederum im Endeffekt Veränderungen an der Präsentation der Anwendung vornimmt.

Sobald alle Low-Level-Events und High-Level-Events abgearbeitet sind, ruft das `WingsServlet` den Externalizer auf, der sich ebenfalls in der Session befindet.

[33] GUI-Toolkits sind Zusammenstellungen von Widgets und Hilfsklassen, die zur Anwendungsentwicklungen mit diesen Widgets benötigt werden.

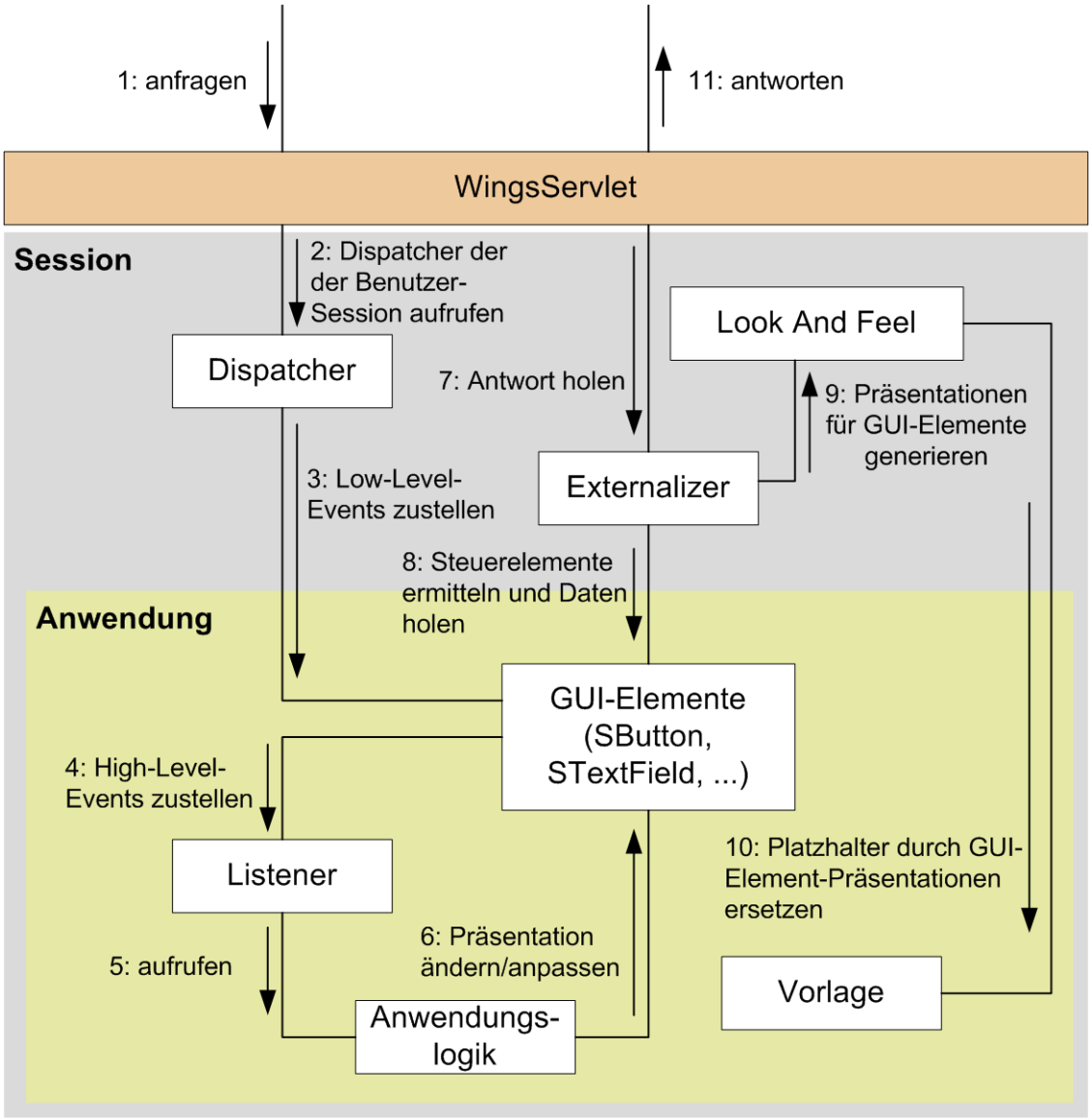

Abbildung 3.13 Verwendung von wingS

Der Externalizer muss nun eine Antwort generieren. In den meisten Fällen wird das eine HTML-Seite sein, aber es können auch Antworten in anderen Dokumentenbeschreibungssprachen wie zum Beispiel WML[34] generiert werden.

Um diese Unabhängigkeit von der Sprache zu erlangen, in der die Antwort verfasst wird, verwendet wingS ein Konzept, das auch in Swing bekannt ist. Es handelt sich um das Konzept des ,,austauschbaren Look And Feels“. Bei Swing kümmern sich diese Look And Feels um das Zeichnen der einzelnen GUI-Elemente. Bei wingS werden diese Elemente nicht gezeichnet. Bei wingS generiert ein Look And Feel für jedes GUI-Element Code in der Antwort-Sprache, der dann in

[34] WML = Wireless Markup Language; HTML-artige Dokumentenbeschreibungssprache für Mobiltelefone und andere Kleingeräte mit Internetzugang

der Anwendung, die die Antwort erhält – meistens ein Browser –, zur Anzeige eines oder mehrerer Widgets führt, die für das wingS-GUI-Element stehen. Das ,,CSS1 Look And Feel", das das Standard-Look And Feel in wingS ist, generiert XHTML mit CSS[35].

Wie in Swing, so stehen auch in wingS mehrere LayoutManager zur Verfügung, mit denen die GUI-Elemente in GUI-Containern angeordnet werden können. Jedes Look And Feel erzeugt daraus Code in der Antwort-Sprache, der dafür sorgt, dass die Elemente entsprechend platziert werden.

Ein LayoutManager, den es bei Swing nicht gibt, ermöglicht es, eine Vorlage, die in der Antwort-Sprache verfasst ist, zu verwenden. Eine solche Vorlage enthält Platzhalter für die GUI-Elemente der Anwendung. Das Look And Feel ersetzt die Platzhalter durch den Code für das jeweilige GUI-Element.

Sobald die Antwort fertig generiert ist, wird sie an den anfragenden Rechner zurückgeschickt.

Anhang A.7 enthält ein einfaches Beispiel und eines, das den Einsatz einer Vorlage zeigt.

3.9.1 Bewertung

wingS eignet sich für die Entwicklung von Prototypen, kleinen Anwendungen und Anwendungen, die in einem Intranet laufen.

Für die Entwicklung von Prototypen ist das Positive an wingS, dass diese sehr schnell entwickelt werden können. Jeder Entwickler, der Desktop-Anwendungen entwickelt hat, kann mühelos eine wingS-Anwendung entwickeln. Die gleichen Elemente und die gleiche Art, eine Anwendung zu programmieren, finden sich auch bei wingS.

Kleine Anwendungen lassen sich auch sehr gut mit wingS entwickeln. Bei kleinen Anwendungen kann ignoriert werden, dass wingS-Anwendungen langsamer sind als zum Beispiel mit Struts entwickelte Web-Anwendungen. Das Gleiche gilt auch für Anwendungen im Intranet. Im Intranet wird der Performance-Verlust wahrscheinlich sogar gar nicht auffallen, da die Verbindung zwischen Rechnern in einem Intranet für gewöhnlich schneller ist als die ins Internet.

Ein Nachteil bei wingS ist jedoch, dass aus Prototypen irgendwann Produkte werden, dass kleine Anwendungen über die Zeit groß werden und Intranet-Anwendungen vielleicht doch irgendwann einmal vom Internet aus zugreifbar sein müssen, weil die Firma, die die Anwendung betreibt, Mitarbeiter im Außendienst beschäftigt.

Auf den ersten Blick sieht es so aus als könnte man Web-Anwendungen mit wingS auf die gleiche Weise entwickeln wie Desktop Anwendungen mit Swing. Sobald die Anwendung komplexer wird, wird jedoch klar, dass dies nur für sehr einfache Anwendungen gilt. Obwohl die wingS-GUI-Elemente die gleichen Ereignisse anbieten, die auch in Swing verfügbar sind, werden sie nicht so häufig ausgelöst wie in einer Swing-Anwendung. So kann man in einer Swing-Anwendung einen Listener für ein Ereignis anmelden, das jedes Mal ausgelöst wird, wenn der Benutzer eine Taste drückt während sich der Cursor in einem Texteingabefeld befindet. Dieses Ereignis ist auch in einer wingS-Anwendung verfügbar, aber es wird nur ausgelöst nachdem der Benutzer ein Formular mit einem Klick auf einen Knopf an den Server geschickt hat. In einer wingS-Anwendung kann man dieses Ereignis also nicht nutzen, um dem Benutzer während der Eingabe ein Feedback darüber zu geben, ob das, was er bisher eingegeben hat, ein valider Wert ist.

wingS erreicht sein Ziel, die Programmierung von Web-Anwendungen so einfach wie die Programmierung von Desktop-Anwendungen zu machen, nicht, denn in entscheidenden Aspekten kann auch wingS die Einschränkungen nicht verbergen, denen Web-Anwendungen unterworfen sind.

[35] CSS = Cascading Style Sheets; ein w3c-Standard (siehe auch [w3c-website]) zur Beschreibung des Layouts von HTML-Elementen

Ein positiver Aspekt von wingS, der auch für die Entwicklung von Web-Anwendungen ohne wingS interessant ist, ist die Verwendung von Platzhaltern für Dateneingabewidgets in den sonst statischen HTML-Seiten, die die Präsentation einer Web-Anwendung ausmachen.

Dies ermöglicht eine stärkere Trennung der Präsentation vom sonstigen Teil einer Anwendung als sie bisher vorgenommen wurde (Siehe hierzu die Muster „Trennung von Funktion und Interaktion“ und „Trennung von Präsentation und Interaktion“ in ´[Züllighoven 98]). Web-Designer können die Präsentation ganz allein gestalten und müssen sich mit den Entwicklern der Anwendung nur darüber einigen, welche Felder ein Formular hat und wie diese heißen. Der Entwickler kann dann die GUI-Komponenten definieren, die an deren Plätzen eingesetzt werden. Er kann die GUI-Elemente auch jederzeit austauschen, die für die Dateneingabe verwendet werden, ohne dass der Web-Designer das erfahren muss.

Eine weitere Anforderung an eine Lösung der in dieser Arbeit behandelten Probleme ist

> Web-Anwendungen sollen in den statischen HTML-Seiten, die ihre Präsentation bilden Platzhalter für Dateneingabewidgets verwenden, denn das Wissen darüber, welche Widgets für das Eingeben bestimmter Daten benötigt werden, gehört zur Anwendungslogik. Die Platzhalter sollen zur Laufzeit durch die benötigten Widgets ersetzt werden.

3.10 JavaServer Faces

JavaServer Faces oder auch kurz JSF genannt, ist ein neuer Standard, der sich zurzeit in der Entwicklung befindet. Eine erste Vorschauversion sowohl der Spezifikation als auch einer Referenzimplementierung ist bereits vorhanden und bildet die Grundlage für die nun folgenden Betrachtungen (siehe [McClanahan 02]).

Im Mittelpunkt der JavaServer Faces steht die Definition eines Oberflächenkomponenten-Rahmenwerkes für Web-Anwendungen. Für den Anwendungs-Entwickler stellt sich jede Seite der Anwendung als eine Menge von **Oberflächenkomponenten**[36] dar, die den Widgets in einer Desktop-Anwendung ähneln.

Abbildung 3.14 zeigt vereinfacht die Verwendung der JavaServer Faces in einer Anwendung.

Der Lebenszyklus lässt sich grob in vier Phasen aufteilen (Die Spezifikation sieht eine andere Aufteilung des Lebenszyklus in Phasen vor. Diese wird in Anhang A.8 vorgestellt). Zwei dieser Phasen heben sich von den anderen ab, da es viele Stellen in den anderen beiden Phasen gibt, an denen in diese beiden Phasen gewechselt werden kann. Diese besonderen Phasen sind „Fehlererfassung“ und „Generierung der Antwort“.

In der „Fehlererfassung“ werden Fehler erfasst, die auftreten, während der Lebenszyklus durchlaufen wird. Ein Objekt, das einmal pro Session existiert – der `FacesContext` –, bekommt bei einem Fehler eine Fehlernachricht mit einer Beschreibung des Ortes, an dem der Fehler aufgetreten ist, mitgeteilt. Diese Fehlernachricht wird dann verwahrt bis sie irgendwann von einer interessierten Stelle abgeholt wird. Es gibt verschiedene Arten von Fehlern. Fehler können auftreten, wenn eine Oberflächenkomponente versucht aus den Anfrageparametern die zu ihr gehörenden zu lesen

[36] Hier wird der Begriff „Oberflächenkomponente“ in Anlehnung an den in der Spezifikation benutzten Begriff „User Interface Component“ verwendet. „Komponente“ wird hier nicht in dem Sinn verwendet, wie er in [Szyperski 98] für Software-Komponenten definiert wird, sondern mehr in dem ursprünglichen Sinn, wo eine Komponente einfach nur ein Teil eines Ganzen – hier einer Oberfläche – ist (siehe hierzu [Wahrig 00])

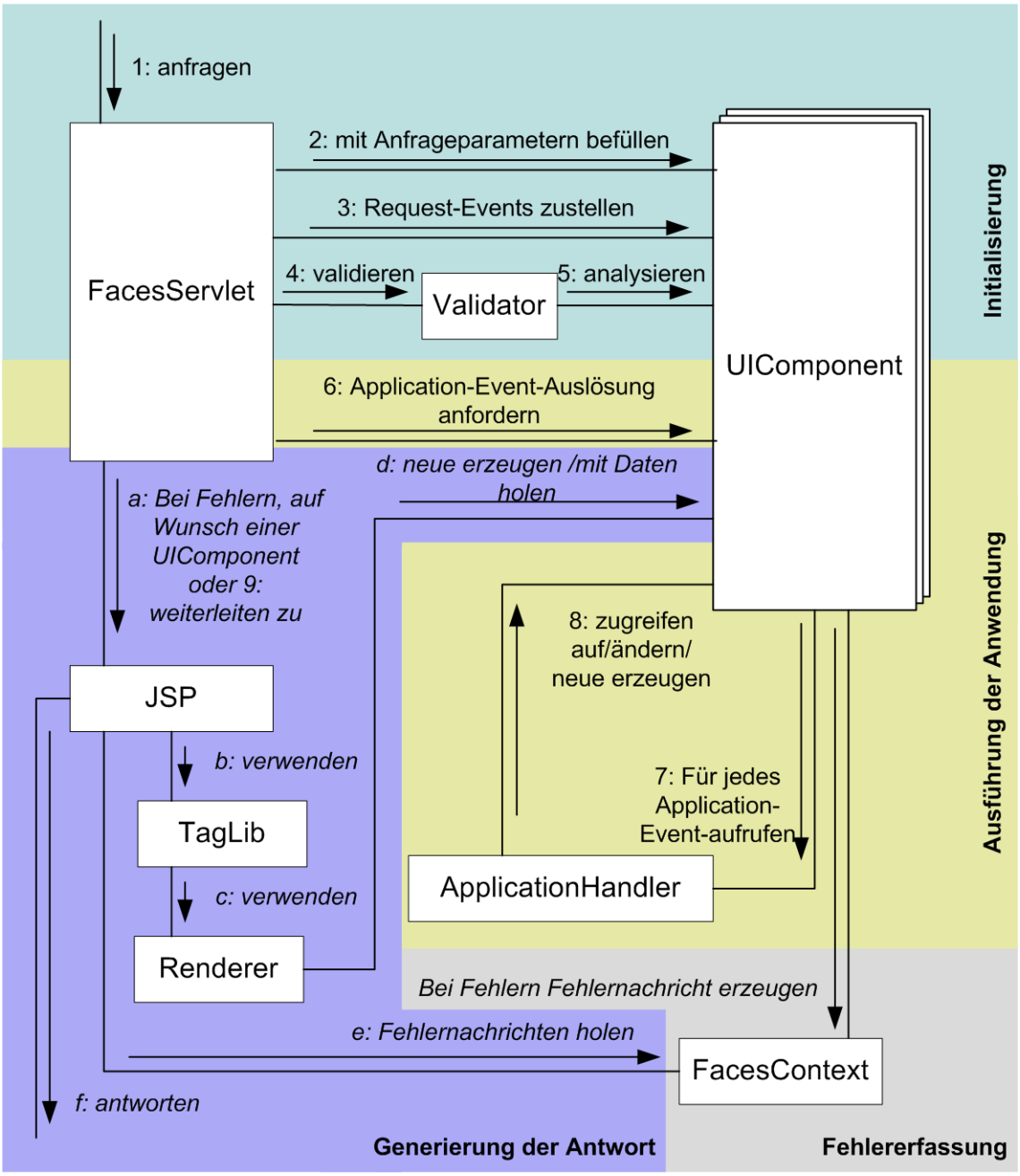

Abbildung 3.14 Vereinfachte Darstellung der Verwendung der JSF

und in etwas anderes als eine Zeichenkette zu konvertieren. So kann es vorkommen, dass in eine Oberflächenkomponente, die für die Eingabe von Datumsangaben vorgesehen ist, etwas anderes eingegeben wird. Fehler können auch bei der Validierung der Daten oder der Ausführung der Anwendungslogik auftreten.

In die ,,Generierung der Antwort“ wird aus verschiedenen Gründen gewechselt. Der natürlichste ist, dass die Anfrage bearbeitet wurde, und die Oberflächenkomponenten mit den Werten für die

Antwort befüllt wurden. Ein anderer Grund ist, dass ein Fehler aufgetreten ist, der dem Benutzer mitgeteilt werden muss. Es kann aber auch sein, dass eine Seite erneut angezeigt werden soll, weil sich an ihr nichts als die Anzeige einer Oberflächenkomponente geändert hat. Wenn der Benutzer auf einen Knoten in einer Baumansicht klickt, muss die Ansicht dieser Oberflächenkomponente angepasst werden, aber es besteht kein Bedarf den ganzen Weg inklusive des Aufrufs der Anwendungslogik zu gehen. In diesem Fall wird auch ohne Verzögerung in die Generierung der Antwort gewechselt.

Die Generierung der Antwort fängt damit an, dass die Kontrolle an eine durch die Anwendungslogik festgelegte JSP übergeben wird. Die JSP enthält die statische Struktur, die in HTML verfasst ist, und verwendet JSF-TagLibs, die den HTML-Code für die Anzeige der Oberflächenkomponenten, erzeugen. Die Oberflächenkomponenten selbst erinnern an Interaktionsformen (Siehe hierzu [Sturm 98]). Es gibt für jede Interaktionsart eine Oberflächenkomponente. Es gibt eine für das Auslösen einer Aktion, eine für die Eingabe eines Textes, eine für die Auflistung von Werten und so weiter. Für jede dieser Oberflächenkomponenten gibt es eine oder mehrere Arten, wie sie auf einer Seite dargestellt werden können. Die Oberflächenkomponente für das Auslösen einer Aktion kann zum Beispiel als Knopf oder als Verweis dargestellt werden. Für jede Möglichkeit, eine Oberflächenkomponente darzustellen gibt es einen Tag in der TagLib. Dieses Tag verwendet einen Renderer, der zu einer Oberflächenkomponente den HTML-Code erzeugen kann. Das Renderer-Konzept ermöglicht es – wie schon bei wingS (siehe Abschnitt 3.9) – Renderer für andere Web-Sprachen zu konstruieren. Damit wird eine Web-Anwendung davon unabhängig, welche Dokumentenbeschreibungssprache auf der Präsentationsebene verwendet wird. Eine Anwendung lässt sich auf eine andere Web-Plattform (zum Beispiel für Mobil-Telefone) portieren ohne dass die Anwendungslogik davon betroffen ist.

Soll nicht die JSP angezeigt werden, von der die Anfrage ausging, sondern eine neue, so kann es sein, dass es die entsprechenden Oberflächenkomponenten nicht gibt. Ist das der Fall, werden sie erzeugt und dann von den Renderern zum Erzeugen des Codes verwendet.

In der JSP können nicht nur die Tags für die spezifischen Darstellungsformen der Oberflächenkomponenten verwendet werden, sondern auch Tags, die angeben, wie die Inhalte der Oberflächenkomponenten validiert werden sollen.

Es gibt auch JSF-Tags, die die Fehlernachrichten aus dem `FacesContext` holen und anzeigen können.

Wie bei jeder Web-Anwendung fängt auch der Lebenszyklus einer JSF-Anwendung mit dem Eintreffen einer Anfrage an. Anfragen werden vom `FacesContext`, dem einzigen Servlet in einer JSF-Anwendung, empfangen. Der Empfang einer Anfrage läutet den Beginn der Phase ,,Initialisierung“ ein. Zuerst wird jede Oberflächenkomponente aufgerufen und aufgefordert sich aus den Anfrageparametern mit zu ihr gehörenden zu befüllen. Jede Oberflächenkomponente hat einen eindeutigen Namen über den sie diese Parameter heraussuchen kann. Einfache Oberflächenkomponenten erhalten maximal einen Anfrageparameter da sie nur ein Widget wie zum Beispiel einen Knopf oder ein Eingabefeld haben. Komplexere Oberflächenkomponenten, die sich aus mehreren Oberflächenkomponenten zusammensetzen, erhalten mehrere Anfrageparameter. Da nur die Oberflächenkomponenten die Bedeutung ihrer Anfrageparameter kennen, können auch nur sie diese Anfrageparameter richtig verarbeiten.

Sobald alle Oberflächenkomponenten ihre Anfrageparameter verarbeitet haben, werden die ,,Request Events“ zugestellt. Einige Request Events betreffen die Anzeige der Oberflächenkomponente. Ein Beispiel dafür ist das Klicken auf einen Knoten in einer Baumansicht. Solche Ereignisse werden von der Oberflächenkomponente direkt verarbeitet und nach der Verarbeitung wird direkt in die Generierung der Antwort gewechselt, um die Änderung der Anzeige wirksam zu machen. Andere Request Event werden von den Oberflächenkomponenten in Application Events umgewandelt, die

bei der Ausführung der Anwendungslogik zugestellt werden. Ein Ereignis dieser Art wird zum Beispiel ausgelöst, wenn der Benutzer auf einen Knopf oder Verweis klickt.

Zum Abschluss der Initialisierungs-Phase werden die Validatoren, die für einzelne Oberflächenkomponenten angemeldet sind, ausgeführt. Wie in FormProc (siehe Abschnitt 3.6) und beim Struts-Validator (siehe Abschnitt 3.8) gibt es auch in der Referenzimplementierung eine Reihe von vordefinierten Validatoren, die über Tags aus der TagLib für die einzelnen Oberflächenkomponenten in den JSPs konfiguriert werden können. Es besteht auch die Möglichkeit, eigene Validatoren zu entwickeln. Ein Validator holt sich den Inhalt aus der Oberflächenkomponente, der er zugeordnet ist, und validiert diesen. Tritt hier ein Fehler auf, wird in die Fehlererfassung und anschließend in die Generierung der Antwort gewechselt.

Nach der Initialisierung wird in die Phase ,,Ausführung der Anwendung“ gewechselt. Hier kommt die Anwendungslogik zum Einsatz. Alle Oberflächenkomponenten stellen dem `ApplicationHandler` ihre Ereignisse zu, die von diesem verarbeitet werden. Der `ApplicationHandler` aktualisiert bestehende Oberflächenkomponenten, löscht nicht mehr benötigte und erzeugt neue Oberflächenkomponenten. Sind alle Ereignisse abgearbeitet, wird in die Generierung der Antwort gewechselt.

3.10.1 Bewertung

JavaServer Faces bietet etwas, womit keine der bisher vorgestellten Web-Technologien aufwarten konnte. Die Anwendungslogik geht bei JSF nur mit Objekten um. ,,HTML“ und ,,Anfrageparameter“ sind Begriffe, die hier keine Rolle spielen. Damit ähnelt JSF auf den ersten Blick wingS (siehe Abschnitt 3.9). JSF unterscheidet sich jedoch in dem wichtigen Punkt, dass es nicht versucht, vor dem Entwickler zu verbergen, dass er eine Web-Anwendung entwickelt. Entwickler, die eine Anwendung mit JSF entwickeln sind sich bewusst, dass sie eine Web-Anwendung entwickeln, denn bei JSF spielt der Begriff ,,Seite“ eine Rolle und Oberflächenkomponenten bieten nur Ereignisse an, die in einer Web-Anwendung Sinn machen und im vollen Maße unterstützt werden können.

JSF abstrahiert an den richtigen Stellen. Oberflächenkomponenten nehmen dem Entwickler die Last ab, sich um die Anfrageparameter, die mit einer Anfrage kommen, zu kümmern. Sie sorgen auch dafür, dass die richtigen Widgets in jeder generierten Antwort erscheinen.

Ein Nachteil bei JSF ist, dass die Tags, die hier verwendet werden, konkrete Widgets für die Dateneingabe angeben. Damit werden die Anwendungslogik und die Präsentation stärker aneinandergebunden als notwendig. Die Betrachtung von wingS (Siehe hierzu Abschnitt 3.9) hat gezeigt, dass eine Verwendung von Vorlagen mit Platzhaltern, die zur Laufzeit durch die benötigten Widgets ersetzt werden, vorteilhafter ist.

Ein weiterer Nachteil ist, dass die Validierungsregeln in den JSPs über spezielle Validierer-Tags angegeben werden. Auch das koppelt die Anwendungslogik und die Präsentation stärker aneinander als notwendig. Die Betrachtung von FormProc (Siehe hierzu Abschnitt 3.6) und Struts-Validator (siehe Abschnitt 3.8) hat gezeigt, dass eine separate Konfiguration hier die gewünschte Trennung bietet.

Die Möglichkeit, eigene Oberflächenkomponenten zu entwickeln, birgt zwei Vorteile für die Entwicklung von Web-Anwendungen:

- Ermöglicht Konstruktion von Oberflächenkomponenten für das Erfassen von Fachwerten, da JSF-Oberflächenkomponenten von den Widgets, die für ihre HTML-Präsentation verwendet

werden, wegabstrahieren und sowohl die Generierung dieser Präsentation wie auch die Validierung der eingegebenen Daten und Konvertierung in andere Datentypen in einer Einheit zusammenführen.

- Möglichkeit Fachwerte konsequent in der Anwendung einzusetzen, da die Oberflächenkomponenten Eingaben mit den Fachwert-Fabriken validieren und Fachwerte liefern können

Eine Anforderung an die Lösung der in dieser Arbeit behandelten Probleme ist also

> Web-Anwendungen sollen Fachwerte verwenden. Die Verwendung von Fachwerten ist eine Maßnahme in die Richtung der Herstellung und Sicherstellung von Validität.

Eine weitere Anforderung ist

> Die Oberflächenkomponenten (oder allgemeiner Widgets), die für das Erfassen von Fachwerten verwendet werden, sollen an einer zentralen Stelle verwaltet werden. Da dadurch für das Eingeben von Fachwerten eines bestimmten Typs immer die gleiche Art von Oberflächenkomponente verwendet wird, wird hiermit für Konsistenz in der Anwendung gesorgt.

3.11 Zusammenfassung

In diesem Kapitel wurden Web-Technologien und deren Unterstützung für die Herstellung von Konsistenz und Validität vorgestellt. Angefangen wurde mit der Grundtechnologie für Web-Anwendungen, der Dokumentenbeschreibungssprache HTML.

HTML – Hypertext Markup Language – ist die Sprache, in der nahezu alle im Web verfügbaren Dokumente geschrieben sind. Diese Dokumente werden typischerweise über das HTTP – Hypertext Transfer Protocol – von entfernten Rechner angefordert und empfangen. HTML bot in den ersten Versionen, in denen sie für den Austausch von wissenschaftlichen Dokumenten gedacht war, nur die Möglichkeit, die Struktur des Dokumentes festzulegen. Mittlerweile ist die Ausdrucksfähigkeit in HTML stark gewachsen, so dass eine genaueres Layout der Dokumente möglich ist.

HTML bietet zwei Arten der Interaktion an. Einerseits kann man auf Verweise zu anderen Seiten klicken und bekommt dann das jeweilige Dokument zu sehen. Andererseits kann man auch Formulare verwenden und hat mit diesen mehr als nur die Möglichkeit in andere HTML-Dokumente zu wechseln. Mit **HTML-Formularen** lassen sich Web-Anwendungen konstruieren, die zum Teil den Anwendungen aus der Desktop-Welt ähneln.

Bei den HTML-Formularen wurde die Unterstützung für die Herstellung von Validität untersucht, und es wurden zwei Arten entdeckt. Die **verzögerte Validierung** kommt dann zum Einsatz, wenn ein Formular nach dem Abschicken auf dem Serverrechner validiert wird. Hierfür kommen in Perl oder C geschriebene CGI-Programme zum Einsatz. Wenn die Validierung fehlschlägt, zeigt die Anwendung eine Fehlerseite an, die beschreibt, welche Felder fehlerhaft ausgefüllt wurden, oder die Anwendung zeigt die Ausgangsseite an und fügt dort an einer Stelle oder jeweils neben den betreffenden Feldern die Fehlermeldungen ein. Das letztere ist hierbei vorzuziehen, da es nicht zu viel Konzentrationsvermögen vom Benutzer abverlangt.

Die zweite Art der Validierung ist die **sofortige Validierung**. Gibt der Benutzer hierbei einen Wert in ein Feld ein und wechselt in ein anderes, dann wird der Inhalt des Feldes sofort validiert und ein Fehler wird gleich darauf durch eine entsprechende Meldung oder ein erscheinendes Symbol neben dem betreffenden Feld angezeigt. Diese Art der Validierung hat den Vorteil, dass der Benutzer nicht das ganze Formular ausfüllen und abschicken muss, um zu sehen, ob er irgendwo einen Fehler begangen hat, sondern, dass er gleich nach der Eingabe eine Rückkopplung über die Richtigkeit der Eingabe erhält. Der Nachteil ist, das JavaScript nicht standardisiert ist und verschiedene Browserarten und -versionen verschiedene JavaScript-Versionen unterstützen. Außerdem steht es dem Benutzer frei, die Ausführung von JavaScript-Funktionen abzuschalten. Ein weiterer Nachteil ist, dass die Größe der Seiten mit der Komplexität der Prüfungsalgorithmen zunimmt, bis es an einem bestimmten Punkt nicht mehr für den Benutzer zumutbar ist, auf das Herunterladen einer solchen Seite vom entfernten Serverrechner und deren Aufbau auf dem lokalen Rechner zu warten. Eine wohlüberlegte Mischung aus verzögerter und sofortiger Validierung scheint die ideale Lösung zu sein.

Bei der Betrachtung der Validierungsmöglichkeiten bei HTML-Formularen wurde folgende Anforderung an eine Lösung der in dieser Arbeit behandelten Probleme aufgestellt

> Web-Anwendungen sollen bei der Validierung von Daten ein zweistufiges Verfahren einsetzen, das sich aus einer Vor-Validierung auf dem Rechner des Benutzers und einer Haupt-Validierung auf dem Server zusammensetzt. Die Vor-Validierung soll einfach und generisch sein. Die Hauptvalidierung soll so akkurat wie möglich sein. Es kann vorkommen, dass die Haupt-Validierung Fehler entdeckt, die die Vor-Validierung nicht entdeckt hat.

Hierauf wurde das **Servlet API** und die **JavaServer Pages** vorgestellt. Das Servlet API ist eine Java-Klassenbibliothek, die die Programmierung von Web-Anwendungen gegenüber der Programmierung von CGI-Programmen vereinfacht. Zur Programmierung von Web-Anwendungen mit diesen Technologien ist fast kein Wissen über HTTP nötig. Es muss lediglich eine Klasse implementiert werden, die eine Operation enthält. Die Anfrageparameter werden in einem Anfrageobjekt an diese Operation übergeben und können mit aus Desktop-Anwendungen bereits bekannten Mitteln ausgelesen werden. Für die Ausgabe steht ebenfalls ein eigenes Objekt zur Verfügung, in das die resultierende HTML-Seite geschrieben werden kann.

Obwohl das Servlet API die Programmierung von Web-Anwendungen drastisch vereinfacht, hat das Schreiben der Antwort-HTML-Seite in einem Servlet auch seine Nachteile. So kann die HTML-Seite hierbei nicht von einem Web-Designer entwickelt werden, denn Web-Designer haben normalerweise kein Wissen über Java, und können somit auch keine Servlets entwickeln. Außerdem können hier keine HTML-Editoren zum Einsatz kommen, denn bei einem Servlet handelt es sich um eine Java-Klasse, die ein HTML-Dokument generiert.

Als Reaktion auf diese Probleme wurde die Technologie der JavaServer Pages eingeführt. Bei JavaServer Pages handelt es sich um HTML-Seiten mit eingestreutem Java-Code, weshalb sie auch mit HTML-Editoren bearbeitet werden können. Web-Designer können den statischen Teil einer JSP gestalten und Java-Entwickler können dann den dynamischen Java-Teil einfügen.

Bereits in der ersten JSP-Spezifikation wurden zwei Architekturen für Web-Anwendungen mit JSPs vorgestellt. Bei der ersten, der **Model 1-Architektur**, greifen die JSPs über JavaBeans auf die Geschäftsdaten zu. Obwohl hier der Zugriff über JavaBeans erfolgt, ist der Zugriff auf die Daten so gut wie direkt und die Anwendungslogik liegt in den JSPs. Eine bessere aber auch komplexere Architektur ist die **Model-2-Architektur**. Diese ist durch die bei Desktop-Anwendungen

verwendete MVC-Architektur inspiriert. Die JSP Seiten stellen bei der Model-2-Architektur das View dar. Ein oder mehrere Servlets, die Controller, vermitteln zwischen den JSPs und der Anwendungslogik . Hier ist eine sehr viel stärkere Trennung von Funktionalität und Präsentation vorzufinden als dies bei der Model-1-Architektur der Fall ist. Model 2-Architekturen sind deshalb den Model-1-Architekturen vorzuziehen.
Eine Anforderung an die Lösung, die bei dieser Betrachtung aufgestellt wurde, ist

> Web-Anwendungen sollen eine Model 2-Architektur verwenden, um eine Trennung zwischen Präsentation und Anwendungslogik zu erreichen.

Nach der Vorstellung der Servlet-API und der JavaServer Pages wurde **FormProc**, eine API zur Validierung von HTML-Formularen, betrachtet. FormProc ermöglicht es, für die Widgets der in einer Anwendung verwendeten HTML-Formulare Validatoren zu konfigurieren, die die Daten in den Widgets validieren. FormProc ist ein Schritt in Richtung Validität in Web-Anwendungen. Das größte Problem FormProcs ist, dass nur auf HTML-Widget-Ebene validiert werden kann. Soll zum Beispiel ein Datum mit drei Eingabefeldern erfasst werden – das erste für den Tag, das zweite für den Monat und das dritte für das Jahr –, so können diese nur einzeln für sich validiert werden. Es kann dann zum Beispiel nur validiert werden, ob im Eingabefeld für den Tag eine Zahl zwischen 1 und 31 steht. Es kann in diesem Fall jedoch nicht validiert werden, dass die drei Eingabefelder zusammen ein valides Datum enthalten. Die Erkenntnis dieses Problems führte zur Aufstellung der folgenden Anforderung

> Ein Formular besteht aus Feldern, die Daten aufnehmen können. Wird ein Formular angezeigt, so kann es sein, dass mehrere Widgets für ein Feld verwendet werden. Die Validierung eines Formulars muss die Felder des Formulars validieren und nicht die einzelnen Widgets.

Ein weiteres Problem bei FormProc ist, dass zur Verwendung von FormProc Java-Code in die JSPs geschrieben werden muss. FormProc bietet keine TagLib an, die dies verhindert. Dies führte zur Aufstellung der folgenden Anforderung

> Eine Lösung soll eine JSP-TagLib anbieten, um zu verhindern, dass Java-Code für die Verwendung der Lösung in JSPs eingestreut werden muss.

Struts ist ein Rahmenwerk für die Entwicklung von Web-Anwendungen mit Model-2-Architekturen. Bei Struts gibt es ein Servlet – das `ActionServlet` –, das die Anfragen empfängt und die Anfrageparameter in das Model der Anwendung – die `ActionForm`s füllt. Die `Action`, die nach der Befüllung ihrer `ActionForm` aufgerufen wird, enthält die Anwendungslogik der Anwendung. Sie liest die `ActionForm` und liest und verändert als Reaktion auf die Anfrage die Geschäftsdaten, die sich in einer Datenbank befinden. Zuletzt füllt die `Action` eine `ActionForm` für die Antwort und die nächste JSP wird aufgerufen. Diese verwendet Struts-TagLibs, um auf die Daten der Antwort-`ActionForm` zuzugreifen, und generiert die Antwort, die zum anfragenden Rechner zurückgeschickt wird.

Struts größtes Problem ist, dass die `Action` sehr feingranular sind. Normalerweise wird für jede Anfragemöglichkeit - jeden Knopf oder Verweis in einer Web-Anwendung - eine eigene `Action` implementiert. Das führt zu einer Fragmentierung der Anwendungslogik, die nur selten fachlich motiviert ist. Eine bessere Idee ist es, `Action`s nur als Vermittler zwischen der Web-Anwendung und fachlichen Dienstleistern einzusetzen, die die fachliche Anwendungslogik enthalten. Diese Dienstleister können nach fachlichen Gesichtspunkten konstruiert werden und müssen auch nichts über die Benutzungsschnittstellen wissen, über die sie verwendet werden. Damit folgt folgende Anforderung

> Web-Anwendungen sollen fachliche Dienstleister verwenden und somit eine Trennung zwischen der fachlichen Logik und der Logik erreichen, die sich ausschließlich mit den speziellen Anforderungen der Web-Anwendung befasst.

Der Struts-Validator ist ein Zusatz zu Struts, der die Definition von Validatoren für die Daten in den `ActionForm`s ermöglicht. Seine Funktionsweise ist sehr ähnlich zu der von FormProc. Sein Vorteil ist die gute Integration in Struts. Sein Nachteil ist – wie bei FormProc –, dass er nur auf Widget-Ebene validieren kann, denn die `ActionForm`s werden mit den Daten aus den einzelnen Widgets befüllt und erlauben keine Gruppierung von Widgets zu größeren Einheiten.

Das Rahmenwerk **wingS** geht einen ganz anderen Weg. wingS ist ein **Swing für Web-Anwendungen**. Web-Anwendungen lassen sich beinahe so wie Desktop-Anwendungen schreiben, die das GUI-Toolkit Swing verwenden. Die Klassen fangen hier statt mit einem „J“ mit einem „S“ an. Für die Swing-Klasse `JButton` gibt es zum Beispiel ein Gegenstück in wingS mit dem Namen `SButton`. Die wingS-Klassen haben fast die gleichen Schnittstellen wie die von Swing. Lediglich in einigen Fällen, in denen sich nicht verbergen lässt, dass wingS-Anwendungen Web-Anwendungen und nicht Desktop-Anwendungen sind, unterscheiden sich die wingS-Klassen von ihren Swing-Gegenstücken. Mit wingS sollen auch Entwickler, die noch keine Web-Anwendungen geschrieben haben, Web-Anwendungen entwickeln können.

Tatsächlich ist wingS gut für die Prototypenentwicklung von Web-Anwendungen geeignet, da die Entwicklung einer einfachen Anwendung schnell zu erledigen ist. Soll jedoch aus einem Prototypen eine richtige Web-Anwendung werden, wird meistens ein Web-Designer hinzugezogen werden müssen. Auch dafür bietet wingS Unterstützung, denn der Web-Designer kann sein Layout ohne Beschränkungen entwickeln und Plätze für die dynamischen Oberflächenelemente ausweisen, die dann von wingS unter Zuhilfenahme eines dafür vorgesehenen LayoutManagers gefüllt werden können.

Ein Problem von wingS ist, dass Web-Anwendungen Beschränkungen unterworfen sind, die bei Desktop-Anwendungen nicht existieren. So kehrt der Kontrollfluss nur dann zum Server-Teil der Anwendung zurück, wenn der Benutzer auf einen Knopf oder einen Verweis klickt. Entwickler von Desktop-Anwendungen sind es gewohnt, auf jeden Tastendruck und jede Mausbewegung reagieren zu können. Diese Kontrolle kann wingS den Entwicklern nicht geben und scheitert deshalb, die Entwicklung von Web-Anwendungen so einfach wie die Entwicklung von Desktop-Anwendungen zu machen.

Ein Vorteil von wingS ist jedoch die erwähnte Verwendung von Vorlagen mit Platzhaltern für Widgets. Dieses Konzept ist auch für die Entwicklung von Web-Anwendung ohne wingS interessant, weshalb daraus eine Anforderung an die Lösung der in dieser Arbeit behandelten Probleme folgt

Web-Anwendungen sollen in den statischen HTML-Seiten, die ihre Präsentation bilden, Platzhalter für Dateneingabewidgets verwenden, denn das Wissen darüber, welche Widgets für das Eingeben bestimmter Daten benötigt werden, gehört zur Anwendungslogik. Die Platzhalter sollen zur Laufzeit durch die benötigten Widgets ersetzt werden.

JavaServer Faces (kurz JSF) ist die letzte Web-Technologie, die in diesem Kapitel vorgestellt wurde. Einordnen kann man JavaServer Faces zwischen Struts und wingS. JSF beinhaltet eine Architektur, die Model-2-konform ist, obwohl es hier nicht das Hauptziel ist, ein Model-2-Rahmenwerk zu schaffen. JSF konzentriert sich sehr stark auf den View-Bereich. Eine Bibliothek von Oberflächenelementen stellt das Herzstück der JavaServer Faces dar. Die Entwicklung von Web-Anwendungen mit JSF teilt sich in zwei Teile. Zum einen entwickeln Web-Designer die JSPs, in denen sie unter anderem Tags verwenden, die zu den JavaServer Faces gehören. Zum anderen entwickeln die Anwendungsentwickler die Anwendungslogik. Ohne zu wissen, wie die JSPs aussehen, greifen sie auf Oberflächenkomponenten zu, die abstrakte Arten von Widgets vertreten. So gibt es ein Oberflächenelement `UICommand`, das überall dort verwendet wird, wo eine Aktion ausgelöst werden kann. Dabei ist es für den Entwickler der Anwendungslogik unerheblich, wie ein solcher Aktionsauslöser konkret aussehen wird. Das kann ein Verweis, ein Knopf oder noch etwas anderes sein. Wichtig ist für diesen Entwickler nur, dass eine Aktion ausgelöst werden kann und die Anwendungslogik darauf reagieren muss. JavaServer Faces zielt also auf eine Trennung von Handhabung und Präsentation ab. Die Entwickler der Anwendungslogik bekommen eine objektorientierte Schnittstelle zu den JSPs, während die Web-Designer mit HTML und einigen zusätzlichen Tags wie gehabt die Präsentation der Anwendung entwickeln. Die objektorientierte Schnittstelle, die hier dem Entwickler der Anwendungen geboten wird, funktioniert in beide Richtungen. Wenn es zum Beispiel eine spezialisierte Oberflächenkomponente gibt, die für das Eingeben eines Datums gedacht ist, dann kann der Entwickler an diese Komponente ein Datum übergeben. Diese Komponente wird beim Anzeigen der Seite das Datum richtig anzeigt. Wenn ein Benutzer ein anderes Datum in die Oberflächenkomponente eingibt, wird diese die Eingabe bei der nächsten Anfrage in ein Datum-Objekt konvertieren, mit dem die Anwendung arbeiten kann. Der Entwickler muss sich also nicht damit befassen, welche Widgets im HTML-Formular verwendet werden, um die Daten aus einer Anfrage richtig verarbeiten zu können.

Ein Nachteil der JavaServer Faces ist, dass in den JSPs Tags verwendet werden, die angeben welche Oberflächenkomponenten verwendet werden und wie diese dargestellt werden sollen. Eine Verwendung von Vorlagen wie bei wingS würde hier eine weitere sinnvolle Trennung der Anwendungslogik von der Präsentation bringen.

Ein weiterer Nachteil ist die Verwendung von Validator-Tags, mit denen angegeben wird, wie eine bestimmte Oberflächenkomponente validiert werden soll. Werden diese Tags verwendet, so enthält die Präsentation einen Teil der Anwendungslogik, denn das Wissen über die Art der Daten, die erfasst werden, ist Teil der Anwendungslogik.

Die Möglichkeit, eigene Oberflächenkomponenten zu entwickeln, birgt zwei Vorteile für die Entwicklung von Web-Anwendungen:

- Ermöglicht Konstruktion von Oberflächenkomponenten für das Erfassen von Fachwerten, da JSF-Oberflächenkomponenten von den Widgets, die für ihre HTML-Präsentation verwendet

werden, wegabstrahieren und sowohl die Generierung dieser Präsentation wie auch die Validierung der eingegebenen Daten und Konvertierung in andere Datentypen in einer Einheit zusammenführen.

- Möglichkeit Fachwerte konsequent in der Anwendung einzusetzen, da die Oberflächenkomponenten Eingaben mit den Fachwert-Fabriken validieren und Fachwerte liefern können

Damit können folgende zwei Anforderungen an eine Lösung der in dieser Arbeit behandelten Probleme aufgestellt werden

Web-Anwendungen sollen Fachwerte verwenden. Die Verwendung von Fachwerten ist eine Maßnahme in die Richtung der Herstellung und Sicherstellung von Validität.

Die Oberflächenkomponenten (oder allgemeiner Widgets), die für das Erfassen von Fachwerten verwendet werden, sollen an einer zentralen Stelle verwaltet werden. Da dadurch für das Eingeben von Fachwerten eines bestimmten Typs immer die gleiche Art von Oberflächenkomponente verwendet wird, wird hiermit für Konsistenz in der Anwendung gesorgt.

4 Designalternativen

> *There's More Than One Way To Do It – That's the Perl Slogan, and you'll get tired of hearing it, unless you're the Local Expert, in which case you'll get tired of saying it. Sometimes it's shortened to TMTOWTDI, pronounced "tim-toady". But you can pronounce it however you like. After all, TMTOWTDI*
>
> — Larry Wall et al. : *Programming Perl*

In diesem Kapitel werden zunächst die Anforderung an eine Lösung der Probleme, die Thema dieser Arbeit sind, erneut aufgelistet. Nach der Auflistung der Anforderungen werden drei Designalternativen vorgestellt, von denen die Beste für eine Implementierung ausgewählt wird.

4.1 Anforderungen

Dieser Abschnitt listet die im vorigen Kapitel bei der Betrachtung der Web-Technologien und deren Unterstützung für die Herstellung von Konsistenz und Validität in Web-Anwendungen aufgestellten Anforderung an eine Lösung der Probleme, die Thema dieser Arbeit sind, auf. Jeder Anforderung wird eine Bezeichnung gegeben, über die sie im weiteren Verlauf der Arbeit referenziert wird.
Die Anforderung **Zweistufige Validierung ermöglichen** ist (siehe hierzu auch Abschnitt 3.3.3)

> Web-Anwendungen sollen bei der Validierung von Daten ein zweistufiges Verfahren einsetzen, das sich aus einer Vor-Validierung auf dem Rechner des Benutzers und einer Haupt-Validierung auf dem Server zusammensetzt. Die Vor-Validierung soll einfach und generisch sein. Die Hauptvalidierung soll so akkurat wie möglich sein. Es kann vorkommen, dass die Haupt-Validierung Fehler entdeckt, die die Vor-Validierung nicht entdeckt hat.

Die Anforderung **Model-2-Architektur verwenden** ist (siehe hierzu auch Abschnitt 3.5.1)

> Web-Anwendungen sollen eine Model 2-Architektur verwenden, um eine Trennung zwischen Präsentation und Anwendungslogik zu erreichen.

Die Anforderung **Trennung Feld und Anzeigeelement** ist (siehe hierzu auch Abschnitt 3.6.1)

> Ein Formular besteht aus Feldern, die Daten aufnehmen können. Wird ein Formular angezeigt, so kann es sein, dass mehrere Widgets für ein Feld verwendet werden. Die Validierung eines Formulars muss die Felder des Formulars validieren und nicht die einzelnen Widgets.

Die Anforderung **JSP-Taglib anbieten** ist (siehe hierzu auch Abschnitt 3.6.1)

> Eine Lösung soll eine JSP-TagLib anbieten, um zu verhindern, dass Java-Code für die Verwendung der Lösung in JSPs eingestreut werden muss.

Die Anforderung **Fachliche Dienstleister verwenden** ist (siehe hierzu auch Abschnitt 3.7.1)

> Web-Anwendungen sollen fachliche Dienstleister verwenden und somit eine Trennung zwischen der fachlichen Logik und der Logik erreichen, die sich ausschließlich mit den speziellen Anforderungen der Web-Anwendung befasst.

Die Anforderung **Platzhalter für Dateneingabe in HTML** ist (siehe hierzu auch Abschnitt 3.9.1)

> Web-Anwendungen sollen in den statischen HTML-Seiten, die ihre Präsentation bilden, Platzhalter für Dateneingabewidgets verwenden, denn das Wissen darüber, welche Widgets für das Eingeben bestimmter Daten benötigt werden, gehört zur Anwendungslogik. Die Platzhalter sollen zur Laufzeit durch die benötigten Widgets ersetzt werden.

Die Anforderung **Fachwerte verwenden** ist (siehe hierzu auch Abschnitt 3.10.1)

> Web-Anwendungen sollen Fachwerte verwenden. Die Verwendung von Fachwerten sind eine Maßnahme in die Richtung der Herstellung und Sicherstellung von Validität.

Die Anforderung **Zentrale Widgetverwaltung** ist (siehe hierzu auch Abschnitt 3.10.1)

> Die Oberflächenkomponenten (oder allgemeiner Widgets), die für das Erfassen von Fachwerten verwendet werden, sollen an einer zentralen Stelle verwaltet werden. Da dadurch für das Eingeben von Fachwerten eines bestimmten Typs immer die gleiche Art von Oberflächenkomponente verwendet wird, wird hiermit für Konsistenz in der Anwendung gesorgt.

4.2 Design auf Basis von Struts

Bei der Designalternative, die auf Struts (siehe Abschnitt 3.7) basiert, spielen zwei neue Komponenten eine tragende Rolle (Diese Designalternative ist in Abbildung 4.1 skizziert). Die **WebWidget-Fabrik** ist ein Singleton, an dem **WebWidgets** für Fachwerttypen registriert und angefordert werden können. Diese Fabrik ist mit der `PFFactory` beim JWAM-Formularwesen (siehe Abschnitt 2.4) vergleichbar.

Ein `WebWidget` kann eine HTML-Repräsentation von sich generieren. Die HTML-Repräsentation eines `WebWidget`s kann mehrere HTML-Widgets enthalten. Zwischen einem `WebWidget` und seinen HTML-Widgets besteht also keine 1-zu-1-Beziehung sondern eine 1-zu-n-Beziehung. Neben einfachen Widgets wie Eingabefelder und Aufklapplisten kann es also auch komplexere Widgets wie zum Beispiel ein Widget für die Eingabe eines Zeitpunktes geben, das aus mehreren Eingabefeldern besteht.

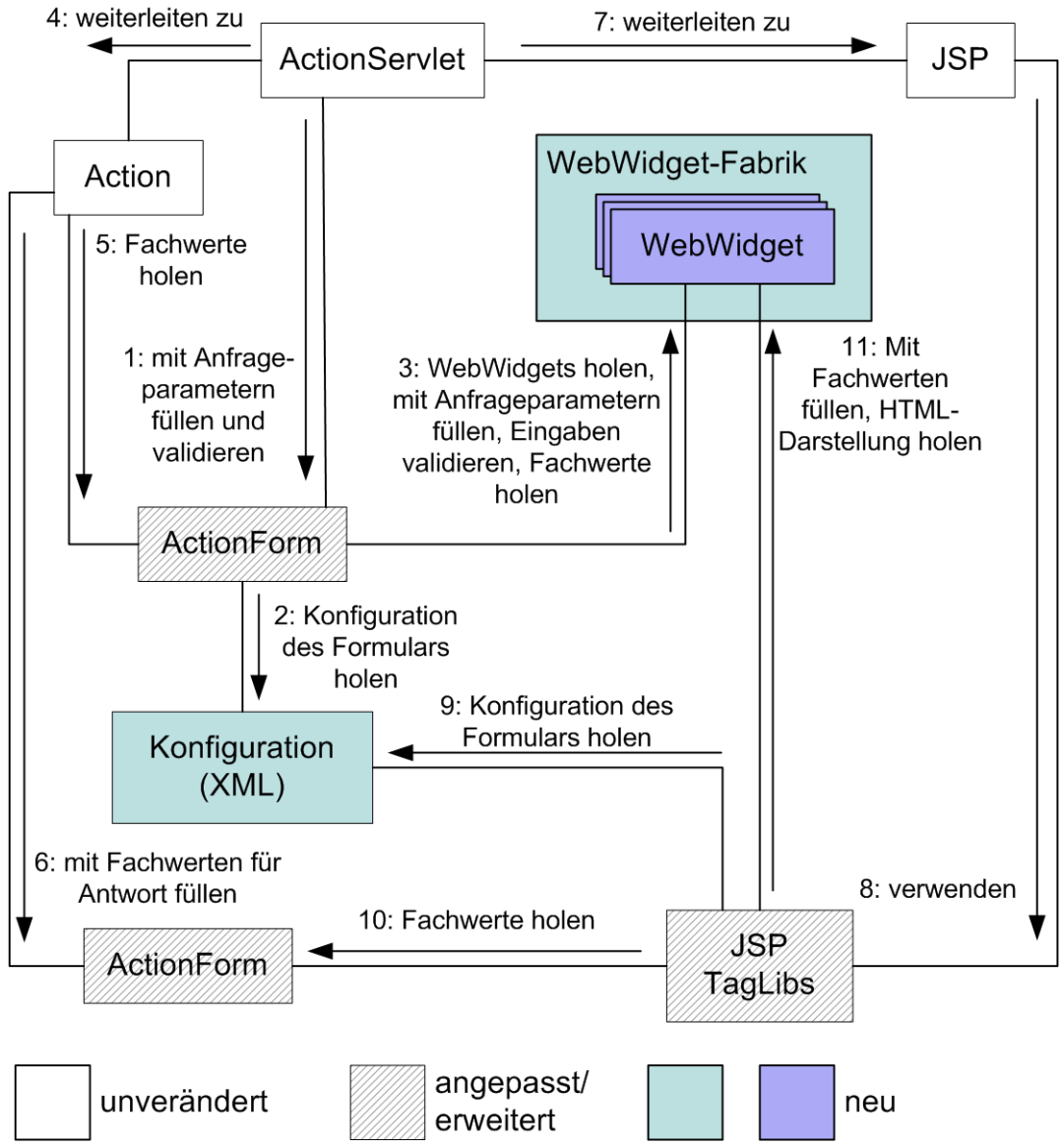

Abbildung 4.1 Designalternative auf Basis von Sturts

Ein `WebWidget` kann aus den Parametern einer Anfrage rekonstruieren, wie der Benutzer die HTML-Widgets, die zu ihm gehören, befüllt hat. Aus dieser Rekonstruktion kann es mit Hilfe der Fachwert-Fabrik feststellen, ob die Eingabe eine Repräsentation eines gültigen Fachwertes ist. Wird von einem `WebWidget` nach der Validierung einer Eingabe eine HTML-Repräsentation angefordert, so sorgt das `WebWidget` dafür, dass die Repräsentation eine entsprechende Markierung enthält, falls es mit einen ungültigen Wert befüllt ist.

`WebWidget`s können für einen bestimmten Fachwerttyp entwickelt werden, oder sie können generisch sein. Generische `WebWidget`s wie Eingabefelder, die eine beliebige Fachwert-Fabrik verwenden können, die Fachwerte aus Zeichenketten erzeugen kann, erkennen eine Untergruppe von Fachwert-Fabriken, die einen regulären Ausdruck für die Vor-Validierung einer Eingabe zurückgeben können.

Ist ein `WebWidget` mit einer solchen Fabrik ausgestattet, generiert es in die HTML-Präsentation den benötigten Code, um eine Vor-Validierung auf dem Rechner des Benutzers durchzuführen (siehe Abschnitt 3.3.2).

Die Designalternative benötigt eine neue Konfiguration, die wie die Konfiguration von Struts in einer XML-Datei enthalten ist. Die Konfiguration enthält eine Auflistung der verfügbaren `WebWidget`s und ordnet sie den Fachwerttypen zu. Das System lädt diese Konfiguration beim Starten und registriert die `WebWidget`s an der `WebWidget`-Fabrik. Darüber hinaus enthält die Konfiguration eine Auflistung aller in der Anwendung verwendeten Formulare und gibt zu jedem Formularfeld an, welchen Fachwerttyp dieses hat.

Die JSPs verwenden eine TagLib, die ein Tag enthält, das anstelle der Formularfelder in den JSPs verwendet wird. Dieses Tag kennt den Namen des Formularfeldes, für das es steht, und kann über das umschließende `form`-Tag herausfinden, zu welchem Formular das Feld gehört. Wenn eine JSP ausgeführt wird, die Exemplare dieses Tags verwendet, findet jedes dieser Exemplare über die Konfiguration heraus, welcher Fachwerttyp verwendet werden soll. Ist der Fachwerttype ermittelt, holt das Tag von der `WebWidget`-Fabrik das benötigte `WebWidget`, befüllt es mit den Daten aus einem eventuell existierenden Antwort-`ActionForm` und schreibt die HTML-Repräsentation des `WebWidget`s an seiner Stelle in das Antwort-HTML.

Weder die Verwendung von JavaBeans-Klassen, die von `ActionForm` erben, noch die Verwendung von `DynaActionForm`s passt mit dem hier beschriebenen System zusammen. Die zusätzliche Konfiguration enthält bereits die Definition der Formulare. Die Verwendung der beiden `ActionForm`-Arten würde zu Redundanzen führen. Eine JavaBeans-Implementierung müsste jeweils Attribute haben, die den Felder in der Formulardefinition entsprechen. Bei der Verwendung einer `DynaActionForm` müssten die Feld-Namen in der Struts-Konfiguration wiederholt werden. Schwerer wiegt jedoch, dass bei den Standard-Arten eine 1-zu-1-Beziehung zwischen den HTML-Widgets und den Feldern der Formulare bestehen. Das ist bei dieser Designalternative nicht der Fall. Deshalb gibt es eine neue Art von `ActionForm`, die alle Parameter von der Anfrage annimmt. Wird nach dem Befüllen die Validierung angestoßen, lädt die `ActionForm` die Konfiguration des Formulars und geht alle Formularfelder durch. Für jedes Formularfeld wird das passende `WebWidget` geholt, mit den Daten gefüllt und validiert. Wenn mindestens ein `WebWidget` eine invalide Eingabe enthält, schlägt die Validierung des `ActionForm`s fehl und die JSP, von der die Anfrage ausgelöst wurde, wird erneut aufgerufen (siehe Abschnitt 3.8). Verläuft die Validierung erfolgreich, kann die `Action` auf die Fachwerte zugreifen, die während der Validierung in der `ActionForm` erzeugt wurden.

4.2.1 Bewertung

Die `WebWidget`s ermöglichen die einfache Vor-Validierung auf dem Rechner des Benutzers. Der Entwickler muss hierzu nur Fachwert-Fabriken zur Verfügung stellen, die reguläre Ausdrücke für diese Validierung zurückgeben können. Die Nutzung dieses Features erfordert nur wenig Arbeit auf seiten des Entwicklers. Damit ist die Anforderung „Zweistufige Validierung ermöglichen“ erfüllt.

Die HTML-Repräsentation eines `WebWidget`s kann aus mehreren HTML-Widgets bestehen. Da `WebWidget`s bei einer Validierung die Inhalte aller zu ihnen gehörenden HTML-Widgets beachten, ist eine Beschränkung auf die Validierung einzelner HTML-Widgets nicht mehr gegeben. Damit ist die Anforderung „Trennung Feld und Anzeigeelement“ erfüllt.

Die `WebWidget`-Fabrik verwaltet die `WebWidget`s zentral und erfüllt somit Anforderung „Zentrale Widgetverwaltung“.

`WebWidget`s erzeugen Fachwerte. Die `Action`s verwenden die Fachwerte, die während der Validierung eines Anfrage-`ActionForm`s erzeugt werden. Damit wird die Anforderung ,,Fachwerte verwenden“ erfüllt.

Die Designalternative enthält eine TagLib, die es ermöglicht Platzhalter anstelle von Dateneingabe-Widgets zu verwenden. Diese Platzhalter werden während der Laufzeit durch die benötigten `WebWidget`s ersetzt. Damit erfüllt die Designalternative die Anforderungen ,,JSP-Taglib anbieten“ und ,,Platzhalter für Dateneingabe in HTML“.

Die Anforderung ,,Model-2-Architektur verwenden“ wird bereits von Struts erfüllt, da es ein Rahmenwerk für die Entwicklung von Anwendungen mit einer Model 2-Architektur ist. Die Designalternative ändert daran nichts. Die Erfüllung der Anforderung ,,Fachliche Dienstleister verwenden“ obliegt dem Entwickler einer Anwendung.

Obwohl die Designalternative alle Anforderungen erfüllen kann, gibt es einen uneleganten Aspekt in ihr, der auf das Design von Struts zurückzuführen ist. Wenn eine Anfrage eintrifft, wird zunächst eine `ActionForm` gefüllt. Erst wenn die `ActionForm` validiert wird, können die `WebWidget`s die Anfragedaten analysieren und die Eingaben validieren. Die `ActionForm` nimmt hier zwei Rollen ein. Zuerst ist sie ein generischer Container für die Anfragedaten. Nach der Validierung enthält sie die Fachwerte des Formulars. Besser wäre es, wenn die `WebWidget`s ihre Daten früher validieren könnten und nur die Fachwerte in die `ActionForm` gefüllt würden. Da Struts selbst das Konzept eines `WebWidget`s nicht kennt, ist dies jedoch nicht möglich und die Designalternative bleibt eine verwendbare Lösung der in dieser Arbeit behandelten Probleme mit einem konzeptionellen Schönheitsfehler.

4.3 Design auf Basis von wingS

Auch die wingS-Designalternative (zu wingS siehe Abschnitt 3.9; siehe Abbildung 4.2 für eine Skizze der Designalternative) enthält eine zentrale als Singleton realisierte Fabrik, die in diesem Fall für die Erfassung von Fachwerten entwickelte wingS-Widgets verwaltet.

Die Fachwert-Widgets sind Erweiterungen der normalen wingS-Widgets. Sie werden realisiert, indem Subklassen der wingS-Widgetklassen gebildet werden, die die zusätzliche Eigenschaft haben, Eingaben darauf prüfen zu können, ob sie Repräsentationen von Fachwerten sind. Kann eine Fachwert-Fabrik einen regulären Ausdruck für die Vor-Validierung auf dem Rechner des Benutzers anbieten, so kann ein Fachwert-Widget den benötigten Code für die Vor-Validierung generieren und über eine Operation der Oberklasse anmelden.

Diese Designalternative enthält im Gegensatz zu der auf Struts basierenden (siehe Abschnitt 4.2) keine zusätzliche Konfiguration. Die Anwendungslogik muss sicherstellen, dass die Widgets registriert werden, um sie später abholen zu können.

wingS-Anwendungen reagieren meistens nur auf `ActionEvent`s, denn das sind Ereignisse, die ausgelöst werden, wenn ein Benutzer auf einen Knopf oder einen Verweis in einer HTML-Seite klickt. Die anderen Ereignisse machen in wingS-Anwendung trotz ihrer Existenz keinen Sinn (siehe Abschnitt 3.9.1). Um eine Validierung der Fachwert-Widgets zu ermöglichen, steht eine ‚Validierer-Fassade‘ zur Verfügung, die für alle möglichen `ActionEvent`s in einer Anwendung registriert werden muss. Außerdem müssen der Validierer-Fassade alle Fachwert-Widgets bekannt gemacht werden, die verwendet werden. Wenn nun ein `ActionEvent` auftritt, ruft die Validierer-Fassade jedes Fachwert-Widget auf und fordert es auf, sich zu validieren. Die Fachwert-Widgets validieren darauf ihre Inhalte mit Hilfe ihrer Fachwert-Fabrik und ändern zum Beispiel ihre Schriftfarbe in rot, falls die Validierung fehlschlägt. Falls die Validierung aller Fachwert-Widgets erfolgreich verläuft, leitet die Validierer-Fassade das Ereignis an die Anwendungslogik weiter. Im Falle eines Misserfolgs, sind verschiedene Aktionen möglich. Die Validierer-Fassade kann das Ereignis einfach ignorieren und

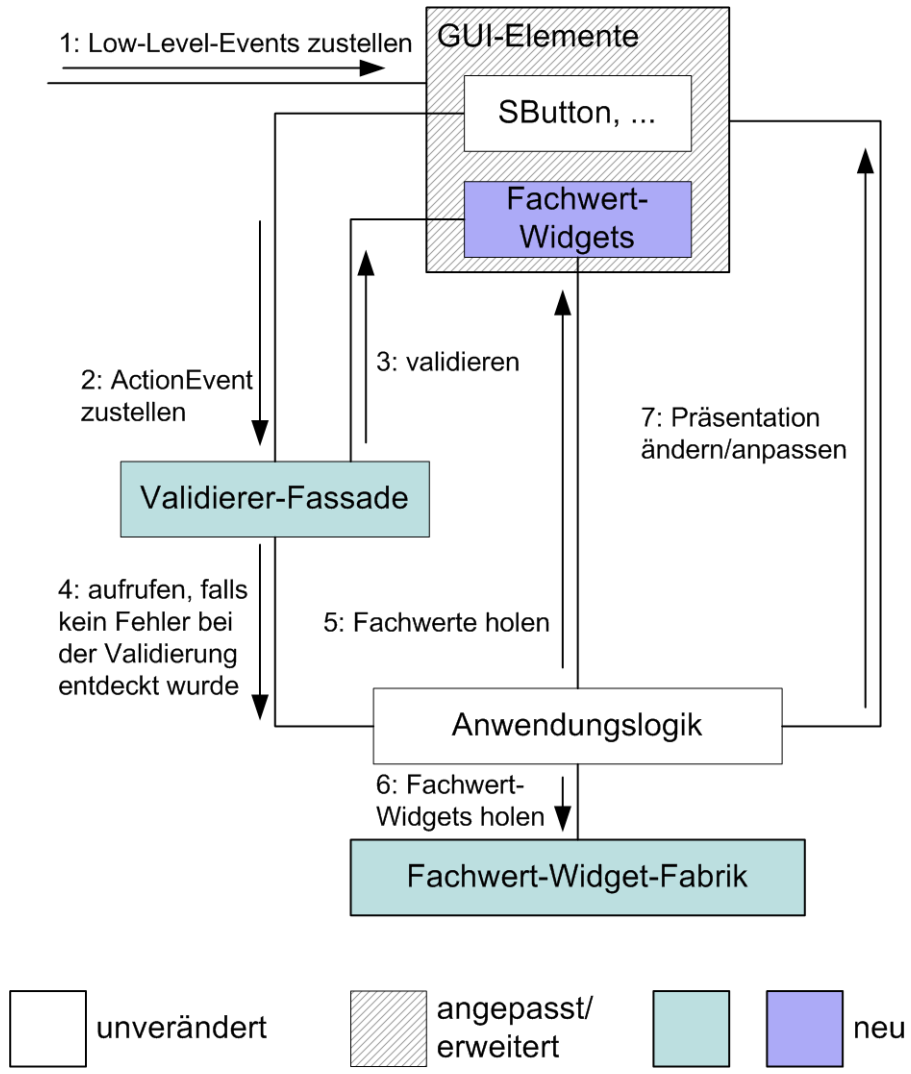

Abbildung 4.2 Designalternative auf Basis von wingS

nicht weiter verschicken, nach einer bestimmten Anzahl von Fehlversuchen ein neues Ereignis generieren und an die Anwendungslogik schicken, oder auch im Falle einer fehlgeschlagenen Validierung das Ereignis an die Anwendungslogik weiterleiten.

4.3.1 Bewertung

Die Fachwert-Widgets, die bei der wingS-Designalternative verwendet werden, haben die Möglichkeit JavaScript-Funktionen und -Aufrufe zu definieren, die für eine Vor-Validierung auf dem Rechner des Benutzers verwendet werden können. Somit erfüllt die Designalternative die Anforderung „Zweistufige Validierung ermöglichen“. Ein Problem ist jedoch, das Entwickler, die Anwendungen mit wingS entwickeln, dies tun, weil sie zu wenige Erfahrungen mit der Entwicklung von Web-Anwendungen haben. Meistens haben diese Entwickler nicht die Kenntnisse, die benötigt werden, um eine Vor-Validierung mit JavaScript zu programmieren. Solange nur die generischen Fachwert-Widgets verwendet werden und diese Entwickler keine eigenen, komplexeren Widgets

entwickeln müssen, ist das jedoch kein Problem. Ein anderes Problem ist, dass wingS einerseits die Präsentation von den Widgets durch das Konzept des austauschbaren Look And Feels trennt, aber andererseits die Spezifizierung von JavaScript-Funktionen an die Widgets bindet. Damit ist das Look And Feel nicht wirklich austauschbar, wenn eine Vor-Validierung mit JavaScript erfolgen soll. Die Präsentation, die generiert wird, muss HTML sein und der Entwickler, der die Vor-Validierung programmiert, muss wissen, wie die Präsentation des Widgets aussieht, um die Vor-Validierung richtig programmieren zu können.

Da wingS von sich aus wie Swing eine MVC-artige Architektur verwendet, und die Model 2-Architektur sich an MVC orientiert, ist die Anforderung ,,Model-2-Architektur verwenden“ erfüllt.

Die Anforderung ,,Trennung Feld und Anzeigeelement“ ist erfüllt, weil wingS-Widgets zu komplexeren Widgets kombiniert werden können.

Die Anforderung ,,JSP-Taglib anbieten“ ist für diese Designalternative irrelevant, da keine JSPs verwendet werden. Die Erfüllung der Anforderung ,,Fachliche Dienstleister verwenden“ obliegt dem Anwendungsentwickler. Anforderung 6 ist von wingS inspiriert und deshalb auch erfüllt.

Wird diese Designalternative verwendet, so liegt es nahe, dass Fachwerte verwendet werden, denn für Anwendungen, die keine Fachwerte verwenden, bietet sie keine Vorteile. Die Anforderung ,,Zentrale Widgetverwaltung“ ist durch die Fachwert-Widget-Fabrik erfüllt. Die Anforderung ,,Fachwerte verwenden“ ist dadurch erfüllt, dass es keinen Sinn macht, in der Anwendungslogik nicht auch Fachwerte zu verwenden, wenn das System um die Anwendungslogik herum Fachwerte verwendet.

Obwohl diese Designalternative alle Anforderungen erfüllen kann, ist sie weniger geeignet für die Entwicklung von Web-Anwendungen als die auf Struts basierende Designalternative. Das liegt hauptsächlich daran, dass Struts eine bessere Basis für Web-Anwendungen ist als wingS (Siehe hierzu auch die Bewertung von wingS in Abschnitt 3.9.1). Die Designalternative macht deutlich, dass wingS kein Swing für das Web ist, und dass Web-Anwendungen sich nicht auf die gleiche Art entwickeln lassen wie Desktop-Anwendungen. Ist sich ein Entwickler dieses Umstandes bewusst kann er mit wingS und der hier vorgestellten Erweiterung kleine Anwendungen schnell entwickeln. Dies ist jedoch nur für Anwendungen zu empfehlen, die absehbar nicht ausgebaut werden sollen.

4.4 Design auf Basis von JavaServer Faces

Die Designalternative, die auf JavaServer Faces basiert (Siehe Abschnitt 3.10) ist in Abbildung 4.3 skizziert. Die Designalternative erweitert JSF um Fachwert-Oberflächenkomponenten und eine passende zentral als Singleton realisierte Fabrik, an der Fachwert-Oberflächenkomponenten für Fachwerttypen registriert und abgeholt werden können.

Wie die Struts-Designalternative (siehe Abschnitt 4.2), hat diese Designalternative eine zusätzliche Konfiguration, in der die Oberflächenkomponenten den Fachwerttypen zugeordnet und die Formulare der Anwendung konfiguriert werden.

Die JSPs verwenden ein neues Tag, das wie in der Struts-Designalternative nur einen Namen als Attribut hat. Über den Namen und den Namen des es einschließenden `form`-Tags liest dieses Tag die Konfiguration des Formularfeldes, für das es steht, und holt von der Fachwert-Oberflächenkomponenten-Fabrik die passende Oberflächenkomponente, wenn die Anwendungslogik diese nicht bereits erzeugt hat. Diese Oberflächenkomponente generiert ihre HTML-Repräsentation, die dann durch das Tag in das Antwort-HTML eingefügt wird.

Die Fachwert-Oberflächenkomponenten validieren die Eingaben sobald sie mit den Anfrageparametern befüllt werden (Siehe hierzu Abbildung 3.14). Sie passen also in das JSF-System problemlos hinein. Dieser Zeitpunkt ist dafür vorgesehen, die Zeichenketten in andere Datentypen zu konvertieren. Tritt hierbei ein Fehler auf, wird die JSP, die die Anfrage ausgelöst hat, erneut angezeigt

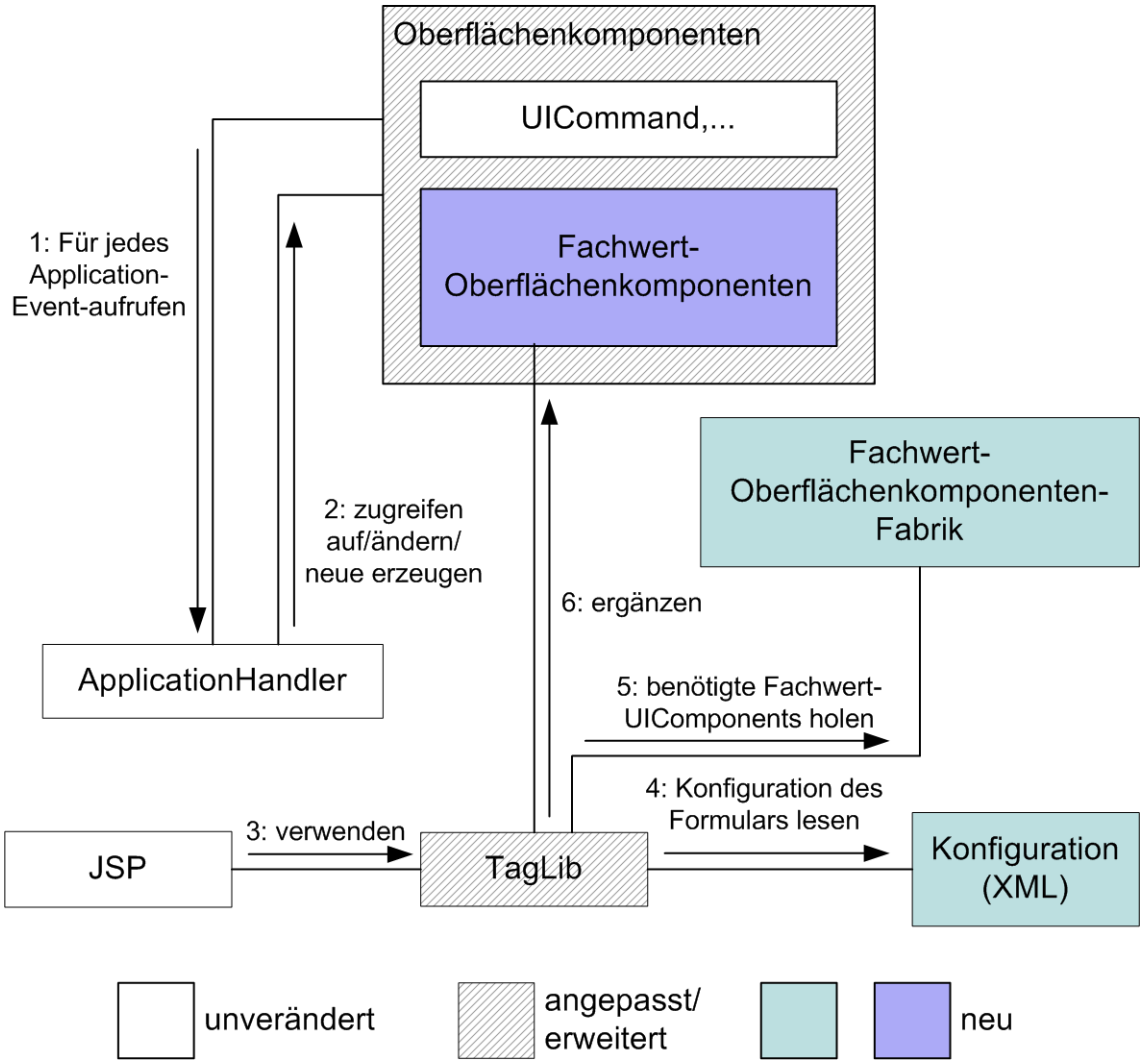

Abbildung 4.3 Designalternative auf Basis von JSF

und die Fachwert-Oberflächenkomponente macht an ihrer Präsentation deutlich, dass sie fehlerhaft ausgefüllt wurde.

Fachwert-Oberflächenkomponenten generieren den benötigten Code für die Vor-Validierung, wenn die Fachwert-Fabrik, die sie verwenden, einen regulären Ausdruck zurückgeben kann, der hierfür verwendet werden kann. Dieser Code wird von der Fachwert-Oberflächenkomponente in ihre HTML-Repräsentation eingefügt und auf dem Rechner des Benutzers ausgeführt.

Die Fachwert-Oberflächenkomponenten verwenden keine Renderer. Sie erzeugen ihre HTML-Repräsentation selbst und verarbeiten auch selbst die Anfragedaten. Dies scheint zunächst eine Einschränkung in der Flexibilität zu sein und zu einem gewissen Grade trifft diese Vermutung auch zu. So kann jetzt eine Anwendung, die Fachwert-Oberflächenkomponenten verwendet, die eine HTML-Präsentation erzeugen, nicht auf einem Mobiltelefon zum Laufen gebracht werden, das die Verwendung von WML erfordert. Mit den Standard-Oberflächenkomponenten ist das möglich, denn hier müssen nur die Renderer ausgetauscht werden.

Mehrere Gründe sprechen dafür, die Renderer nicht zu verwenden:

- **Einfache Konfigurierbarkeit:** Wenn Renderer verwendet würden, würde es nicht genügen nur die Zuordnung der Oberflächenkomponenten zu den Fachwerttypen zu konfigurieren. Es müsste auch konfiguriert werden, welche Renderer für welche Oberflächenkomponenten verwendet werden können. Das Tag, das in den JSPs verwendet wird, müsste außerdem eine Möglichkeit haben, zu ermitteln, welche Art von Renderer in einem konkreten Fall benötigt wird. Wenn es sich in einem HTML-Dokument befände, müsste es den HTML-Renderer verwenden. Wenn es sich in einem WML-Dokument befände, müsste es den WML-Renderer verwenden.

- **Einfache Oberflächenkomponenten:** Auch komplexe Fachwert-Oberflächenkomponenten sind meistens übersichtlich. Fachwert-Oberflächenkomponenten verarbeiten meistens keine Low-Level-Events und passen ihre HTML-Repräsentation nur an, um deutlich zu machen, dass sie fehlerhaft ausgefüllt wurden. Fachwert-Oberflächenkomponenten führen auch keine Konvertierungen der Anfragedaten in andere Datentypen durch. Dafür und für die Validierung der Daten verwenden sie Fachwert-Fabriken. Fachwert-Oberflächenkomponenten erzeugen meistens also nur ihre HTML-Repräsentation und Verarbeiten mit Hilfe der Fachwert-Fabriken die Anfrageparameter

- **Leichtere und schnellere Entwicklung neuer Fachwert-Oberflächenkomponenten :** Fachwert-Oberflächenkomponenten sollen bei Bedarf auch vom Entwickler einer Web-Anwendung entwickelt werden können. Wenn hierzu nur eine Klasse mit ein paar Operationen implementiert werden muss, finden sich Entwickler schneller zurecht und können die benötigten Fachwert-Oberflächenkomponenten schneller entwickeln.

- **Unterschiedliche Anforderungen unterschiedlicher Plattformen :** Es ist eine Illusion, dass nur die Präsentation einer Web-Anwendung ausgetauscht werden muss, um sie zum Beispiel auch auf einem Mobiltelefon verwenden zu können. Andere Plattformen haben andere Anforderungen, denen man auch in der Struktur und dem Benutzungsmodell der Anwendung begegnen muss. Es kommt also sehr selten vor, dass eine Anwendung gleichzeitig eine HTML-Präsentation und eine WML-Präsentation verwendet. Wenn jedoch für jede Plattform eine eigene Anwendung konstruiert wird, die aber immer noch auf gemeinsame fachliche Dienstleister zugreift, sind Renderer nicht mehr notwendig, denn jede Anwendung kann die benötigten Oberflächenkomponenten für sich konfigurieren.

4.4.1 Bewertung

Fachwert-Oberflächenkomponenten können HTML-Repräsentationen haben, die mehrere HTML-Widgets enthalten. Sie fügen Code für die Vor-Validierung in ihre HTML-Repräsentationen ein und validieren die Eingaben mit Fachwert-Fabriken. Da Fachwert-Oberflächenkomponenten keine 1-zu-1-Beziehung zu den sie vertretenden HTML-Widgets in der Präsentation haben, können sie gut für die Erfassung der Daten für je ein Formularfeld verwendet werden. Damit erfüllt die Designalternative die Anforderungen ,,Zweistufige Validierung ermöglichen“ und ,,**Trennung Feld und Anzeigeelement**“.

Obwohl es nicht das Ziel von JSF ist, eine Architektur für Web-Anwendungen vorzugeben, ist die Architektur einer JSF-Anwendung typischerweise eine Model 2-Architektur. Dabei hat JSF nicht das Problem der Feingranularität des Controller-Teils, das bei Struts besteht (siehe Abschnitt 3.7.1). Bei JSF gibt es einen `ApplicationHandler`, der die Anfragen in einer der Anwendung

entsprechenden Art verteilen kann. Die Designalternative erfüllt somit die Anforderung „Model-2-Architektur verwenden“ besser als es bei der Struts-Designalternative möglich ist.

Wie die Struts-Designalternative beinhaltet auch diese eine TagLib mit einem Tag, das als Platzhalter für Dateneingabewidgets fungiert. Anforderungen „JSP-Taglib anbieten“ und „Platzhalter für Dateneingabe in HTML“ sind hiermit erfüllt.

Die Erfüllung der Anforderung „Fachliche Dienstleister verwenden“ obliegt auch bei dieser Designalternative dem Entwickler der konkreten Anwendung. Es bietet sich an, die Anfragen vom `ApplicationHandler` auf fachlich abgegrenzte Handler zu verteilen, die fachliche Dienstleister zur Erfüllung ihrer Funktion verwenden. So könnte es zum Beispiel einen Handler für die Benutzerverwaltung geben, der einen oder mehrere fachliche Dienstleister verwendet, die für die Benutzerverwaltung zuständig sind.

Die Anforderung „Zentrale Widgetverwaltung“ wird durch die Fachwert-Oberflächenkomponenten-Fabrik erfüllt, die die Oberflächenkomponenten für Fachwerte zentral verwaltet.

Da das ganze System um die Anwendungslogik Fachwerte verwendet, wird auch die Anwendungslogik einer Anwendung, die diese Designalternative verwendet, Fachwerte verwenden und somit die Anforderung „**Fachwerte verwenden**“ erfüllen.

Die Designalternative, die auf JavaServer Faces basiert, ist die beste der vorgestellten Designalternativen, weil sie sich problemlos in das System integrieren lässt. Die auf Struts basierende Designalternative hat dabei Probleme, da sie etwas tut, was in Struts nicht vorgesehen ist. Bei der Struts-Designalternative wurde zwar versucht, diese so gut wie möglich zu Struts passend zu machen, aber das resultierte darin, dass die `ActionForm`s zu generischen Containern werden mussten, die zunächst alle Anfrageparameter aufnehmen, und erst später mit Hilfe der WebWidgets validiert werden. Bei der wingS-Designalternative wurde eine Validierer-Fassade eingeführt, die die `ActionEvent`s, die in einer Anwendung auftreten können, abfängt und als Reaktion auf sie eine Validierung der Fachwert-Widgets verursacht. Dies ist jedoch eine wenig befriedigende Umgehung der Unzulänglichkeiten von wingS. Wenn der Entwickler einer Anwendung vergisst, die Validierer-Fassade einzusetzen, findet keine Validierung statt obwohl Fachwert-Widgets verwendet werden. Die JSF-Designalternative passt sehr gut zu den JavaServer Faces. Sie setzt an Punkten des Systems an, die für die Aufgaben vorgesehen sind, die von der Designalternative erledigt werden. Die Designalternative befreit Anwendungen, die mit ihr entwickelt werden, auch von den zwei in Abschnitt 3.10.1 genannten Nachteilen. Es werden keine Tags mehr verwendet, die die konkreten Oberflächenkomponenten angeben und Validatoren-Tags werden auch nicht mehr benötigt, denn deren Aufgabe wird durch die Fachwert-Fabriken erledigt.

4.5 Zusammenfassung

In diesem Kapitel wurden eingangs alle Anforderungen an eine Lösung der in dieser Arbeit behandelten Probleme aufgelistet. Diese Anforderungen sind im Kapitel 3 bei der Untersuchung einiger Web-Technologien und deren Unterstützung für die Herstellung von Konsistenz und Validität in Web-Anwendungen aufgestellt worden.

Nach der Auflistung der Anforderungen wurden drei Designalternativen für eine Lösung vorgestellt.

Die erste der Designalternativen basiert auf Struts (zu Struts siehe Abschnitt 3.7). Sie führt das Konzept der WebWidgets ein, die eine HTML-Repräsentation von sich generieren können und Anfragedaten mittels Fachwert-Fabriken validieren und in Fachwerte konvertieren. Die WebWidgets werden von einem Tag verwendet, das in JSPs als Platzhalter für sie dient. Dieses Tag fügt an seiner Stelle in das Antwort-HTML die HTML-Repräsentation des jeweiligen Web-Widgets ein. WebWidgets werden auch in `ActionForm`s verwendet, wo sie zur Validierung der Anfrageparameter einer Anfrage benutzt werden.

Die Struts-Designalternative erfüllt alle Anforderungen. Kritikwürdig an dieser Designalternativ ist, dass sie zum Teil unelegant ist und sich nicht nahtlos in Struts integrieren lässt.

Die zweite Designalternative basiert auf wingS (zu wingS siehe Abschnitt 3.9). Bei dieser werden wingS-Widgets verwendet, die die zusätzlichen Fähigkeiten haben, Eingaben mit Fachwert-Fabriken zu validieren und in Fachwerte zu konvertieren. Um eine Validierung in Gang zu setzen, wird eine Validierer-Fassade verwendet, die auf alle `ActionEvent`s der Anwendung reagiert und als Reaktion die Validierung aller an ihr angemeldeten Widgets in Gang setzt. Schlägt die Validierung fehl, wird das Ereignis je nach Konfiguration entweder an die Anwendungslogik weitergeleitet oder ignoriert. Wenn das Ereignis nach einer fehlgeschlagenen Validierung ignoriert wird, wird die Präsentation der Anwendung nur aktualisiert, wobei die Widgets, die fehlerhaft befüllt wurden, dies in ihrer Präsentation deutlich machen.

Auch die zweite Designalternative kann alle Anforderungen erfüllen. Sie ist jedoch eine schlechtere Designalternative als die, die auf Struts basiert. Die Struts-Designalternative profitiert von der Tatsache, das Struts besser geeignet ist für die Entwicklung von Web-Anwendungen als wingS.

Die dritte Designalternative basiert auf der Technologie JavaServer Faces (zu JSF siehe Abschnitt 3.10).

Diese Designalternative setzt wie auch die beiden anderen eine zentrale Fabrik ein, die in diesem Fall Fachwert-Oberflächenkomponenten verwaltet. Fachwert-Oberflächenkomponenten ähneln den WebWidgets der Struts-Lösung. Sie können eine HTML-Repräsentation generieren und verwenden Fachwert-Fabriken, um Anfrageparameter zu validieren und in Fachwerte zu konvertieren.

Wie in der Struts-Designalternative, wird auch hier ein Tag eingesetzt, dass als Platzhalter für die Oberflächenkomponenten fungiert, deren HTML-Repräsentationen es in das Antwort-HTML an seiner Stelle einfügt. Auch hier wird eine zusätzliche Konfiguration verwendet, die die Oberflächenkomponenten zu Fachwerttypen zuordnet und die Formulare und deren Felder definiert.

Die JSF-Designalternative ist die beste, der hier vorgestellten, da sie sich nahtlos in die Umgebung der JavaServer Faces integrieren lässt. JSF vereinbart den objekt-orientierten Zugang am besten mit den Beschränkungen, denen Web-Anwendungen unterworfen sind. Es bietet die beste Basis für eine Erweiterung, die die Probleme löst, die das Thema dieser Arbeit sind, und für die Entwicklung komplexer Web-Anwendungen. Von diesen Vorteilen profitiert die Designalternative und ergänzt die Technologie an den richtigen Stellen.

5 Realisierung

> *Knowledge is just opinion that you trust enough to act upon.*
>
> — Orson Scott Card : *Children of the Mind*

Im Folgenden wird die Realisierung der auf **JavaServer Faces basierenden Designalternative** vorgestellt. Diese wurde im vorigen Kapitel als diejenige identifiziert, die am besten geeignet ist, die Pröbleme, deren Lösung die Motivation dieser Arbeit ist, zu lösen.

5.1 Konfiguration

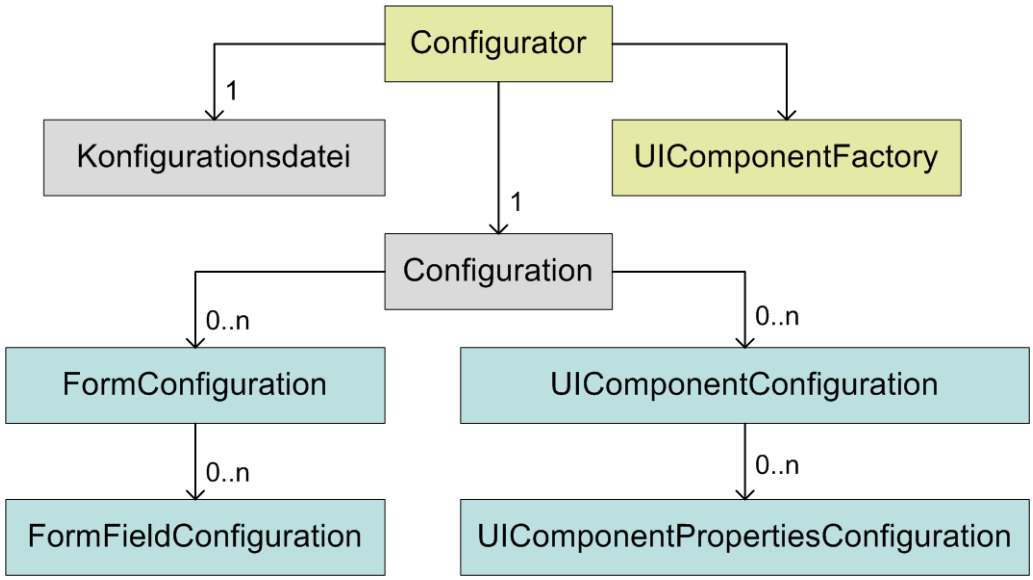

Abbildung 5.1 Die Klassen der Konfiguration

Die Konfiguration ist ein zentraler Teil der Lösung. Über sie werden alle wichtigen Aspekte gesteuert. Typischerweise wird die Konfiguration in einer XML-Datei spezifiziert, die von einem Singleton `Configurator` beim Starten des Systems eingelesen wird (siehe Abbildung 5.1). Dieses Singleton erzeugt aus den Daten, die es aus der XML-Datei lädt ein Objektgeflecht, dessen Wurzel ein Objekt vom Typ `Configuration` ist. Das Geflecht enthält Objekte für die Konfiguration von Formularen und Oberflächenkomponenten.

Die Konfiguration eines Formulars besteht aus dem Namen des Formulars und den Konfigurationen der Felder des Formulars. Eine Formularfeld-Konfiguration enthält den Namen des Feldes, den Fachwerttyp und die Angabe, ob das Feld verpflichtend ist.

Die Konfiguration einer Oberflächenkomponente besteht aus dem Klassennamen der Oberflächenkomponente, der Angabe des Fachwerttyps, einer Angabe der Fachwert-Fabrik, die verwendet werden soll, und aus der Angabe, ob eine Oberflächenkomponente für die ausschließliche Anzeige eines Fachwertes verwendet werden kann oder, ob ein Fachwert mit ihr auch eingegeben werden kann. Es können auch die Eigenschaften einer Oberflächenkomponente konfiguriert werden. Das ist wichtig, wenn zum Beispiel für die Erfassung eines Fachwertes ein Eingabefeld mit einer bestimmten Länge verwendet werden soll.

Nachdem die Konfiguration geladen wurde, in der noch einige andere Aspekte des Systems konfiguriert werden können als hier gezeigt (für eine genaue Untersuchung der Konfigurationsmöglichkeiten

siehe Abschnitt B.1), erzeugt der `Configurator` alle konfigurierten Oberflächenkomponenten und registriert sie an der `UIComponentFactory`, einem Singleton, dass die Oberflächenkomponenten und deren Zuordnung zu Fachwerttypen verwaltet.

Der `Configurator` ist außer für die Oberflächenkomponenten, die von der `UIComponentFactory` verwaltet werden, die Instanz, über die Zugriff auf die Konfiguration genommen werden kann. Er bietet Operationen an, die alle Zugriffe auf die Konfiguration vreinfachen. So muss nicht das Objektgeflecht durchsucht werden, um herausfinden zu können, welchen Fachwerttyp ein Formularfeld hat. Dafür und für andere Analysen auf der Konfigurationen gibt es entsprechende Operationen.

Am `Configurator` kann das das Geflecht der Konfigurations-Objekte gesetzt werden, was die Konfiguration auf andere Weisen als über eine Konfigurationsdatei ermöglicht. Zum Beispiel könnte eine Anwendung die Konfiguration in einer Datenbank sichern, über eine eigene Klasse laden und in ein passendes Geflecht von Konfigurations-Objekten konvertieren, das dann am `Configurator` gesetzt werden würde.

Somit ist die Konfiguration des Systems einerseits einfach über eine XML-Datei möglich und andererseits auch derart flexibel, das der Ort ihrer Persistenzhaltung einfach angepasst werden kann.

5.2 TagLib

Die TagLib der Lösung enthält zwei Tags, das `formfield`-Tag und das `uicomponent`-Tag.

Das `formfield`-Tag hat zwei Attribute:

- **name** Der Name des Formularfeldes, für das das Tag der Platzhalter ist.
- **edit** Dieses Attribut gibt an, ab das Formularfeld nur angezeigt werden soll oder ob es auch mit einem anderen Fachwert befüllt werden können soll.

Mit seinem Namen und dem Namen des umschließenden `form`-Tags ermittelt das Tag durch die Konfiguration, welchen Fachwerttyp das Feld hat und ob es verpflichtend ist. Ist der Fachwerttyp ermittelt, wendet sich das Tag an die `UIComponentFactory` und lässt sich die passende Oberflächenkomponente herausgeben. Um eine passende Oberflächenkomponente zu erhalten, muss das Tag der `UIComponentFactory` mitteilen, ob die Oberflächenkomponente nur zum Anzeigen oder auch zum Ändern des Wertes verwendet werden soll.

Die Angabe, ob ein Feld editiert werden soll oder nicht, ist hierbei ein problematischer Punkt. Diese Angabe ist eine fachlich begründete Angabe und gehört somit nicht in die Präsentation. Wollte man dieser Forderung genügen, müssten Formulare mehrfach definiert werden. Es müsste für jede Art, in der ein Formular verwendet wird, eine neue Konfiguration erstellt werden, in der angegeben ist, welche Felder editierbar sind und welche nicht. Das ist aber auch schlecht, weil es sich fachlich gesehen in allen Fällen um das gleiche Formular handelt. Ein weiteres Problem bei dieser Vorgehensweise ist, dass das Formular bei einer Veränderung an mehreren Stellen gepflegt werden muss. Der hier gewählte Weg ist ein Kompromiss. Er verlagert einen Teil des Anwendungswissens in die Präsentation, der sich vorraussichtlich nicht häufig ändert. Seiten, die Informationen anzeigen und Seiten, auf denen Informationen bearbeitet und erfasst werden, ändern ihre Bestimmungen nicht während sich die Informationsarten, die angezeigt oder bearbeitet werden, in der Entwicklung einer Anwendung häufig ändern können.

Das `uicomponent`-Tag ist für die Situationen gedacht, in denen der vorgestellte Umgang mit Formularen nicht genügt. Es gibt Situationen, in denen Formulare anders angezeigt werden müssen als gewöhnlich. So kann es sein, dass in einem Formular ein bestimmtes Feld nur dann sichtbar sein soll, wenn eine bestimmte Bedingung erfüllt ist. Dies lässt sich mit der Konfiguration

und dem `formfied`-Tag nicht bewerkstelligen. In diesem Fall wird die Anwendung alle Oberflächenkomponenten selbst erzeugen, die für die Seite, die als nächste angezeigt werden soll, verwendet werden. Die Anwendung hat hierbei die Möglichkeit, auf die `UIComponentFactory` zuzugreifen oder selbst die Oberflächenkomponenten zu erzeugen. Das `uicomponent`-Tag prüft, ob eine Oberflächenkomponente mit dem im `name`-Attribut spezifizierten Namen existiert und zeigt gegebenenfalls die HTML-Repräsentation der Oberflächenkomponente an. Ist keine entsprechende Oberflächenkomponente vorhanden, bleibt der Platz, der von dem Tag eingenommen wird, im generierten HTML unbesetzt. Das `uicomponent`-Tag ist ein mächtiges Werkzeug zur Erstellung dynamischer Seiten.

5.3 Die Oberflächenkomponenten für Fachwerte

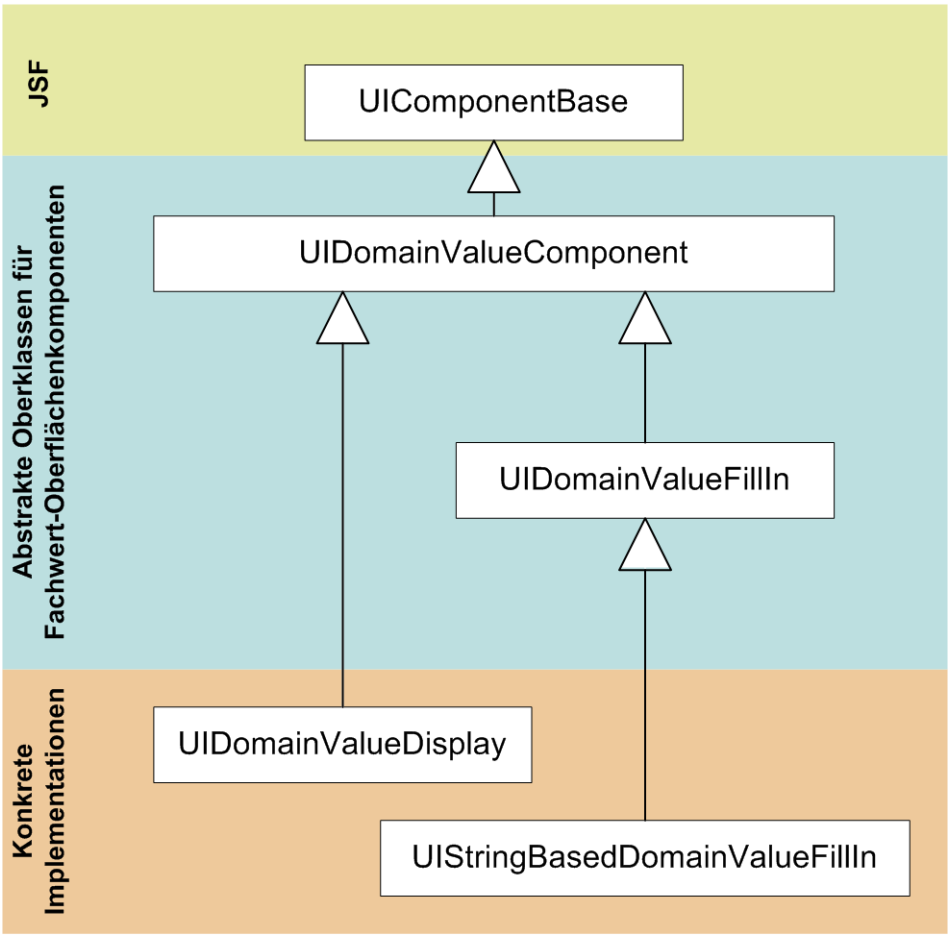

Abbildung 5.2 Die Klassen der Fachwert-Oberflächenkomponenten

Die Oberklasse für alle Oberflächenkomponenten, die mit Fachwerten umgehen können, ist `UIDomainValueComponent` (siehe Abbildung 5.2). Diese ist eine abstrakte Klasse, die von `UIComponentBase`, einer Basisklasse für Oberflächenkomponenten bei JSF, erbt und das Verhalten einer solchen Oberflächenkomponente definiert.

```
public abstract class UIDomainValueComponent extends UIComponentBase
{
  public Factory getDomainValueFactory();
  public void setDomainValueFactory(Factory factory);
  public boolean hasDomainValueFactory();
  public abstract boolean canHandleDomainValueFactory(Factory factory);

  public DomainValue getDomainValue();
  public void setDomainValue(DomainValue domainValue);
  public boolean hasDomainValue()
  public boolean canHandleDomainValue(DomainValue domainValue);
  public void removeDomainValue();

  public String[] getSupportedProperties();
  public boolean isPropertySupported(String propertyName);
  public Class getPropertyType(String propertyName);
  public final void setProperty(String name, Object value);
  public final boolean hasProperty(String name);
  public final Object getProperty(String name);
}
```

Das vorhergehende Listing zeigt nur die Schnittstelle der Klasse. Die Klasse selbst implementiert fast alle Operationen. Einige dieser Operationen können in Subklassen reimplementiert werden, falls dies notwendig wird. Andere Operationen sind nicht reimplementierbar, da der durch sie realisierte Algorithmus beschädigt werden könnte, wenn eine der Operationen in einer Subklasse eine neue Implementierung erhalten würde.

Die Operationen dieser Klasse teilen sich in drei Gruppen auf.

Die erste Gruppe der Operationen beschäftigt sich mit der **Fachwertfabrik**, die von der Oberflächenkomponente verwendet wird. Hier kann eine solche Fabrik gesetzt und auch wieder geholt werden. Die beiden anderen Operationen dieser Gruppe werden verwendet, um zwei Vorbedingungen der beiden ersten Operationen zu prüfen. Die Fabrik kann nur dann geholt werden, wenn sie vorhanden ist, und gesetzt werden kann nur eine Fabrik, mit der die Oberflächenkomponente umgehen kann.

Eine bestimmte Oberflächenkomponente kann zum Beispiel nur mit Fachwertfabriken umgehen, die ihre Fachwerte aus einer einfachen Zeichenkette erzeugen können. Andere Oberflächenkomponenten können zum Beispiel ganz speziell auf einen Fachwerttyp und damit auch eine ganz spezielle Fachwertfabrik zugeschnitten sein. `canHandleDomainValueFactory()` sogt dafür, dass eine Oberflächenkomponente nur mit Fachwert-Fabriken verwendet wird, mit denen sie umgehen kann.

Die zweite Gruppe von Operationen beschäftigt sich mit dem **Setzen und Abfragen des Fachwertes**, der in einer Oberflächenkomponente angezeigt wird oder vom Benutzer eingegeben wurde. `canHandleDomainValue()` stellt sicher, dass nur solche Fachwerte gesetzt werden, die von der verwendeten Fachwert-Fabrik erzeugt werden könnten.

Die Wichtigkeit dieser Einschränkung lässt sich an einem kleinen Beispiel zeigen. Angenommen es gibt ein Oberflächenkomponente, die generell mit Fachwertfabriken umgehen kann, die ihre Fachwerte aus einfachen Zeichenketten erzeugen können. In diesem Fall sei die Fabrik eine, die Datumsfachwerte erzeugen kann – also „10.01.2003“ und so weiter. Wenn nun anwendungsseitig an diese Oberflächenkomponente ein Zeit-Fachwert – also etwa „10:42“ – übergeben würde, dann würde dieser Fachwert zunächst angezeigt werden. Der Benutzer könnte diese Zeitangabe nun aber

nicht verändern ohne darüber benachrichtigt zu werden, dass seine Eingabe kein gültiger Datum-Fachwert ist. Die Fachwerte also, die programmseitig gesetzt werden können, müssen vom gleichen Typ sein wie die Fachwerte, die der Anwender eingeben kann.

Die letzte Operation in dieser Gruppe – `removeDomainValue()` – wird benötigt, um einen einmal gesetzten Fachwert auch wieder entfernen zu können. Die Oberflächenkomponente kehrt nach einem Aufruf der Operation in ihren Anfangszustand zurück, in dem sie keinen Fachwert hat. Dadurch kann eine Oberflächenkomponente mehrmals hintereinander verwendet werden.

Durch die letze Gruppe von Operationen wird das Konzept von **Eigenschaften** für Oberflächenkomponenten realisiert. Dies sind die Eigenschaften, die in der Konfiguration angegeben werden können. Der Umgang mit diesen Eigenschaften ist in dieser Lösung zu einem gewissen Maße generisch. Ein Vorteil dieser Variante ist, dass sich das Setzen dieser Eigenschaften beim Einlesen der Konfigurationsdatei besonders einfach bewerkstelligen lässt. Ein anderer Vorteil ist, dass das in der Konfigurationsdatei sichtbare Konzept der Eigenschaften auch in den konkreten Oberflächenkomponenten auftritt. Gäbe es diese generische Behandlung nicht, müsste es für jede Eigenschaft jeweils eine eigene Operation zum Setzen und eine zum Lesen der Eigenschaft geben.

Um die Generezität nicht ausufern zu lassen und eine gewisse Art von Typsicherheit zu erreichen, muss jede Oberflächenkomponente, an der Eigenschaften gesetzt werden können sollen, zwei Operationen reimplementieren. Die erste – `getSupportedProperties()` – gibt die Namen der für die Oberflächenkomponente unterstützten Eigenschaften an. Damit wird verhindert, dass Eigenschaften gesetzt werden, die von der Oberflächenkomponente nicht unterstützt werden. Wenn eine Web-Anwendung in eine Servlet-Engine geladen wird und der Ladevorgang erfolgreich abgeschlossen wird, ist sichergestellt, dass die Eigenschaften, die in der Konfiguration angegeben wurden, von der Oberflächenkomponente unterstützt werden und die Konfigurationsdatei keine Tippfehler enthält. Die zweite Operation – `getPropertyType()` – stellt die Typsicherheit her.

Wird beim Einlesen der Konfiguration eine Eigenschaft gesetzt, dann wird über die erwähnte Operation der Typ ermittelt, von dem die Eigenschaft ist, und es wird versucht die Zeichenkette, die für die Eigenschaft in der Konfigurationsdatei eingetragen wurde, in ein Objekt des entsprechenden Typs zu konvertieren. Geht die Erzeugung des Objektes mit der Operation oder dem Konstruktor problemlos vonstatten, dann wird die Eigenschaft gesetzt. Anderenfalls, wird der Ladevorgang abgebrochen, und der Entwickler der Web-Anwendung hat die Gelegenheit, die Konfigurationsdatei zu korrigieren.

`UIDomainValueComponent` eignet sich gut als direkte Oberklasse für Fachwert-Oberflächenkomponenten, die Fachwerte nur anzeigen können. In Abbildung 5.2 ist `UIDomainValueDisplay` als ein Beispiel für diese Art von Oberflächenkomponenten zu sehen.

Für Oberflächenkomponenten, mit denen Fachwerte erfasst werden können, gibt es die Klasse `UIDomainValueFillIn`. Diese Klasse enthält zusätzliche Operationen, die verwendet werden, um anzuzeigen, dass die Angabe eines Wertes verpflichtend ist oder dass die Oberflächenkomponente einen invaliden Wert enthält. Damit wird für Konsistenz in der Präsentation gesorgt, denn gleiche Umstände werden auf die gleiche Art angezeigt. Ein Beispiel für diese Art von Oberflächenkomponenten ist `UIDomainValueStringBasedFillIn`, das ein HTML-Eingabefeld für die Erfassung von Fachwerten ist, die aus einer Zeichenkette erzeugt werden können. In Anhang B.3 werden einige Implementationsdetails dieser Oberflächenkomponente vorgestellt.

Die hier vorgestellte Implementierung der Fachwert-Oberflächenkomponenten erlaubt einerseits die Entwicklung einfacher Oberflächenkomponenten für Fachwerte, indem vieles bereits in den Oberklassen vorimplementiert ist. In einer konkreten Oberflächenkomponente muss nur das implementiert werden, was für diese Oberflächenkomponente spezifisch ist. Andererseits ermöglicht die Implementierung die Entwicklung eigener komplexer Oberflächenkomponenten, die zum Beispiel das vorimplementierte Generieren des Vor- und Nachspann ignorieren können.

Die Verwendung der Eigenschaften der Fachwert-Oberflächenkomponenten ermöglicht die einfache Konfiguration der Oberflächekomponenten. Obwohl die Verwaltung generisch erfolgt, sorgen zwei Mechanismen dafür, dass nur definierte Eigenschaften gesetzt werden können und dass diese Eigenschaften bestimmte Typen haben.

5.4 Vorprüfung mit JavaScript

Die Konfiguration enthält nur die Angaben darüber, ob eine Vorprüfung durchgeführt werden soll, und welche JavaScript-Funktion dazu verwendet werden soll. Damit gibt es keine Einschränkungen darin, wie diese Funktionalität implementiert wird und wo.

Werden die bereits vorhandenen Oberflächenkomponenten für Fachwerte verwendet, dann kommt eine Einschränkung hinzu, die die Signatur der Funktion betrifft.

Die Parameter, die eine solche Funktion verarbeiten muss, sind die folgenden:

- der reguläre Ausdruck, gegen den eine Eingabe geprüft werden soll
- der Inhalt der Oberflächenkomponente in einer Zeichenkette
- ein Array von HTML-Elementen, die diesen Inhalt zusammen beinhalten, oder falls es nur ein einziges solches HTML-Element gibt, dieses Element statt des Arrays

Es fällt auf, das zum einen die HTML-Elemente übergeben werden und zum anderen der Wert, der in sie eingegeben wurde. Auf den ersten Blick scheint dies redundant zu sein. Diese Redundanz besteht aber nur, wenn es nur Oberflächenkomponenten gibt, die jeweils ein einziges HTML-Element verwenden.

Bei komplexen Oberflächenkomponenten kommt man ohne diese Parameter nicht aus. Als ein Beispiel soll hier eine Oberflächenkomponente dienen, in die ein Autokennzeichen eingegeben werden kann. Autokennzeichen bestehen in Deutschland aus drei Teilen, die von Bindestrichen getrennt werden. Diese sind das Städtekennzeichen, zwei beliebige Buchstaben und eine Nummer. Würde man der Funktion nun nur den regulären Ausdruck und die HTML-Elemente übergeben, müsste diese aus den Inhalten der HTML-Elemente die Zeichenkette zusammensetzen, die dann gegen den regulären Ausdruck getestet wird. Dann müsste es aber für jede spezielle Oberflächenkomponente eine spezielle JavaScript-Funktion geben, die die Validierung übernimmt. Das würde bei großen Formularen unter Umständen zu einer großen Menge an JavaScript-Code führen, der auf den Rechner des Anwenders übertragen werden müsste, was sich verlangsamend auf die Interaktion mit der Anwendung auswirken würde.

Die Standardimplementierung der JavaScript-Funktion für die Vor-Prüfung wird in Anhang B.4 vorgestellt.

Für die Vor-Validierung ist nur die Signatur der JavaScript-Funktion vorgegeben, die verwendet werden soll. Werden nur eigene Fachwert-Oberflächenkomponenten verwendet, so entfällt auch diese Einschränkung. Diese Funktionalität ist also sehr flexibel und lässt sich problemlos an die Erfordernisse der jeweiligen Anwendung anpassen.

5.5 Zusammenfassung

In diesem Kapitel wurde eine Lösung der in dieser Arbeit behandelten Probleme vorgestellt (siehe Abschnitt 1.1).

Dazu wurde zuerst die Konfiguration der Lösung vorgestellt. Die Konfiguration ist die Schaltzentrale. Hier können die Fachwerttypen und ihre zugehörigen Oberflächenkomponenten konfiguriert werden wie auch die Formulare mit ihren Feldern und Fachwerttypen.

Danach wurde die TagLib der Lösung vorgestellt. Diese besteht aus zwei Tags. Das erste Tag ist das `formfield`-Tag. Über dieses Tag wird automatisch zur Laufzeit die richtige Oberflächenkomponente ausgewählt und angezeigt. Der Entwickler der JSPs muss nichts über diese Oberflächenkomponenten wissen. Er muss sie nur anordnen. Damit wird eine klare Trennung zwischen Funktion und Präsentation herbeigeführt. Da für einen Fachwerttyp immer dieselbe Oberflächenkomponente ausgewählt wird, ist die Oberfläche der Anwendung konsistent. Für Anwendungen, die ein dynamischeres Verhalten erfordern, als mit der Konfiguration von Formularen und dem `formfield`-Tag möglich ist, ist das `uicomponent`-Tag bestimmt. Dieses Tag prüft, ob es eine Oberflächenkomponente mit dem im `name`-Attribut definierten Namen gibt und zeigt sie gegebenenfalls an. Auch bei der Verwendung dieses Tags ist bei der Entwicklung der JSPs kein Wissen über die letztendlich verwendeten Oberflächenkomponenten notwendig. Sie müssen jedoch in der Anwendungslogik erzeugt werden. Dies führt zu einer größeren Flexibilität, denn es können in verschiedenen Situationen verschiedene Oberflächenkomponenten an derselben Stelle in einer JSP gesetzt werden oder in anderen Situationen einige Stellen in der JSP freibleiben.

Schließlich wurden die Oberflächenkomponenten für Fachwerte vorgestellt. Diese konvertieren Eingaben in die passenden Fachwerte und sorgen für die richtige Anzeige der Fachwerte. Die Lösung enthält Oberklassen für Fachwert-Oberflächenkompnenten, mit denen sich leicht spezielle Oberflächenkomponenten entwickeln und einsetzen lassen. Die Verwendung dieser Oberflächenkomponenten sorgt für Validität in der Web-Anwendung, da diese nur valide Daten präsentiert bekommt.

Darauf wurde die Vorprüfung mit JavaScript vorgestellt. Der Benutzer bekommt hiermit eine schnellere Rückkopplung über die Validität der Eingaben. Tippfehler können also oft frühzeitig erkannt und korrigiert werden, ohne dass eine Kommunikation mit dem Server notwendig wird.

6 Zusammenfassung und Ausblick

> *Que sera, sera*
> *Whatever will be, will be*
> *The future's not ours to see*
> *Que sera, sera*
> *What will be, will be*
>
> — Doris Day : *Whatever Will Be, Will Be*

Die Motivation für diese Arbeit ist die Erkenntnis gewesen, dass die Verwendung von HTML-Formularen bei der Entwicklung von Web-Anwendungen viele Nachteile mit sich bringt. Es gibt nur unzureichende Möglichkeiten, den Typ eines Formularfeldes zu definieren. HTML-Formulare können also nicht die Validität der eingegebenen Daten sicherstellen. In HTML-Formularen wird gleichzeitig definiert, welche Felder das Formular hat und wie und mit welchen Widgets es angezeigt werden soll. In HTML-Formularen findet also eine Vermischung von Präsentation und Funktion statt. HTML-Formulare können auch nicht sicherstellen, dass für einen bestimmten Typ von Daten immer dieselbe Art von Widgets verwendet wird. Die Herstellung der Konsistenz der Präsentation wird von den HTML-Formularen nicht unterstützt. Trotz dieser Nachteile kann bei der Entwicklung von Web-Anwendungen nicht auf HTML-Formulare verzichtet werden, denn HTML-Formulare sind das mächtigste Mittel zur Konstruktion der Präsentation einer Web-Anwendung. Konzepte zu erarbeiten, wie Validität und Konsistenz in Web-Anwendungen erreicht werden können, ist das Ziel der Arbeit gewesen.

Ich habe verschiedene Web-Technologien und deren Unterstützung für die Herstellung von Validität und Konsistenz in Web-Anwendungen untersucht und vorgestellt. Die Untersuchungen fingen mit den Basis-Technologien für Web-Anwendungen – HTML, HTML-Formulare – an. Danach habe ich die Basistechnologien für Web-Anwendungen, die in der Programmiersprache Java entwickelt werden, – Servlet-API, JavaServer Pages – untersucht. Ich habe diese Untersuchungen mit der Betrachtung höherer Technologien – Struts, wingS und JavaServer Faces – abgeschlossen.

Die Untersuchung der Web-Technologien hat zu zwei Ergebnissen geführt. Das erste dieser Ergebnisse ist die Erkenntnis, dass keine der untersuchten Technologien das Ziel dieser Arbeit erreichen kann. Es gibt einige Ansätze, die jedoch nicht weit genug gehen und zum Teil auch konzeptionelle Schwächen aufweisen. Das zweite Ergebnis ist eine Sammlung von Anforderungen, die eine Lösung, die das Ziel erreicht, erfüllen muss.

Die Anforderungen sind

- Web-Anwendungen sollen bei der Validierung von Daten ein zweistufiges Verfahren einsetzen, das sich aus einer Vor-Validierung auf dem Rechner des Benutzers und einer Haupt-Validierung auf dem Server zusammensetzt. Die Vor-Validierung soll einfach und generisch sein. Die Hauptvalidierung soll so akkurat wie möglich sein. Es kann vorkommen, dass die Haupt-Validierung Fehler entdeckt, die die Vor-Validierung nicht entdeckt hat.

- Web-Anwendungen sollen eine Model 2-Architektur verwenden, um eine Trennung zwischen Präsentation und Anwendungslogik zu erreichen.

- Ein Formular besteht aus Feldern, die Daten aufnehmen können. Wird ein Formular angezeigt, so kann es sein, dass mehrere Widgets für ein Feld verwendet werden. Die Validierung eines Formulars muss die Felder des Formulars validieren und nicht die einzelnen Widgets.

- Eine Lösung soll eine JSP-TagLib anbieten, um zu verhindern, dass Java-Code für die Verwendung der Lösung in JSPs eingestreut werden muss.

- Web-Anwendungen sollen fachliche Dienstleister verwenden und somit eine Trennung zwischen der fachlichen Logik und der Logik erreichen, die sich ausschließlich mit den speziellen Anforderungen der Web-Anwendung befasst.
- Web-Anwendungen sollen in den statischen HTML-Seiten, die ihre Präsentation bilden, Platzhalter für Dateneingabewidgets verwenden, denn das Wissen darüber, welche Widgets für das Eingeben bestimmter Daten benötigt werden, gehört zur Anwendungslogik. Die Platzhalter sollen zur Laufzeit durch die benötigten Widgets ersetzt werden.
- Web-Anwendungen sollen Fachwerte verwenden. Die Verwendung von Fachwerten ist eine Maßnahme in die Richtung der Herstellung und Sicherstellung von Validität.
- Die Oberflächenkomponenten (oder allgemeiner Widgets), die für das Erfassen von Fachwerten verwendet werden, sollen an einer zentralen Stelle verwaltet werden. Da dadurch für das Eingeben von Fachwerten eines bestimmten Typs immer die gleiche Art von Oberflächenkomponente verwendet wird, wird hiermit für Konsistenz in der Anwendung gesorgt.

Ich habe drei Designalternativen vorgestellt, von denen jede jeweils auf einer der höheren Technologien basiert, die ich bei der Untersuchung der Web-Technologien vorgestellt habe. Bei der Modellierung jeder Designalternative habe ich versucht, sie so nahtlos wie möglich in die jeweilige Technologie zu integrieren. Es hat sich herausgestellt, dass diese nahtlose Integration am besten bei der Designalternative funktioniert, die auf JavaServer Faces basiert. Das liegt daran, dass diese Technologie selbst schon konzeptionell in die richtige Richtung geht. Mit der Designalternative, die auch einige Nachteile der JavaServer Faces nivelliert, kann das Ziel also am besten erreicht werden.

Zum Abschluss der Arbeit habe ich die Realisierung der JSF-Lösung vorgestellt und damit belegt, dass die Umsetzung der Designalternative machbar ist. Die Realisierung teilt sich in vier Bereiche auf

- **Die Konfiguration** ist die Schaltzentrale des Systems. Hier werden Formulare mit ihren Formularfeldern definiert und Oberflächenkomponenten Fachwerttypen zugeordnet.
- **Die TagLib** besteht aus zwei Tags. Eines dieser Tags greift auf die Formulardefinition und die Zuordnung der Oberflächenkomponenten zu den Fachwerttypen zu. Es wird als Platzhalter für die Oberflächenkomponenten verwendet, durch die es sich während der Laufzeit ersetzt. Das andere Tag dient auch als Platzhalter, aber es greift nicht auf die Konfiguration zu. Ein Exemplar dieses Tags ersetzt sich durch eine Oberflächenkomponente mit dem gleichen Namens, wenn eine solche existiert oder lässt anderenfalls den Platz frei. Dieses Tag wird zur Dynamisierung der Seiten verwendet, wenn die Verwendung der Formulardefinitionen nicht genügt.
- **Fachwert-Oberflächenkomponenten** sind Oberflächenkomponenten, die mit Fachwerten umgehen können. Das bedeutet, dass sie Eingaben darauf validieren können, ob sie valide Repräsentationen von Fachwerten sind, und Eingaben in Fachwerte konvertieren können. Die Realisierung enthält einige Klassen, die den generischen Teil einer Fachwert-Oberflächenkomponente vorimplementieren und so die Entwicklung spezieller Fachwert-Oberflächenkomponenten erleichtern. Ebenso sind generische Oberflächenkomponenten vorhanden, die ohne weiteren Programmieraufwand eingesetzt werden können.
- **Die Vor-Validierung mit JavaScript** wird ebenfalls von der Realisierung unterstützt. Eine Beispielimplementierung der JavaScript-Funktion, die in den meisten Fällen ausreicht, ist vorhanden und kann leicht verwendet werden. Eigene Implementierungen sind jedoch auch integrierbar. Damit die Vor-Validierung mit JavaScript funktionieren kann, ist lediglich eine

erweiterte Fachwert-Fabrik für jeden verwendeten Fachwerttyp notwendig, die einen regulären Ausdruck für die Validierung herausgeben kann.

Beim derzeitigen Stand der Technik ist JavaServer Faces die beste Technologie, mit der Web-Anwendungen entwickelt werden können. Die vorgestellte Lösung erweitert diese Technologie in entscheidenden Punkten und erreicht das Ziel, Validität und Konsistenz in Web-Anwendungen herzustellen.

Eine neuer Standard, der gerade in der Entwicklung ist, heißt XForms (siehe [w3c-website]). Dieser Standard beschäftigt sich mit der Trennung des Inhaltes von Formularen und der Darstellung von Formularen. Es ist zu erwarten, dass sich XForms auf das Thema ,,Herstellung und Sicherstellung von Validität und Konsistenz in Web-Anwendungen“ auswirken wird. Wie weit diese Auswirkungen gehen werden, wird zur gegebenen Zeit erkundet werden müssen.

Andere Entwicklungen in der Java-Welt gehen weg von der Verwendung von JSPs zu einer vollkommenen Trennung des statischen und dynamischen Teils einer HTML-Seite. Der dynamische Teil enthält in diesen neuen Entwicklungen die Widgets und die Aktionen, die mit ihnen ausgeführt werden. Eine dieser Aktionen ist auch die Validierung des Inhaltes der Widgets. Eines der Produkte aus diesem Bereich ist ,,JetBrains Fabrique“, das zurzeit jedoch in einem sehr frühen Entwicklungsstadium ist. Sobald es jedoch eine gewisse Reife erreicht hat, kann es interessant sein, zu untersuchen, wie es sich auf die Entwicklung von Web-Anwendungen auswirkt.

In dieser Arbeit ist nur die Validierung auf Feld-Ebene betrachtet worden. Eine Untersuchung, ob sich die Validierung auf der Material-Ebene und der Material-Kontext-Evene durch ein Rahmenwerk oder auf andere Weise unterstützen lässt könnte jedoch auch interessant sein, da immer komplexere Web-Anwendungen konstruiert werden, bei denen diese beiden Validierungs-Ebenen eine immer größere Rolle spielen.

A Beispiele und vertiefende Betrachtungen einzelner Technologien

Dieses Kapitel enthält einige Beispiele zu Themen, die im Haupttext der Arbeit angesprochen werden und gibt auch einen tieferen Einblick in die in diesen Beispielen verwendeten Technologien.

A.1 Beispiel zur sofortigen Validierung mit JavaSript

In diesem Abschnitt wird ein Beispiel für die in Abschnitt 3.3.2 angesprochene sofortige Validierung von HTML-Formularen gegeben.

```
<html>
  <head>
    <script language="JavaScript" type="text/javascript">

      function isValid()
      {
        var datum = document.vform.datum.value;
        var match = datum.match(/\d{1,2}\.\d{1,2}\.\d{4}/);
        return (datum.length == 0 || (match && match.length == 1
                                      && match[0].length == datum.length));
      }

      function validate()
      {
        var validityMarker = document.getElementById("validitiyMarker");
        if(isValid())
        {
          validityMarker.src="valid.gif";
          validityMarker.title="";
        }
        else
        {
          validityMarker.src="invalid.gif";
          validityMarker.title="\"" + datum + "\" ist kein Datum";
        }
      }

      function checkAndSubmit()
      {
        if(isValid())
        {
          document.vform.submit();
        }
      }

    </script>
  </head>
  <body>
    <form name="vform" action="servlet/MyServlet" method="get">
      Datum : <input name="datum" onblur="javascript:validate()" type="text"/>
      <img title="" src="valid.gif" id="validitiyMarker" border="0"/>
      <br/><br/>
```

```
      <a href="javascript:checkAndSubmit()">
        <img src="abschicken.gif" border="0"/>
      </a>
    </form>
  </body>
</html>
```

Das Formular ist sehr einfach gehalten. Es hat ein Eingabefeld, in das ein Datum eingegeben werden soll. Mit dem Zusatz `onblur="javascript:validate()"` an diesem Feld wird festgelegt, dass sobald der Cursor dieses Feld verlassen hat, die JavaScript-Funktion `validate()` aufgerufen werden soll. Diese Funktion ist in dem durch das öffnende und schließende `script`-Tag umschlossenen Bereich definiert.

JavaScript hat gewisse Ähnlichkeiten zu **Java**, unterscheidet sich aber auch in grundsätzlichen Dingen davon. So gibt es in JavaScript das bei Java unbekannte Schlüsselwort `function`, das eine Funktion einleitet. Auch bietet JavaScript Zugriff auf Bereiche der HTML-Seite, in der eine Funktion ausgeführt wird. JavaScript ist schwach typisiert und ähnelt in diesem Aspekt Visual Basic.

Eine neue Variable wird durch das Konstrukt `var <variablenName>;` definiert, wobei `<variablenName>` der Name der neuen Variable ist. An einer Variable wird kein expliziter Typ festgemacht. Wird der Variable nun eine Zeichenkette zugewiesen, so hat sie von nun an den Typ Zeichenkette. Wird ihr später ein Array oder eine Zahl zugewiesen, so wechselt sie entsprechend ihren Typ. Wenn eine Variable, die eine Zeichenkette enthält, bei einer Rechenoperation benutzt wird, wird automatisch versucht, die Variable zum entsprechenden Typ zu konvertieren.

Die Ähnlichkeiten zu Java tauchen bei der Struktur und den Kontrollanweisungen auf. Geschweifte Klammern und Semikola werden wie in Java benutzt. Die `for`- und die `if`-Anweisung sind auch die gleichen wie in Java. Sogar einige aus Java bekannte Klassen existieren in JavaScript und haben ähnliche Operationen. `String` und `Date` sind hier als die prominentesten zu nennen, wobei `String` eine im Hinblick auf reguläre Ausdrücke erweiterte Klasse ist.

Doch nun zurück zum Beispiel. Gleich die erste Zeile der Funktion `validate()` zeigt eine der Besonderheiten von JavaScript. Hier wird der Variablen `validityMarker` ein Element aus der HTML-Seite[37] zugewiesen, das über seine ID gesucht wird. Dieses Element ist das `img`-Tag[38] in der HTML-Seite.

In `validate()` wird nun in der `if`-Anweisung eine andere Funktion `isValid()` gerufen, deren Ergebnis wie ein Wahrheitswert behandelt wird. Diese Funktion holt sich mit `document.vform.datum.value` den Inhalt des Eingabefeldes und führt auf diesem die Funktion `match()` aus, die alle Stellen aus der Zeichenkette heraussucht, die zum übergebenen regulären Ausdruck passen. Zu dem hier gezeigten regulären Ausdruck[39] passt alles, was aus ein bis zwei Ziffern für den Tag, gefolgt von einem Punkt, gefolgt von ein bis zwei weiteren Ziffern für den Monat, gefolgt von einem weiteren Punkt und den vier Ziffern für das Jahr besteht[40]. Da ein Aufruf von `match()` ein Array von Treffern zurückgibt, wird geprüft, ob dieses Array nur ein Element enthält und ob die Länge dieses Elementes der Länge der Eingabe entspricht, denn dann steht in dem Feld genau ein Datum und nichts anderes.

---

[37] `document` ist in JavaScript das Objekt, das für die HTML-Seite steht.

[38] Über das `img`-Tag lassen sich in HTML-Seiten Bilder einbinden

[39] In JavaScript wird für reguläre Ausdrücke die in Perl verwendete Syntax benutzt. Ein gutes Buch zum Thema Perl ist [Wall 00]. [Münz 01] bietet eine gute Übersicht über JavaScript und damit auch eine kurze Zusammenfassung der Syntax der regulären Ausdrücke.

[40] Um das Beispiel übersichtlich zu halten wurde darauf verzichtet zusätzlich zu prüfen, ob das eingegebene Datum ein gültiges Datum ist

Obwohl die Operation `isValid` keinen Rückgabewert spezifiziert, was keine Funktion in JavaScript tut, kann der Wahrheitswert der Auswertung zurückgegeben und in `validate` benutzt werden. Je nach Ergebnis der Auswertung setzt validate am `img`-Tag, auf das die Variable `validityMarker` verweist, ein Bild und in das Attribut `title` einen Text, der im Browser angezeigt wird, wenn sich der Mauszeiger über dem Bild befindet.

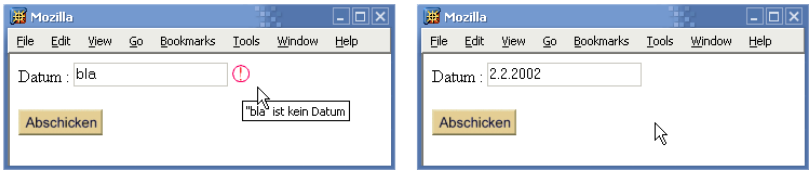

Abbildung A.1 Beispiel zur sofortigen Validierung. Links: Nach einer fehlerhaften Eingabe; Rechts: Nach einer validen Eingabe

Das Beispiel ist in Abbildung A.1 zu sehen. Die linke Seite zeigt die HTML-Seite nachdem eine fehlerhafte Eingabe gemacht wurde, während die rechte Seite dieselbe HTML-Seite bei einer validen Eingabe zeigt.

A.2 Beispiel zur Verwendung der Servlet-API

In diesem Abschnitt wird ein Beispiel zur Verwendung der in Abschnitt 3.4 angesprochenen Servlet-API vorgestellt und erklärt.

```
public class LoginServlet extends HttpServlet
{
  protected void doGet (HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException
  {
    response.setContentType("text/html");
    PrintWriter writer = response.getWriter();
    writer.println("<html>");
    writer.println("  <head>");
    writer.println("    <title>Login</title>");
    writer.println("  </head>");
    writer.println("  <body>");
    writer.println("    Hallo, Benutzer "
                   + request.getParameter("username") + "!");
    writer.println("  </body>");
    writer.println("</html>");
    writer.flush();
    writer.close();
  }
}
```

Der Entwickler muss bei der Programmierung eines Servlets nur mindestens eine Operation der Oberklasse `HttpServlet` reimplementieren.

Im HTTP sind verschiedene Anfragearten möglich. Die am meisten verwendeten sind GET und POST. Bei der Anfrageart GET werden die Anfrageparameter direkt in der **URL**[41] mit angegeben. Das sieht dann zum Beispiel so aus

```
http://www.myserver.com/servlet/MyServlet?name=beeger
                                          &password=geheim&domain=DOM-WPS
```

Bei einer POST-Anfrage werden die Parameter für den Benutzer unsichtbar in der Anfrage verpackt und an den Server-Rechner geschickt. Wenn ein Servlet nur POST-Anfragen akzeptieren soll, muss der Entwickler die Operation `doPost(HttpServletRequest, HttpServletResponse)` reimplementieren. Sollen GET-Anfragen bearbeitet werden, wird `doGet(HttpServletRequest, HttpServletResponse)` reimplementiert. Beim Aufruf der Operationen werden jeweils ein Objekt, das für die Anfrage steht und alle Parameter der Anfrage enthält, und eines, das für die Antwort steht, und in welches das Servlet seine Antwort schreiben muss, als Parameter mitgegeben.

Das obige Servlet könnte zum Beispiel von der in Abschnitt 3.2 vorgestellten HTML-Seite aus aufgerufen werden. Diese Seite hat ein Formular mit mehreren Feldern, von denen eines `username` heißt. Dieses Feld wird im Servlet mit `request.getParameter("username")` ausgelesen und in eine dynamisch erzeugte HTML-Seite eingefügt. Im Browser würde beim `username` ,,beeger“ die Zeile ,,Hallo, Benutzer beeger!“ erscheinen.

A.3 JSP-Beispiel

In diesem Abschnitt wird ein Beispiel für eine JSP gezeigt. JavaServer Pages werden in Abschnitt 3.4 vorgestellt.

Dem Servlet aus Anhang A.2 entspricht die folgende JSP

```
<html>
  <head>
    <title>Login</title>
  </head>
  <body>
    Hallo, Benutzer <%=request.getParameter("username")%>!
  </body>
</html>
```

Dieses Beispiel zeigt eines von mehreren speziellen JSP-Tags, dessen Benutzung dazu führt, dass das Ergebnis des Codes zwischen `<%=` und `%>` in den `PrintWriter` geschrieben wird.

A.4 FormProc-Beispiel

Die folgende JSP verwendet FormProc (siehe Abschnitt 3.6)

```
<%@ page language="java" %>
<%@ page import="java.io.File" %>
<%@ page import="java.util.Iterator" %>
<%@ page import="org.formproc.*" %>
<%@ page import="org.formproc.validation.ValidationResultMap" %>
<%@ page import="rbdipl.PersonBean" %>

<jsp:useBean id="results" class="org.formproc.FormResult" scope="session"/>
<jsp:useBean id="person" class="rbdipl.PersonBean"  scope="session"/>
```

[41] URL = Universal Resource Locator. Eine URL gibt die eindeutige Adresse einer HTML-Seite im Internet an. Ein Beispiel für eine URL ist www.jwam.de

```
<jsp:useBean id="form" class="org.formproc.servlet.HttpForm" scope="session"/>
<jsp:useBean id="formManager" class="org.formproc.FormManager" scope="page"/>
<%
  form.setName("datumform");
  form.setTarget(person);
  formManager.configure(form);
%>

<html>
  <head>
  </head>
  <body>
    <form action="FormBearbeiter" method="POST">
      Datum : <input  name="geburtsdatum" type="text"
                    value="<%=person.getGeburtsdatum()%>">
      <%
        if(!results.getValidationResults().isValid("geburtsdatum"))
        {
      %>
        <img src="invalid.gif" border="0"
          title="&quot;<%=person.getGeburtsdatum()%>&quot;
                 ist kein gültiges Datum"/>
      <%}%>
      <br/><br/>
      <input type="submit" value="Abschicken"/>
    </form>
  </body>
</html>
```

Der erste Teil der JSP – alle Zeilen mit `<%@` am Anfang – stellt fest, dass die Programmiersprache, die hier ins HTML eingestreut ist, Java ist, und importiert einige benötigte Klassen.

Der zweite Teil – alle Zeilen mit `<jsp:usebean` – erzeugt einige Objekte. Einige davon sind in der Session – `scope="session"` – und eines ist nur auf der Seite sichtbar – `scope="page"`.

Hierauf wird das Objekt `form` vom Typ `HttpForm`, einer Klasse der FormProc-API, mit einem Namen und einem Zielobjekt versehen. Das Zielobjekt ist eine JavaBean, die nach dem Abschicken des Formulars mit den Daten aus dem Formular befüllt wird. Die Klasse `PersonBean` sieht wie folgt aus

```
package rbdipl;

public class PersonBean
{
  public String getGeburtsdatum ()
  {
    return _geburtsdatum;
  }

  public void setGeburtsdatum (String geburtsdatum)
  {
```

```
    _geburtsdatum = geburtsdatum;
  }

  private String _geburtsdatum = "";
}
```

Da diese Klasse eine `get`- und eine `set`-Operation für das Attribut `geburtsdatum` hat, ist sie JavaBeans konform und jedes Objekt, das von dieser Klasse erzeugt wird, ist eine JavaBean – oder kurz Bean.

Nach dem Setzen des Zielobjektes des Formulars, wird der `FormManager`, ebenfalls ein Teil von FormProc, mit dem `form` konfiguriert. Bei der Konfiguration wird zunächst eine Datei formproc.xml gelesen.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<config>
  <shared-validator name="datum">
    <validator type="class" classname="rbdipl.DatumValidator"/>
  </shared-validator>
  <form name="datumform"
        loader="com.anthonyeden.lib.resource.ClassPathResourceLoader"
        path="datumform.xml"
        monitor="true"/>
</config>
```

Diese Datei enthält zum einen die Zuordnung von Namen zu den zu verwendenden Validatoren und zum anderen die Zuordnung von Namen zu weiteren XML-Dateien, die Formulare beschreiben.

Der Validator für Datumsangaben ist eine Java-Klasse

```
package rbdipl;

import org.formproc.validation.FormElementValidator;
import org.formproc.validation.ValidationResult;
import org.formproc.validation.FormElementValidationResult;
import org.formproc.FormElement;
import org.formproc.FormData;

import de.jwamx.lang.domainvalue.dvDate;

public class DatumValidator extends FormElementValidator
{
  public ValidationResult validate (FormElement formElement,
    FormData formData) throws Exception
  {
    FormElementValidationResult result = null;
    if(dvDate.Factory.instance().isValid(formData.getValue().toString()))
    {
      result = new FormElementValidationResult(formData);
    }
    else
```

```
    {
      result = new FormElementValidationResult(formData, this);
    }
    return result;
  }
}
```

Diese Klasse verwendet die Fachwertklasse `dvDate` um ein Datum auf Gültigkeit zu überprüfen.
Die Definition des Formulars in der Datei ,,datumform.xml“ sieht wie folgt aus

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<form>
  <name>datumform</name>
  <element name="geburtsdatum">
    <validator type="shared" name="datum"/>
  </element>
</form>
```

Diese ordnet nun jedem Eingabefeld des Formulars den zu verwendenden Validator zu. Da der Typ des Validators, der dem Eingabefeld ,,geburtsdatum“ zugeordnet wird, `shared` ist, weiß der `FormManager`, dass er hier einen Validator benutzen soll, der in der Hauptkonfigurationsdatei formproc.xml mit dem Namen ,,datum“ angegeben wurde.
Das Formular wird nach einem Druck auf den Knopf ,,Abschicken“ an das Servlet ,,FormBearbeiter“ geschickt. Der Name ,,FormBearbeiter“ wird in einer weiteren Konfigurationsdatei web.xml mit der Klasse `FormBearbeiter` in Verbindung gebracht.

```
public class FormBearbeiter extends HttpServlet
{
  protected void doPost (HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
  {
   doGet(request, response);
  }

  protected void doGet (HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException
  {
    HttpSession session = request.getSession(true);
    HttpForm form = (HttpForm) session.getAttribute("form");
    try
    {
      FormResult results = form.process(request);
      if(!results.isValid())
      {
        session.setAttribute("results", results);
        getServletContext().getRequestDispatcher(
          "/index.jsp").forward(request, response);
      }
      else
```

```
      {
        getServletContext().getRequestDispatcher(
          "/formok.jsp").forward(request, response);
      }
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

Dieses Servlet stößt durch einen Aufruf von `form.process()` die Validierung des Formulars an, bei der in diesem Fall auch die bereits vorgestellte Klasse `DatumValidator` zum Einsatz kommt. Wurden bei der Validierung invalide Eingaben gefunden, ruft das Servlet die Seite mit dem Formular – index.jsp – erneut auf. Anderenfalls wird eine andere Seite aufgerufen. In diesem Beispiel ist dies eine einfache Seite, die eine Meldung enthält, dass das Formular richtig ausgefüllt wurde.

In der Seite index.jsp werden die Validierungsergebnisse ausgewertet. Ist das Feld richtig ausgefüllt worden oder wurde die Seite zum ersten Mal aufgerufen, so wird nur das Eingabefeld angezeigt. War die vorige Eingabe fehlerhaft, so wird hinter dem Eingabefeld, in dem die vorige Eingabe sichtbar ist, ein Bild eingefügt, das signalisiert, dass hier eine fehlerhafte Eingabe gemacht wurde. Mit dem Attribut `title` des Tags `img` wird eine Meldung angegeben, die angezeigt wird, sobald sich der Mauszeiger über dem Bild befindet.

A.5 Struts-Beispiel

Dieser Abschnitt stellt ein Beispiel für das in Abschnitt 3.7 besprochene Rahmenwerk Struts vor und gibt dabei einige tiefere Einblicke in die technischen Details.

A.5.1 Konfiguration

```
<struts-config>
  <form-bean name="loginForm"
             type="net.beeger.tpr.form.LoginForm">
  </form-bean>

  <form-bean name="userForm"
             type="org.apache.struts.action.DynaActionForm">
    <form-property name="username" type="java.lang.String"/>
    <form-property name="password" type="java.lang.String"/>
    <form-property name="firstName" type="java.lang.String"/>
    <form-property name="name" type="java.lang.String"/>
    <form-property name="birthday" type="java.util.Date"/>
  </form-bean>

  <actionMappings>
    <action path="/login"
            type="net.beeger.tpr.web.struts.action.LoginAction"
            name="loginForm"
```

```
            scope="request"
            input="/login.jsp">
      <forward name="failure" path="/loginFailed.jsp" />
    </action>
  </actionMappings>

  <global-forwards>
    <forward name="login" path="/login.jsp" />
    <forward name="weekView" path="/weekView.jsp" />
    <forward name="monthView" path="/monthView.jsp" />
    <forward name="userManagement" path="/userManagement.jsp" />
    <forward name="newUser" path="/newUser.jsp" />
  </global-forwards>
</struts-config>
```

Das vorangehende Listing zeigt die Konfiguration einer Struts-Anwendung.

Zuerst wird eine `ActionForm` mit dem Namen ,,loginForm“ konfiguriert. loginForm ist ein Exemplar der JavaBean-konformen Klasse `LoginForm`.

Die zweite `ActionForm` ist eine dynamische `ActionForm`. Hier wird die Klasse `DynaActionForm` verwendet, die bereits in Struts vorhanden ist. Die Attribute dieser `ActionForm` werden in den `form-property`-Tags im entsprechenden `form-bean`-Tag spezifiziert. Beim Attribut `birthday`, das vom Typ `Date` ist wird Struts beim Befüllen des Formulars aus den Anfrageparametern versuchen, die Zeichenkette des Anfrageparameters in ein Date-Objekt zu konvertieren.

Darauf folgend wird ein **Action-Mapping** definiert[42]. Das erste Attribut des `action`-Tags ist `path`. Struts ist so konfiguriert, dass es jede bei der Web-Anwendung ankommende Anfrage, deren URL auf ,,.do“ endet, abfängt. Eine Anfrage der Art

```
http://localhost:8080/tpr/login.do?username=rb&password=geheim
```

ist eine solche Anfrage, die von Struts bearbeitet werden würde. Wenn man nun das ,,.do“ und alles folgende weglässt und von dem Ergebnis alles vor dem letzen ,,/“ weglässt, dann erhält man hier ,,/login“, das der Pfad innerhalb von Struts ist. Genau diesem Pfad wird in einem Action-Mapping eine `Action` zugeordnet. Welchen Typ die `Action` haben soll, die Anfragen für einen bestimmten Pfad bearbeitet, wird über das `type`-Attribut am `action`-Tag festgelegt. Hier ist es die Klasse `LoginAction`, die von der Struts-Klasse `Action` erbt.

Über das `name`-Attribut wird der Name der `ActionForm` angegeben, die mit den Anfrageparametern gefüllt werden soll. Mit dem `scope`-Attribut wird angegeben, wo die `ActionForm`, nachdem sie mit den Anfrageparametern befüllt wurde, gesichert werden soll. Steht hier `request`, so steht die `ActionForm` nur für die Abarbeitung der aktuellen Anfrage zur Verfügung. Wenn hier jedoch `session` steht, wird die `ActionForm` in der Session gesichert und kann auch später noch verwendet werden. Für gewöhnlich werden `ActionForm`s nur für die Abarbeitung einer Anfrage benötigt. Sollen jedoch einmal eingegebene Daten auch später noch zugreifbar sein, bietet es sich an, die `ActionForm` in der Session zu sichern, wo sie für den Rest der Sitzung verfügbar ist.

Über das `input`-Attribut wird definiert, von welchen JSPs die hier konfigurierte `Action` aufgerufen wird. Hier gibt es auch die Möglichkeit, ein längeres Formular auf mehrere JSPs zu verteilen und dann insgesamt als eine Anfrage zu bearbeiten.

[42] In dem zwischen `<action-mappings>` und `</action-mappings>` eingefasstem Block können mehrere dieser Mappings definiert werden und es gibt auch einige Struts-Mappings, die bei jeder Struts-Web-Anwendung vorhanden sind. Diese wurden hier aus Gründen der Übersichtlichkeit weggelassen.

Schließlich enthält das `action`-Tag ein `forward`-Tag, mit dem ein symbolischer Name für eine Weiterleitung auf eine JSP definiert wird. Der Vorteil an diesen symbolischen Namen ist, dass die `Action` nicht geändert werden muss, wenn sich das Ziel ändert. Das entkoppelt die Anwendungslogik von der Präsentationsseite.

Weiterleitungen, die in einem `action`-Tag eingebettet sind, gelten nur für diese Action und können somit auch auf diese Action angepasste Namen haben. Häufig gibt es Weiterleitungen mit den Namen „success[43]“ und „failure[44]“, die verwendet werden, wenn eine `Action` erfolgreich oder nicht erfolgreich ausgeführt wurde. Da diese Namen im Code der `Action` verwendet werden, verbessert das die Lesbarkeit des Codes und macht die Struktur einer Anwedung erkennbarer.

Weiterleitungen, die an mehreren Stellen in der Anwendung verwendet werden, können im Bereich `global-forwards` definiert werden.

A.5.2 Die Action

Der Entwickler muss nur eine einzige Operation für eine `Action` implementieren. Das ist jene, die nun nach der Füllung der `ActionForm` als nächstes aufgerufen wird. Hier ein Beispiel für eine solche Operation in der `LoginAction`

```
public ActionForward execute (ActionMapping mapping,
                              ActionForm form,
                              HttpServletRequest request,
                              HttpServletResponse response)
   throws Exception
{
  ActionForward result = mapping.findForward("failure");
  if(form instanceof LoginForm)
  {
    LoginForm actionForm = (LoginForm) form;
    String username = (String) actionForm.domainValue("username");
    String password = (String) actionForm.domainValue("password");
    HttpSession session = request.getSession(true);
    UserManagement userManagement = (UserManagement) session.getAttribut("users");
    if(userManagement.isValidLogin(username, password))
    {
      session.setAttribute("username", username);
      User user = userManagement.user(username);
      if(user.group().equals(GroupDV.Factory.instance().rootGroup()))
      {
        result = mapping.findForward("userManagement");
      }
      else if(user.startView().equals(CalendarViewDV.Factory.instance().dayView()))
      {
        result = mapping.findForward("weekView");
      }
      else if(user.startView().equals(CalendarViewDV.Factory.instance().monthView()))
      {
```

[43] engl.: success = Erfolg
[44] engl.: failure = Mißerfolg

```
        result = mapping.findForward("monthView");
      }
      else
      {
        result = mapping.findForward("weekView");
      }
    }
    else
    {
      result = mapping.findForward("failure");
    }
  }

  return result;
}
```

Die Operation `execute` enthält den ersten Java-Code, den der Entwickler selbst schreiben muss. Die `Action`s enthalten die Anwendungslogik einer Struts-Anwendung.

Die oben gezeigte `execute`-Operation holt aus der `LoginForm` die beiden Attribute `username` und `password` und prüft deren Gültigkeit für einen Login. Ist die Gültigkeit festgestellt, wird ein passendes `User`-Objekt geholt, an dem dann ermittelt wird, welche Ansicht der Anwendung nach einem Login angezeigt werden soll. Die Operation `execute` gibt ein `ActionForward` zurück, das genügend Informationen bietet, dass zu der nächsten Seite umgeleitet werden kann.

A.6 Beispiel für die Verwendung des Validierungs-Rahmenwerks von Struts

In diesem Abschnitt wird ein Beispiel für die Verwendung des in Abschnitt 3.8 besprochenen Validierungs-Rahmenwerkes von Struts gezeigt.

Um das Rahmenwerk benutzen zu können, muss man es in Struts über ein Struts-eigenes Plugin[45]-Konzept einstöpseln. Auch dies wird über die Struts-Konfigurationsdatei erledigt

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property property="pathname"
                value="/WEB-INF/validator-rules.xml"/>
  <set-property property="pathname"
                value="/WEB-INF/validation.xml"/>
</plug-in>
```

Hier wird festgelegt, dass ein Plugin benutzt werden soll, dessen Schnittstelle zu Struts die Klasse `ValidatorPlugIn` darstellt. Dem Objekt, das von dieser Klasse durch Struts erzeugt wird, werden zwei `pathname`-Parameter übergeben, die angeben, in welchen XML-Dateien sich die Konfiguration des Plugins befinden. Plugins werden beim Starten einer Web-Anwendung geladen und mit den für sie angegebenen Parametern konfiguriert. Die Parameter sind von Plugin zu Plugin verschieden.

`validator-rules.xml` enthält in diesem Beispiel einige generische Validatordefinitionen, die in vielen Web-Anwendungen zum Einsatz kommen können. Hier eine solche Definition

[45] engl.: plug-in = Einschub

```
<form-validation>
  <global>
    <validator name="mask"
      classname="org.apache.struts.util.StrutsValidator"
      method="validateMask"
      methodParams="java.lang.Object,
                    org.apache.commons.validator.ValidatorAction,
                    org.apache.commons.validator.Field,
                    org.apache.struts.action.ActionErrors,
                    javax.servlet.http.HttpServletRequest"
      depends="required"
      msg="errors.invalid">
      <javascript><![CDATA[

        function validateMask(form)
        {
          var bValid = true;
          var focusField = null;
          var i = 0;
          var fields = new Array();
          oMasked = new mask();

          for (x in oMasked)
          {
            if ((form[oMasked[x][0]].type == 'text'
                 || form[oMasked[x][0]].type == 'textarea'
                 || form[oMasked[x][0]].type == 'password')
                && form[oMasked[x][0]].value.length > 0)
            {
              if (!matchPattern(form[oMasked[x][0]].value, oMasked[x][2]("mask")))
              {
                if (i == 0) focusField = form[oMasked[x][0]];
                fields[i++] = oMasked[x][1];
                bValid = false;
              }
            }
          }

          if (fields.length > 0)
          {
             focusField.focus();
             alert(fields.join('\n'));
          }

          return bValid;
        }

        function matchPattern(value, mask)
        {
           var bMatched = mask.exec(value);
```

```
            if(!bMatched)
            {
               return false;
            }

            return true;
         }

      ]]>
      </javascript>
    </validator>
  </global>
</form-validation>
```

Wie zu sehen ist, besteht der Validator, der überprüft, ob das, was in ein Feld gefüllt wurde, einem bestimmten angegebenen Muster entspricht, aus zwei Komponenten. Es gibt hier eine JavaScript-Komponenten, die vor dem Abschicken eines Formulars zum Einsatz kommen kann, und es gibt eine Komponente, die auf dem Server-Rechner eingesetzt wird.

Die Konfiguration, welche Validatoren für welche Felder welcher Formulare zum Einsatz kommen sollen, ist in diesem Beispiel in der XML-Datei `validation.xml` abgelegt. Hier ein Ausschnitt aus dieser Konfigurationsdatei

```
<form-validation>
  <formset>
    <form name="registrationForm">
      <field property="phone" depends="mask">
        <arg0 key="registrationForm.phone.displayname"/>
        <var>
         <var-name>mask</var-name>
         <var-value>${phone}</var-value>
        </var>
      </field>
    </form>
  </formset>
</form-validation>
```

Hier wird festgelegt, dass in einem Formular mit dem Namen ,,registrationForm“ ein Feld mit dem Namen ,,phone“ durch den mask-Validator validiert werden soll. Zusätzlich wird hier mit ,,arg0“ der erste Parameter belegt, der beim Aufruf der Validierungsoperation übergeben wird. Das `key`-Attribut teilt Struts mit, dass es die lokalisierte Benennung des Feldes unter dem gegebenen Schlüssel findet. Der `var`-Block weist einer JavaScript-Variablen `mask` den Namen des Feldes zu. Über diesen Namen findet die JavaScript-Funktion dann das Feld und validiert es schließlich.

A.7 wingS-Beispiel

Dieser Abschnitt stellt ein Beispiel für die Verwendung des in Abschnitt 3.9 besprochenen Rahmenwerkes wingS vor.

Das folgende Listing zeigt eine kleine wingS-Web-Anwendung. Abbilung A.2 zeigt die Anwendung im Betrieb.

```
public class WingsTest
{
  public WingsTest ()
  {
    SFrame frame = new SFrame("WingS-Test");
    final SForm contentPane = new SForm();
    contentPane.setLayout(new SBorderLayout());

    final SPanel mainPanel = new SPanel(new SGridLayout(2,2));
    mainPanel.add(new SLabel("Username : "));
    final STextField username = new STextField();
    username.setName("username");
    username.setColumns(30);
    mainPanel.add(username);

    mainPanel.add(new SLabel("Password : "));
    SPasswordField password = new SPasswordField();
    password.setColumns(30);
    password.setName("password");
    mainPanel.add(password);

    final SButton loginButton = new SButton("Login");
    final SPanel buttonPanel = new SPanel(new SFlowLayout(SConstants.RIGHT_ALIGN));
    buttonPanel.add(loginButton);

    contentPane.add(mainPanel, SBorderLayout.CENTER);
    contentPane.add(buttonPanel, SBorderLayout.SOUTH);

    loginButton.addActionListener(new ActionListener()
    {
      public void actionPerformed (ActionEvent e)
      {
        contentPane.add(new SLabel("Hallo, " + username.getText()));
        contentPane.remove(mainPanel);
        contentPane.remove(buttonPanel);
      }
    });

    frame.getContentPane().add(contentPane, SBorderLayout.CENTER);
    frame.show();
  }
}
```

Zum Vergleich folgt hier die gleiche Anwendung als Swing-Desktop-Anwendung. Abbildung A.3 zeigt diese Anwendung im Betrieb.

```
public class SwingTest
{
  public static void main (String[] args)
  {
```

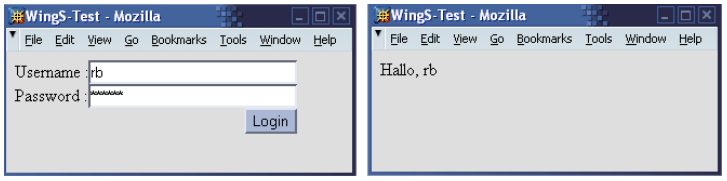

Abbildung A.2 wingS-Web-Anwendung (links vor, rechts nach dem Drücken des Login-Knopfes)

```
JFrame frame = new JFrame("Swing-Test");
final JPanel contentPane = new JPanel();
contentPane.setLayout(new BorderLayout());

final JPanel mainPanel = new JPanel(new GridLayout(2,2));
mainPanel.add(new JLabel("Username : "));
final JTextField username = new JTextField();
username.setName("username");
username.setColumns(30);
mainPanel.add(username);

mainPanel.add(new JLabel("Password : "));
JPasswordField password = new JPasswordField();
password.setColumns(30);
password.setName("password");
mainPanel.add(password);

final JButton loginButton = new JButton("Login");
final JPanel buttonPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
buttonPanel.add(loginButton);

contentPane.add(mainPanel, BorderLayout.CENTER);
contentPane.add(buttonPanel, BorderLayout.SOUTH);

loginButton.addActionListener(new ActionListener()
{
  public void actionPerformed (ActionEvent e)
  {
    contentPane.add(new JLabel("Hallo, " + username.getText()),
                    BorderLayout.CENTER);
    contentPane.remove(mainPanel);
    contentPane.remove(buttonPanel);
    contentPane.revalidate();
  }
});

frame.getContentPane().add(contentPane, BorderLayout.CENTER);
frame.setSize(300, 110);
frame.show();
```

```
  }
}
```

Abbildung A.3 Swing-Desktop-Anwendung (links vor, rechts nach dem Drücken des Login-Knopfes)

Im Code unterscheiden sich beide Anwendungen sehr wenig. Bei der wingS-Anwendung werden lediglich andere Klassen verwendet als bei der Swing-Anwendung. Diese Klassen haben jedoch – soweit das hier zu sehen ist – die gleiche Schnittstelle, so dass sich die Widgets auf die gleiche Art und Weise verwenden lassen.

Die Swing-Anwendung enthält zwei zusätzliche Zeilen, von denen eine bewirkt, dass das Fenster eine vernünftige Ausgangsgröße hat. Die andere Zeile erzwingt das Neuzeichnen des Fensterinhalts, denn sonst würde sich nichts an der Ansicht ändern nachdem der Knopf betätigt wurde.

Die Konfiguration des `WingsServlet` erfolgt in der `web.xml`:

```
<web-app>
  <servlet>
    <servlet-name>WingsTest</servlet-name>
    <servlet-class>org.wings.session.WingServlet</servlet-class>
    <init-param>
      <param-name>wings.mainclass</param-name>
      <param-value>net.beeger.tpr.web.wings.WingsTest</param-value>
    </init-param>
    <init-param>
      <param-name>wings.lookandfeel.deploy</param-name>
      <param-value>/css1.jar</param-value>
    </init-param>
    <init-param>
      <param-name>wings.lookandfeel.name</param-name>
      <param-value>xhtml/css1</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>WingsTest</servlet-name>
    <url-pattern>/WingsTest/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Der Parameter `wings.mainclass` gibt hier die Hauptklasse der Anwendung an. Es wird je ein Objekt pro Sitzung von dieser Klasse erzeugt. Diese Klasse muss also nicht Thread-sicher sein, was ja bei Servlets der Fall ist. Hier können wie in Desktop-Anwendungen Attributvariablen verwendet werden.

Zum Abschluss folgt hier das Beispiel nochmal mit der Verwendung einer Vorlage. Zunächst die Vorlage

```
<table  style="width:100%;height:100%" border="0" cellpadding="0" cellspacing="0">

  <tr>
    <td align="left" class="header">
      <img src="../images/logo1.png" alt="Time / People / Resources - Logo"
           width="184" height="80">
    </td>
    <td class="header"
      style="width:100%;background-image:url(../images/logo2.png);
             background-repeat:repeat-x">
    </td>
  </tr>
  <tr>
    <td colspan="2"  style="width:100%%;height:80%;">
      <table style="width:100%;height:100%;" border="0"
             cellpadding="0" cellspacing="0">
        <tr>
          <td style="width:5px"></td>
          <td align="left" >
            <table>
              <tr>
                <td><object name="usernameLabel" ></object></td>
                <td><object name="usernameFillIn" ></object></td>
              </tr>
              <tr>
                <td><object name="passwordLabel" ></object></td>
                <td><object name="passwordFillIn" ></object></td>
              </tr>
            </table>
            <object name="loginButton" ></object>
          </td>
          <td style="width:5px"></td>
        </tr>
      </table>
    </td>
  </tr>
  <tr>
    <td colspan="2" align="center" style="height:10px">
      Copyright &copy; 2002 by Robert F. Beeger
    </td>
  </tr>
</table>
```

Und hier die angepasste Klasse

```
public class WingsTest
{
  public WingsTest ()
  {
    final SFrame frame = new SFrame("WingS-Test");
```

```
    File cssFile = new File(
      getClass().getResource("template/default.css").getFile());
    frame.headers().add(new Link("stylesheet", null,
       "text/css", null, new FileResource(cssFile, "css", "text/css")));

    final SForm contentPane = new SForm();
    try
    {
      contentPane.setLayout(new STemplateLayout(
        getClass().getResource("template/login.thtml")));
    }
    catch (java.io.IOException e)
    {
      e.printStackTrace();
    }

    contentPane.add(new SLabel("Username : "), "usernameLabel");
    final STextField username = new STextField();
    username.setName("username");
    username.setColumns(30);
    contentPane.add(username, "usernameFillIn");

    contentPane.add(new SLabel("Password : "), "passwordLabel");
    SPasswordField password = new SPasswordField();
    password.setColumns(30);
    password.setName("password");
    contentPane.add(password, "passwordFillIn");

    final SButton loginButton = new SButton("Login");
    contentPane.add(loginButton, "loginButton");

    loginButton.addActionListener(new ActionListener()
    {
      public void actionPerformed (ActionEvent e)
      {
        SOptionPane.showConfirmDialog(frame,
          "Hallo, " + username.getText(), "Login");
      }
    });

    frame.getContentPane().add(contentPane, SBorderLayout.CENTER);
    frame.show();
  }
}
```

Die einzelnen Widgets werden einfach mit Aufrufen wie `contentPane.add(username, "usernameFillIn")` hinzugefügt. Der zusätzliche Name – hier „usernameFillIn“ – findet sich auch in der Vorlage als `name`-Attribut eines `object`-Tags wieder.

Zwei Screenshots des aktualisierten Beispiels sind in Abbildung A.4 zu sehen.

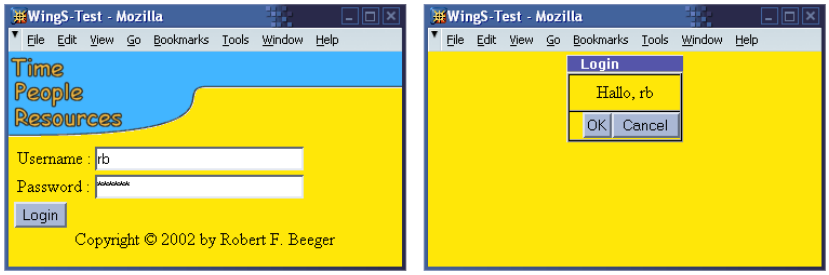

Abbildung A.4 wingS-Web-Anwendung mit STemplateLayout (links vor, rechts nach dem Drücken des Login-Knopfes)

A.8 JSF-Lebenszyklus

Dieser Abschnitt gibt einen detaillierten Einblick in die Phasen eines Lebenszyklus einer JavaServer Faces Anwendung. JavaServer Faces wird in Abschnitt 3.10 eingeführt.

Damit die Anwendungslogik die Daten aus den Oberflächenkomponenten bei der Bearbeitung einer Anfrage herausbekommt und die Komponenten einer Antwort-Seite befüllen kann, definiert die JSF-Spezifikation einen **Lebenszyklus**, der durchlaufen wird. Dieser Zyklus ist in Abbildung A.5 skizziert. Er wird bei jeder Anfrage durchlaufen.

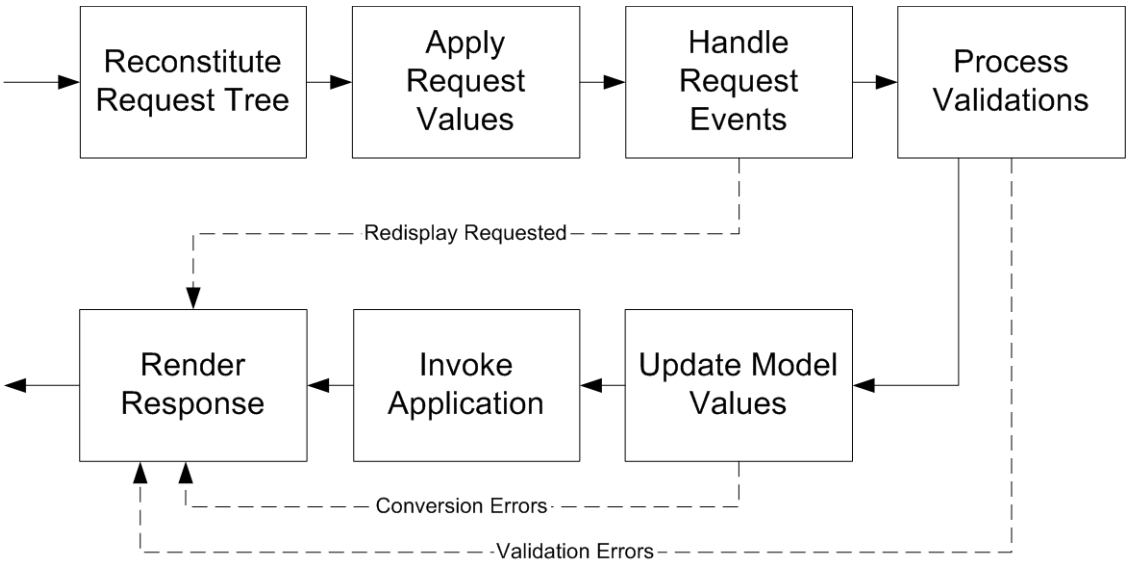

Abbildung A.5 Lebenszyklus einer JSF-Web-Anwendung (nach [McClanahan 02])

In der ersten Phase des Zyklus – **Reconstitute Request Tree** – wird eine baumartige Struktur aus den Oberflächenkomponenten der aufrufenden JSP zusammengebaut. Wenn es zum Beispiel in der aufrufenden JSP ein HTML-Formular mit dem Namen ,,login“ und in diesem zwei Eingabefelder ,,username“ und ,,password“ gibt, dann wird ,,login“ die Wurzel des Baumes sein und die beiden Eingabefelder werden unter diesem Knoten aufgeführt. Der Baum kann auch über explizite Namensangaben in der JSP beeinflusst werden. Heißt zum Beispiel ein Element der JSP ,,/login/user/name“ dann wird ein Wurzelknoten ,,login“ mit einem Kindknoten ,,user“ angelegt, der das Element mit dem Namen ,,name“ als Kind hat.

In der zweiten Phase – **Apply Request Values** – wird jede der Komponenten in der Baumstruktur aufgerufen, worauf diese sich dann die entsprechenden Anfrageparameter aus der Anfrage nimmt und an sich setzt. In dieser Phase können auch Konvertierungen der Zeichenketten, die die Anfrageparameter enthalten, in speziellerer Datentypen erfolgen. Gibt es zum Beispiel eine Oberflächenkomponente, die für die Eingabe von Datumsangaben spezialisiert ist, wird diese die geholten Anfrageparameter in ein Datum-Objekt umwandeln. Dabei können natürlich Fehler auftreten, wenn der Benutzer zum Beispiel ein ungültiges Datum eingetragen hat. Ist dies der Fall, wird die Oberflächenkomponente eine Nachricht an einer zentralen Stelle – dem `FacesContext` – hinterlassen, die die Art des Fehlers beschreibt. In dieser Phase haben Oberflächenkomponenten die Gelegenheit ,,Request Events“ anzumelden, die in der hierauf folgenden Phase ausgeführt werden. Diese Möglichkeit wird hauptsächlich von komplexeren Oberflächenkomponenten genutzt werden. Ein Beispiel hierfür wäre eine Baumansicht, die beim Anklicken eines Eintrags die Kindelemente ein- oder ausblendet. Auch können hier ,,Application Events“ angemeldet werden, die beim Aufrufen der Anwendungslogik in der sechsten Phase verarbeitet werden. Diese Möglichkeit wird in der Referenzimplementierung hauptsächlich von der Oberflächenkomponente `UICommand` verwendet, um zu signalisieren, dass der Benutzer auf den entsprechenden Knopf oder Verweis geklickt hat.

In der dritten Phase – **Handle Request Events** – wird unter anderem nachgesehen, ob eine Oberflächenkomponente beim Holen und Konvertieren der Daten einen Fehler entdeckt und eine entsprechende Nachricht hinterlassen hat. In diesem Fall wird direkt in Phase 7 umgeleitet, in der die JSP erneut mit den Fehlernachrichten angezeigt wird. Die ,,Request Events“, die in der vorigen Phase angemeldet wurden, werden in dieser Phase ausgeführt. Auch sie haben die Möglichkeit, zur Antwortgenerierung umzuleiten. Bei dem erwähnten Baumansicht-Beispiel wäre das beim Anklicken eines Eintrages der Fall. In diesem Fall soll nicht der Umweg über die Anwendungslogik gemacht werden, da durch diese Aktion nichts anwendungsrelevantes geschieht. Es ändert sich lediglich die Anzeige einer Oberflächenkomponente.

In der vierten Phase – **Process Validations** – werden die an den Oberflächenkomponenten angemeldeten Validatoren ausgeführt. Diese Validatoren werden typischerweise in der JSP durch spezielle Tags angegeben, die in die entsprechenden Renderer-Tags verschachtelt sind. Die Referenzimplementierung bietet Validatoren, die prüfen, ob eine Oberflächenkomponente gefüllt ist, ob ihr Inhalt eine bestimmte Länge hat und ähnliches. Wie schon die Oberflächenkomponenten beim Konvertieren der Anfrageparameter können die Validatoren beim Validieren im Fehlerfall eine entsprechende Nachricht im `FacesContext` hinterlegen. Hat mindestens eine Validierung einen Fehler aufgedeckt, wird von dieser Phase direkt in Phase 7 umgeleitet, in der die aufrufende JSP mit den Fehlernachrichten neu aufgerufen wird.

In der fünften Phase – **Update Model Values** – werden die Daten der Oberflächenkomponenten, von denen jetzt angenommen wird, dass sie syntaktisch und semantisch korrekt sind, in das Model geschrieben. Das Model kann aus beliebig vielen Objekten JavaBeans-konformer Klassen bestehen. In der JSP wird an jedem Renderer-Tag angegeben welches Attribut die entsprechende Oberflächenkomponente an welchem JavaBeans-Objekt befüllt. Auch bei der Füllung des Models können Fehler auftreten, die in der bereits bekannten Manier behandelt werden.

In der sechsten Phase – **Invoke Application** – wird schließlich die Anwendungslogik aufgerufen. Der Entwickler muss hier eine Klasse zur Verfügung stellen, die die Schnittstelle `ApplicationHandler` implementiert. Die einzige hier zu implementierende Operation ist `processEvent(FacesContext, FacesEvent)`. Das heißt also, dass an dieser Stelle nicht einfach der Anwendungslogik die Kontrolle überlassen wird, sondern, dass hier für jedes angemeldete ,,Application Event“ ein erneuter Aufruf erfolgt. Die Referenzimplementierung bietet derzeit nur zwei Ausprägungen derartiger Ereignisse. Die eine ist `CommandEvent`, die von `UICommand` zum Signalisieren eines Klicks auf den

entsprechenden Knopf oder Verweis verwendet wird. Die andere ist `FormEvent` und ist eine Spezialisierung der vorigen. Sie wird benutzt, wenn ein Formular an den Serverrechner verschickt wurde, was meistens durch den Klick auf einen Knopf angestoßen wird. Komplexere Oberflächenelemente werden hier andere Ereignisse anbieten.

`processEvent()` gibt einen `boolean` zurück. Solange dieser `false` ist, wird `processEvent()` mit dem nächsten Ereignis aufgerufen. Sobald `processEvent()` jedoch `true` zurückgibt, ist das das Zeichen, dass die Anwendungslogik fertig ist, egal ob noch andere Ereignisse folgen oder nicht. Tritt zum Beispiel bei der Abarbeitung eines Ereignisses ein Fehler auf, macht es unter Umständen keinen Sinn, mit der Abarbeitung der anderen Ereignisse fortzufahren. In diesem Fall werden Fehlernachrichten im `FacesContext` hinterlegt, und die Ausführung der letzten Phase angestoßen.

Wenn die Abarbeitung aller Ereignisse fehlerlos verläuft, wird die Anwendung als Antwort eine andere JSP anzeigen lassen wollen. Für diese neue Seite müssen andere Oberflächenelemente erzeugt und mit anderen Daten gefüllt werden. Deshalb hat die Anwendung hier die Möglichkeit, die Baumstruktur zu ändern, bestehende Oberflächenelemente zu entfernen und neue einzufügen.

In der siebten und letzten Phase – **Render Response** – wird die JSP aufgerufen, die als nächstes angezeigt werden soll. Die speziellen Tags, die in der JSP verwendet werden und angeben welche Renderer welcher Oberflächenkomponenten zur Generierung der einzelnen Bestandteile der JSP verwendet werden sollen, rufen die Renderer auf, die auf die Oberflächenelemente in der Baumstruktur zugreifen und deren Daten in die resultierende HTML-Seite mit dem für die einzelnen Oberflächenkomponenten nötigen Drumherum schreiben.

Damit ist ein Zyklus beendet und geht bei der nächsten Anfrage von vorne los.

A.9 JSF-Beispiel

Dieser Abschnitt stellt ein Beispiel für die Verwendung der in Abschnitt 3.10 besprochenen Web-Technologie JavaServer Faces vor.

Das JSF-Beispiel ist eine Variante des bereits bekannte Login-Beispiel. Den Anfang macht die Login-Seite

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<%@ page language="java" %>

<%@ taglib uri="/html_basic.tld" prefix="faces" %>

<html>
  <faces:usefaces>
    <head>
      <title>Login Into Time / People / Resources</title>
      <link rel="stylesheet" type="text/css" href="/tprj/default.css">
    </head>
    <body>
      <jsp:include page="include/header.jsp" />
      <jsp:useBean id="loginBean" scope="session"
                   class="net.beeger.jsftest.LoginBean" />
      <faces:form id="loginForm" formName="loginForm" >
        <table>
```

```
          <tr>
            <td>Username</td>
            <td>
              <faces:textentry_input id="username"
                                     modelReference="loginBean.username" />
            </td>
          </tr>
          <tr>
            <td>Password</td>
            <td>
              <faces:textentry_input id="password"
                                     modelReference="loginBean.password" />
            </td>
          </tr>
        </table>
        <faces:command_button id="loginButton" commandName="login"
                              label="Login" />
     </faces:form>
     <jsp:include page="include/footer.jsp" />
    </body>
  </faces:usefaces>
</html>
```

Diese JSP verwendet die Tag-Library `html-basic.tld`, die zu der Referenzimplementierung der JavaServer Faces gehört. Vier der dort definierten Tags werden hier verwendet. Das erste ist das `usefaces`-Tag, das allgemeine Initialisierungen für die Benutzung von JSF vornimmt. Dieses darf in keiner JSP fehlen, die mindestens einen der in `html-basic.tld` definierten Tags benutzt. Das öffnende Tag muss vor dem ersten anderem JSF-Tag auftauchen und das schließende nach dem letzten anderen JSF-Tag. Das `form`-Tag erzeugt ein HTML-Formtag und erledigt einige JSF-spezifische Aufgaben im Zusammenhang mit HTML-Formularen.

Das folgende JSF-Tag ist das `textentry_input` und ist eines der Renderer-Tags. Dieses hier wählt für ein `UIInput`-Oberflächenelement den `textentry`-Renderer aus. Das Attribut `modelReference` gibt an, auf welches Model das Oberflächenelement zugreifen soll. Ein Wert `loginBean.username` bedeutet, dass das entsprechende Oberflächenelement auf das Attribut `username` der JavaBean `loginBean` zugreifen soll. Beim Aufruf der Seite wird das Oberflächenelement den derzeitigen Wert des Attributes holen und zur Anzeige bringen. Nach dem Abschicken des Formulars, wird das Oberflächenelement dieses Attribut mit dem Wert setzen, den der Benutzer eingegeben hat.

Das letzte Tag – `command_button` – stellt schließlich ein `UICommand` als Knopf dar.

Außer `usefaces` hat jedes der genannten Tags ein Attribut `id`. Die dort angegebene ID wird dazu verwendet, um eine Beziehung zwischen den Elementen der JSP und der baumartigen Struktur der Oberflächenelemente herzustellen. Deutlicher wird das, wenn man sich einen Teil des HTML-Codes ansieht, der aus der JSP-Seite generiert wird

```
<input TYPE="text" NAME="/loginForm/username">
```

Dieser kleine Ausschnitt ist das, was an HTML-Code aus dem `textentry_input` mit der ID ,,username“ entsteht. Hervorzuheben ist hier, dass die ID nicht einfach zum Namen des `input`-Tags wird, sondern, dass der Name sich aus dem Namen des Formulars, in dem sich das Element befindet, einem ,,/“ und dem Namen des Elementes selbst zusammensetzt, was zur Erhaltung der Baumstruktur führt.

Die `LoginBean` ist wie bereits erwähnt eine einfache JavaBean

```
public class LoginBean
{
  public LoginBean ()
  {
  }

  public String getUsername ()
  {
    return _username;
  }

  public void setUsername (String username)
  {
    _username = username;
  }

  public String getPassword ()
  {
    return _password;
  }

  public void setPassword (String password)
  {
    _password = password;
  }

  private String _username;
  private String _password;
}
```

Und nun ein Blick auf den `ApplicationHandler`

```
public class TestApplicationHandler implements ApplicationHandler
{
  public boolean processEvent (FacesContext facesContext, FacesEvent event)
  {
    boolean result = false;
    if(event instanceof FormEvent)
    {
      FormEvent formEvent = (FormEvent) event;
      if(formEvent.getCommandName().equals("login")
         && formEvent.getFormName().equals("loginForm"))
      {
        TreeFactory treeFactory = (TreeFactory)
        FactoryFinder.getFactory(FactoryFinder.TREE_FACTORY);
        ServletContext sc = facesContext.getServletContext();
        facesContext.setResponseTree(treeFactory.getTree(sc,"/welcome.jsp"));
        result = true;
```

```
      }
    }
    return result;
  }
}
```

Hier werden alle Ereignisse außer einem `FormEvent`, das das Abschicken eines Formulars signalisiert, ignoriert. Ist das Ereignis ein `FormEvent`, so wird geprüft, ob das Formular ,,loginForm“ heißt und der `commandName` ,,login“ ist. Da beides nach dem Klicken auf den Knopf mit der Aufschrift ,,Login“ der oben gezeigten JSP zutrifft, wird mit dem auf diese Prüfung folgenden Konstrukt ein neuer Oberflächenkomponentenbaum erzeugt und als Antwort-Baum gesetzt. Hier wäre nun die Gelegenheit gegeben, andere Oberflächenkomponenten in den Baum zu stecken, die dann von der aufgerufenen JSP verwendet werden könnten. Dies wird hier jedoch nicht getan. Der neu erzeugte Baum bekommt als oberste ID ,,/welcome.jsp“ zugewiesen.

Zum Setzen dieser obersten ID, ist in der JSF-Spezifikation (siehe [McClanahan 02]) Folgendes zu lesen

> ,,Change the response component tree to select an outbound page different than the inbound page being processed, by creating a new Tree instance with a tree identifier that is meaningful to the ViewHandler being utilized, and saving it by calling the setResponseTree() method on the FacesContext instance for the current request.“

Diese ID soll also für einen `ViewHandler` verständlich sein. Das wird sich unter Umständen von einem zum anderen unterscheiden. Der in der Referenzimplementierung enthaltene kann etwas mit URLs und lokalen Pfadangaben anfangen, und so ist `/welcome.jsp` auch nichts anderes als ein Verweis auf eine andere JSP, die aufgerufen werden soll.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<%@ page language="java" %>
<%@ taglib uri="/html_basic.tld" prefix="faces" %>

<html>
  <faces:usefaces>
    <head>
      <title>Welcome</title>
      <link rel="stylesheet" type="text/css" href="/tprj/default.css">
    </head>
    <body>
      <jsp:include page="include/header.jsp" />
      <jsp:useBean id="loginBean" scope="session"
                   class="net.beeger.jsftest.LoginBean" />
      Hello, <faces:output_text id="username"
                                modelReference="loginBean.username" />
      <jsp:include page="include/footer.jsp" />
    </body>
  </faces:usefaces>
</html>
```

Das einzig neu an dieser Seite ist das `output_text`-Tag, das für ein `UIOutput` den `text`-Renderer verwendet.

Die Verwendung der Model-Referenzen macht diese Anwendung sehr einfach. Man muss sich nicht merken, wie die Baumstruktur aufgebaut ist. Außer zum Spezifizieren der nächsten JSP muss man sich gar nicht mit den Klassen befassen, die für die Baumstruktur verwendet werden. Es können in einer JSP Referenzen auf beliebig viele Beans gehalten werden. Die Verwendung dieser Beans bringt die JavaServer Faces ein Stück näher an die Fachlogik.

In einer größeren Anwendung wird man das Bearbeiten einzelner Ereignisse in andere Klassen auslagern und den `ApplicationHandler` nur als kleinen Verteiler konstruieren.

Eine Frage, die noch zu beantworten bleibt, ist die, wie das Ganze zusammengesteckt wird. Auch hier spielt die `web.xml` eine wichtige Rolle

```
<web-app>
  <context-param>
      <param-name>saveStateInClient</param-name>
      <param-value>false</param-value>
  </context-param>

  <listener>
      <listener-class>net.beeger.jsftest.TestContextListener</listener-class>
  </listener>

  <servlet>
      <servlet-name>Faces Servlet</servlet-name>
      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
      <load-on-startup> 1 </load-on-startup>
  </servlet>
  <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>

  <taglib>
    <taglib-uri>/html_basic.tld</taglib-uri>
    <taglib-location>/WEB-INF/tld/html_basic.tld</taglib-location>
  </taglib>
</web-app>
```

Der erste Block setzt einen für alle Servlets sichtbaren Parameter `saveStateInClient` auf `false`, was dazu führt, dass Statusdaten der JSF-Implementierung wie zum Beispiel die zuletzt verwendete Oberflächenkomponentenbaumstruktur in der `HttpSession` gesichert werden. Stünde hier `true`, würden diese Informationen mit in der Seite kodiert werden. Dann ist zwar kein Sessionhandling mehr nötig, aber die Seiten können dann unter Umständen, wenn die Baumstruktur umfangreich ist, auch sehr groß werden. Da viele Web-Anwendungen auch selbst die `HttpSession` verwenden, ist hier `false` fast immer die richtige Einstellung.

Der letzte Block bindet die Definition der Tag-Library ein und die beiden Blöcke davor binden jede URL, die an diese Web-Anwendung geschickt wird und `/faces/` enthält an das zentrale Servlet bei den JavaServer Faces – das `FacesServlet`.

Der Block, der bisher ausgelassen wurde, weist den Servlet-Container an, ein Objekt der Klasse `TestContextListener` zu erzeugen und dieses dann als Listener[46] zu verwenden. Diese Klasse sieht wie folgt aus

```
public class TestContextListener implements ServletContextListener
{
  public TestContextListener ()
  {
  }

  public void contextInitialized (ServletContextEvent event)
  {
      ApplicationHandler handler = new TestApplicationHandler();
      LifecycleFactory factory = (LifecycleFactory)
          FactoryFinder.getFactory(FactoryFinder.LIFECYCLE_FACTORY);
      Lifecycle lifecycle =
          factory.getLifecycle(LifecycleFactory.DEFAULT_LIFECYCLE);
      lifecycle.setApplicationHandler(handler);
  }

  public void contextDestroyed (ServletContextEvent event)
  {
  }
}
```

Die erste Operation – `contextInitialized()` – wird aufgerufen nachdem der `ServletContext` initialisiert wurde, was kurz nach dem Start des Servlet-Containers der Fall ist. Die zweite – `contextDestroyed()` – wird gerufen nachdem der `ServletContext` geschlossen wurde und kurz vor dem Herunterfahren des Servlet-Containers.

In dieser Klasse ist das Starten des Servlet-Containers von Interesse, denn das ist der beste Zeitpunkt, um der JSF-Implementierung mitzuteilen, welcher `ApplicationHandler` verwendet werden soll.

[46] Das Listener-Konzept kommt aus der Programmierung graphischer Benutzungsoberflächen in Java und ist eine Umsetzung des Beobachter-Musters (siehe hierzu [Gamma 98])

B Implementationsdetails der Realisierung

Dieses Kapitel enthält Quellcode und Beispiele, die einige Implementationsdetails der Realisierung zeigen.

B.1 Konfigurationsdatei

In diesem Abschnitt wird stückweise ein Beispiel für eine Konfigurationsdatei gezeigt. Jedes Stück der Konfigurationsdatei wird einzeln erläutert. Die hier gezeigten Teile der Konfiguration enthalten zum Zweck der Übersichtlichkeit gekürzte Klassennamen. Eine solche Kürzung ist an zwei aufeinanderfolgenden Punkten zu erkennen. Die konkreten Klassn sind für die Betrachtungen irrelevant. Sie sollen nur verdeutlichen, an welchen Stellen Angaben zu Klassen auftauchen.

```
<webxconfiguration>
  <mandatory signPlacement="before">
    <sign>
      <![CDATA[
        <span style="color:red" title="%%MANDATORY_EXPLANATION%%">* </span>
      ]]>
    </sign>
    <placeHolder>
      <![CDATA[ <span style="color:#ffe708">* </span> ]]>
    </placeHolder>
  </mandatory>
  <invalid signPlacement="after">
    <sign>
      <![CDATA[
        <img src="images/invalid.png" width="16"
             height="16" title="%%INVALID_EXPLANATION%%">
      ]]>
    </sign>
    <placeHolder>
      <![CDATA[
        <img src="images/invalid_placeholder.png" width="16" height="16">
      ]]>
    </placeHolder>
  </invalid>
```

In diesem Teil der Konfiguration wird festgelegt, auf welche Weise signalisiert wird, ob ein Feld verpflichtend ist oder falsch ausgefüllt wurde.

Das `signPlacement`-Attribut gibt an, ob ein entsprechende Markierung vor die betreffende Oberflächenkomponente oder danach platziert wird.

Das `sign`-Tag enthält das HTML-Fragment, dass vor beziehungsweise hinter die Oberflächenkomponente gepackt wird, wenn das Feld verpflichtend ist beziehungsweise falsch ausgefüllt wurde.

Das `placeHolder`-Tag ist für den umgekehrten Fall zuständig. Das hier spezifizierte HTML-Fragment sollte im Browser den gleiche Raum ausfüllen, der auch vom entsprechenden HTML-Fragment der vorigen Kategorie belegt wird. Dadurch wird verhindert, dass zum Beispiel in einer Situation, in der das HTML-Fragment für die verpflichtenden Felder vor der Oberflächenkomponente dargestellt wird und einige Felder verpflichtend sind, während andere dies nicht sind, die Oberflächenkomponenten verschieden eingerückt erscheinen.

`%%MANDATORY_EXPLANATION%%` und `%%INVALID_EXPLANATION%%` sind zwei Platzhalter, die, falls sie in den entsprechenden HTML-Fragmenten auftauchen, zur Laufzeit durch eine Erkärung ersetzt

werden, warum ein Feld verpflichtend ist, beziehungsweise, warum es als falsch ausgefüllt erachtet wird.

```
<javaScriptPrecheck use="true" functionName="precheck" />
```

Hier wird festgelegt, dass die sofortige Prüfung durch JavaScript zum Einsatz kommen soll und dass die JavaScript-Funktion, die für diese Prüfung aufgerufen werden soll, ,,precheck“ heißt. Eine Eingabe von ,,false“, ,,off“ oder ,,no“ beim Attribut `use` führt zur Abschaltung dieser Funktionalität. Der Entwickler der Anwendung ist dafür zuständig, dass die angegebene Funktion zur Verfügung steht.

```
<uiComponent
  type="net.beeger.jwamwebx.jsf...UIDomainValueStringBasedFillIn"
  domainValueType="de.itwps.integrationguard.webadmin.domainvalue.PasswordDV"
  domainValueFactory="de.itwps..webadmin.domainvalue.PasswordDVFactory"
  forEditing="true">
  <property name="columns" value="20" />
  <property name="password" value="true" />
</uiComponent>

<uiComponent
  type="net.beeger.jwamwebx.jsf..uicomponent.UIDomainValueDisplay"
  domainValueType="de.itwps..webadmin.domainvalue.PasswordDV"
  domainValueFactory="de.itwps..webadmin.domainvalue.PasswordDVFactory"
  forEditing="false" />

<uiComponent
  type="de.itwps..webadmin.uicomponent.UIProjectStateDisplay"
  domainValueType="de.itwps..webadmin.domainvalue.ProjectStateDV"
  domainValueFactory="de.itwps..webadmin.domainvalue.ProjectStateDVFactory"
  forEditing="false" />
```

In diesem Teil wird festgelegt, welche Oberflächenkomponenten für welche Fachwerttypen verwendet werden sollen. Zusätzlich wird hier auch zwischen Oberflächenkomponenten, die zum Einfüllen eines Fachwertes – `forEditing="true"` – verwendet werden, und denen, die lediglich Fachwerte anzeigen können – `forEditing="false"` –, unterschieden.

Die Angabe einer Fachwertfabrik – `domainValueFactory` – ist unerlässlich, da nur über die Verwendung der Fabrik Eingaben auf ihre Gültigkeit geprüft und neue Fachwerte erzeugt werden können.

Auch können hier die Eigenschaften einer Oberflächenkomponente – `property`-Tag – angegeben werden.

```
<model name="login"
       type="de.itwps.integrationguard.webadmin.model.LoginData" />
```

Obwohl die Angabe von Modelreferenzen in den JSPs, wie sie von der JSF-Spezifikation propagiert wird, zu einem schlechten Design führt, bei dem zu viel Fachwissen in die Oberfläche programmiert wird, hat die Verwendung solcher Modelreferenzen auch einen Vorteil. Der Vorteil ist, dass der Anwendungsentwickler, wenn er Modelreferenzen verwendet, sich nur dann mit der Baumstruktur befassen muss, wenn er eine neue JSP angeben will, die die Antwort auf eine Anfrage generieren soll. Er bekommt seine Daten in anwendungsnahen Objekten, den Models, geliefert.

Die hier vorgestellte Lösung ermöglicht es, diesen Vorteil auszunutzen, ohne dass die Modellreferenzen in den JSPs angegeben werden müssen.

Der erste dafür wichtige Teil ist die Angabe, welche Modelreferenznamen Namen von Objekten welchen Typs sind. Dies wird in diesem Teil erledigt.

```
  <form name="login">
    <field name="username"
           domainValueType="de.jwam.lang.domainvalue.StringDV"
           modelReference="login.username" mandatory="true" />
    <field name="password"
           domainValueType="de.itwps..webadmin.domainvalue.PasswordDV"
           modelReference="login.password" mandatory="true" />
  </form>
</webxconfiguration>
```

Abschließend kommt hier die Definition eines Formulars. Ein Formular hat einen Namen – `name`-Attribut – und Formularfelder – `field`-Tags.

Ein Formularfeld hat einen Fachwerttyp – `domainValueType`-Attribut –, über den festgelegt wird, was für Fachwerte in dieses Feld gefüllt werden können. Darüber wird dann auch im Endeffekt die Oberflächenkomponente bestimmt, die für ein Formularfeld verwendet wird.

Ein Formularfeld kann verpflichtend sein – `mandatory="true"` und ein Formular kann eine Modelreferenz haben – `modelReference`-Attribut –, über die ein Attribut eines Objektes mit dem Wert des Feldes gefüllt wird.

B.2 Details zum `formfield`-Tag

Eine beispielhafte Verwendung des `formfield`-Tags sieht wie folgt aus

```
    <faces:form id="login" formName="login">
          <webx:formfield id="username" name="username">
    </faces:form>
```

Das `faces:form`-Tag, in dem sich das `formfield`-Tag befindet, ist hier auch zu sehen, weil es ohne dieses nicht alle benötigten Informationen hat.

Die `id` des Tags ist lediglich für den Aufbau der Baumstruktur wichtig. Sie hat im Weiteren keinen Einfluss auf das Verhalten des Tags.

Der `name` gibt an, dass an Stelle dieses Tags eine Oberflächenkomponente eingesetzt werden soll, die für das Formularfeld mit dem Namen ,,username“ des Formulars mit dem Namen ,,login“ in der Konfiguration festgelegt wurde.

Zwei Operationen aus der Klasse `FormFieldTag` sollen hier weitere Klarheit schaffen

```
public int doStartTag () throws JspException
{
  try
  {
    Stack componentStack = findComponentStack();
    String formName = formName(componentStack);
    _domainValueType = WebXConfigurator.instance().domainValueType(
      formName, getName());
```

```
      _modelReference = WebXConfigurator.instance().modelReference(
        formName, getName());
      if (_modelReference != null)
      {
        String modelObjectName = _modelReference;
        int dotIndex = modelObjectName.indexOf(".");
        if(dotIndex >= 0)
        {
          modelObjectName = modelObjectName.substring(0, dotIndex);
        }
        ModelObjectFactory.instance().ensureModelObjectExistence(
          modelObjectName, pageContext.getSession());
      }
    }
    catch (WebXConfigurationException e)
    {
      e.printStackTrace();
    }
    return super.doStartTag();
  }
```

Die Operation `doStart()` wird an einem Objekt einer Tag-Klasse gerufen nachdem das öffnende Tag gelesen und alle Attribute des öffnenden Tags über entsprechende `set`-Operationen an dem Objekt gesetzt wurden.

Zunächst wird der Oberflächenkomponenten-Stack ermittelt, auf dem alle Oberflächenkomponenten liegen, die zu Tags gehören, die das aktuelle Tag – hier also das `formfield`-Tag – beinhalten. Mit diesem wird dann die Operation `formName()` aufgerufen. Diese Operation läuft solange den Stack hinab, bis sie eine Oberflächenkomponente vom Typ `UIForm` gefunden hat, von der sie den Formularnamen holt, denn zu diesem Formular gehört das Formularfeld.

Mit dem Namen des Formularfeldes, der als Attribut am `formfield`-Tag gesetzt wurde, kann das Tag-Objekt die benötigten Informationen aus der Konfiguration ziehen.

Wie im vorigen Abschnitt zu sehen ist, bestehen Model-Referenzen aus dem Namen des Objektes und dem Namen des Attributs, mit dem ein Formularfeld verbunden wird. Um also den Namen des Model-Objektes zu ermitteln wird von der Model-Referenz alles nach dem Punkt entfernt. Die `ModelObjectFactory` wurde beim Lesen und Analysieren der Konfigurationsdatei mit allen Informationen über die im System verwendeten Model-Objekte versorgt und sorgt bei einem Aufruf der Operation `ensureModelObjectExistence()` dafür, dass ein solches Model-Objekt in der Session existiert. Das ist notwendig, da das Setzen der Attribute eines Model-Objektes mit den Werten in den Oberflächenkomponenten und vice versa von der JavaServer Faces Implementierung übernommen wird. Diese erwartet jedoch, dass ein entsprechendes Objekt bereits vorhanden ist.

Hiernach wird die Kontrolle an die Oberklasse – `FacesTag` – übergeben, die die zu dem Tag zugehörige Oberflächenkomponente über das `id`-Attribut herausfindet und einige JSF-spezifische Dinge erledigt. Wird bei der Suche keine Oberflächenkomponente gefunden – Das ist der Fall, wenn die Seite zum ersten Mal aufgerufen wird –, dann wird die Operation `createComponent()` aufgerufen, die eine neue passende Oberflächenkomponente erzeugt. Diese Operation ist wiederrum im `FormFieldTag` implementiert

```
  public UIComponent createComponent ()
  {
```

```
    UIDomainValueComponent uiDomainValueComponent =
      UIComponentFactory.instance().uiComponent(_domainValueType, true);
    if(uiDomainValueComponent instanceof UIDomainValueFillIn)
    {
      ((UIDomainValueFillIn) uiDomainValueComponent).setMandatory(_mandatory);
    }
    return uiDomainValueComponent;
  }
```

Mit den in `doStartTag()` gesammelten Informationen kann hier bei der `UIComponentFactory` eine für das Formularfeld passende Oberflächenkomponente erfragt werden. Die Oberflächenkomponenten für Fachwerte teilen sich in zwei Gruppen auf. Zu einer dieser Gruppen zählen Oberflächenkomponenten, die Fachwerte nur anzeigen können. In die andere Gruppe fallen all jene Oberflächenkomponenten, in die man darüber hinaus auch Fachwerte eingeben kann. Wenn ein Vertreter der zweiten Sorte erzeugt wird, muss an diesem auch gesetzt werden, ob er für ein verpflichtendes Formularfeld verwendet wird.

B.3 Details der Implementation der `UIDomainValueStringBasedFillIn`

Dieser Abschnitt zeigt am Beispiel von `UIDomainValueStringBasedFillIn` wie die Generierung der HTML-Repräsentation und die Verarbeitung der Anfrageparameter in einer Fachwert-Oberflächenkomponente erfolgt.

Die Operation, die zur Erzeugung der Darstellung verwendet wird, heißt `encodeEnd()`.

```
  public void encodeEnd (FacesContext facesContext) throws IOException
  {
    encodePrologue(facesContext);
    String value = getText();
    if(value == null && hasDomainValue())
    {
      value = getDomainValue().toString();
    }
    ResponseWriter responseWriter = facesContext.getResponseWriter();
    responseWriter.write("<input type=\"");
    if (hasProperty("password")
        && ((Boolean) getProperty("password")).booleanValue())
    {
      responseWriter.write("password");
    }
    else
    {
      responseWriter.write("text");
    }
    if(hasProperty("columns"))
    {
      responseWriter.write("\" size=\"" + getProperty("columns"));
    }
    responseWriter.write("\" name=\"");
    responseWriter.write(getCompoundId());
```

```
    responseWriter.write("\" value=\"");
    if (value != null)
    {
      responseWriter.write(value);
    }
    if(WebXConfigurator.instance().isUseJavaScriptPrecheck()
       && getDomainValueFactory() instanceof RegExpDomainValueFactory)
    {
      responseWriter.write("\" onkeyup=\"");
      responseWriter.write(
        WebXConfigurator.instance().getJavaScriptFunctionName());
      responseWriter.write("('");
      responseWriter.write(((RegExpDomainValueFactory)
        getDomainValueFactory()).validatingRegExp());
      responseWriter.write("', this.value, this)");
    }
    responseWriter.write("\">");
    encodeEpilogue(facesContext);
  }
```

Zuerst wird in die Antwort auf die Anfrage – hier die HTML-Seite, die schließlich angezeigt wird – der Vorspann der Oberflächenkomponente geschrieben. Dazu wird die Operation `encodePrologue()` aufgerufen. Ebenso wird auch am Ende von `encodeEnd()` durch einen Aufruf von `encodeEpilogue()` der Nachspann einer Oberflächenkomponente geschrieben. Im Vorspann oder Nachspann einer Oberflächenkomponente – also vor ihr oder nach ihr – können die Markierungen erscheinen, die angeben, ob ein Feld verpflichtend ist oder ob es falsch ausgefüllt wurde. Wenn ein Feld verpflichtend ist, dann muss vor oder hinter der Oberflächenkomponente die entsprechende Markierung erscheinen. Ist das Feld nicht verpflichtend oder fehlerhaft ausgefüllt, dann muss an derselben Stelle der Platzhalter erscheinen. Nun bleibt noch zu klären, ob das Zeichen vor oder hinter der Oberflächenkomponente erscheinen soll. Dies wird, wie in Abschnitt B.1 gezeigt, in der Konfigurationsdatei definiert.

In den meisten Fällen wird ein Aufruf von `encodePrologue()` und `encodeEpilogue()` die Behandlung dieser Fälle erledigen und eine passende Darstellung erzeugen. Nur bei komplexeren – zum Beispiel mehrzeiligen – Oberflächenkomponenten kann eine eigene Implementierung dieser Funktionalität nötig werden.

Nach dem Schreiben des Vorspanns und vor dem Schreiben des Nachspanns wird der HTML-Code für die Präsentation der Oberflächenkomponente generiert, wobei die Eigenschaften, die in der Konfiguration gesetzt wurden, verwendet werden. In diesem Beispiel wird ein HTML-Eingabefeld verwendet. Der Inhalt des Eingabfeldes richtet sich danach, ob die Anwendung einen Fachwert gesetzt hat, die Oberflächenkomponente bei der letzten Anfrage fehlerhaft befüllt wurde oder ob die Oberflächenkomponente keinen Inhalt hat.

Am Ende der Operation wird auch der Aufruf der JavaScript-Funktion, die zur Vor-Prüfung verwendet wird, in das HTML-Tag geschrieben. Dazu wird der JavaScript-EventHandler `onkeyup` verwendet, der nach jeder Eingabe eines Zeichens aufgerufen wird.

B.4 Standardimplementation der Vor-Prüfung mit JavaScript

Dieser Abschnitt stellt die Standardimplementation der JavaScript-Funktion für die Vor-Validierung vor.

```
function precheck(regExp, value, inputs)
{
  var match = value.match(regExp);
  var valid = (value.length == 0
               || (match && match.length == 1
                   && match[0].length == value.length));

  if(new String(inputs.length).indexOf("undefined") >=0 )
  {
    adaptClass(valid, inputs)
  }
  else
  {
    for(i = 0; i <= inputs.length; i++)
    {
      adaptClass(valid, inputs[i])
    }
  }
}
```

In den ersten beiden Zeilen der Funktion erfolgt die Validierung (siehe Abschnitt 3.3.2). Danach wird zunächst festgestellt, ob sich in `inputs` ein Array von HTML-Elementen oder ein einziges HTML-Element befindet. Hierzu wird das Ergebnis des Ausdrucks `inputs.length` in eine Zeichenkette konvertiert. Enthält diese nun das Wort ,,undefined“, bedeutet das, dass eine Funktion oder ein Attribut mit dem Namen ,,length“ an der Variablen `inputs` nicht existiert. Daraus folgt dann, dass `inputs` kein Array enthält sondern ein einzelnes HTML-Element[47]. Nun wird für jedes Element des Arrays oder den Parameter selbst, wenn er kein Array ist, die Funktion `adaptClass` aufgerufen.

```
function adaptClass(valid, input)
{
  if(valid)
  {
    input.className="validInput";
  }
  else
  {
    input.className="invalidInput";
  }
}
```

Hier wird, je nachdem ob die Eingabe valide ist oder nicht, die CSS-Klasse des HTML-Elements angepasst. Die hier verwendeten CSS-Klassen sind ,,validInput“ und ,,invalidInput“. Diese Klassen werden typischerweise in einer zentralen CSS-Datei definiert, die auch die Definitionen der anderen

[47] C++ und Java-Entwickler werden sich nun vielleicht wundern, warum eine so unelegante Vorgehensweise gewählt wurde und nicht eine Funktion für den einen Fall und eine andere überladene Funktion für den anderen Fall implementiert wurde. Da JavaScript keine Typisierung der Parameter kennt, ist eine Überladung mit gleicher Parameteranzahl und verschiedenen Typen nicht möglich. Es ist jedoch auch nicht möglich mehrere Funktionen mit dem gleichen Namen anzubieten, die eine unterschiedliche Anzahl von Parametern annehmen. Es wird immer die erste Funktion genommen, die den passenden Namen hat.

in der Web-Anwendung verwendeten CSS-Klassen[48] enthält. Im folgenden wird ein Beispiel für eine Definition dieser beiden Klassen angegeben

```
.validInput {
  color:black;
}

.invalidInput {
  color:red;
}
```

In dieser Variante wird die Farbe des Textes der Eingabe in rot verändert, wenn diese fehlerhaft ist. Ansonsten ist die Farbe des Eingabe schwarz.

[48] Zur Erinnerung: CSS = Cascading Style Sheets; ein w3c-Standard (siehe auch [w3c-website]) zur Beschreibung des Layouts von HTML-Elementen

C Bibliographie

[3Amigos 99] Grady Booch, Jim Rumbaugh, Ivar Jacobsen: *Das UML-Benutzerhandbuch*. Bonn: Addison Wesley Longman Inc., 1999

[Beier 03] Betsy Beier, Misha W. Vaughan: *The Bull's-Eye: A Framework for Web Application User Interface Design Guidlines* in *Proceedings of the conference on Human factors in computing systems*. Ft. Lauderdale ACM, April 2003

[Bleek 99] Wolf-Gideon Bleek, Thorsten Görtz, Carola Lilienthal, Martin Lippert, Stefan Roock, Wolfgang Strunk, Ulfert Weiss, Henning Wolf: *Interaktionsformen zur flexiblen Anbindung von Fenstersystemen*. Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 1999

[Bohlmann 00] Holger Bohlmann: *Diplomarbeit: Thin Clients in JWAM*. Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, November 2000

[Burbeck 92] Steve Burbeck: *Applications Programming in Smalltalk-80: How to use Model-View-Controller (MVC)* . http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html: 1992

[Eckstein 98] Robert Eckstein, Marc Loy, Dave Wood: *Java Swing*. Sebastopol: O'Reilly, 1998

[FormProc-website] FormProc. http://formproc.sourceforge.net/

[Gamma 98] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Entwurfsmuster*. Bonn: Addison Wesley Longman Inc., 1998

[Geary 01] David M. Geary: *Advanced JavaServer Pages*. Upper Saddle River: Prentice Hall PTR, 2001

[Girgensohn 95] Andreas Girgensohn et al.: *Dynamic Forms: An Enhanced Interaction Abstraction Based on Forms* in *Human Computer Interaction, Interact ´95*. London: Chapman & Hall, 1995

[Hall 00] Marty Hall: *Core Servlets and JavaServer Pages*. Upper Saddle River: Prentice Hall PTR, 2000

[Hassan 02] Ahmed E Hassan, Richard C. Holt: *Architecture Recovery of Web Applications* in *Proceedings of the 24th Conference on Software Engineering.*. New York: ACM Press, Mai 2002

[jcp-website] Java Community Process. http://www.jcp.org/

[JWAM-website] JWAM. http://www.jwam.de/

[Lippert 99] Martin Lippert: *Diplomarbeit: Die Desktop-Metapher in Systemen nach dem Werkzeug-und-Material-Ansatz*. Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, Oktober 1999

[McClanahan 02] Craig R. McClanahan: *JavaServer Faces Specification - Version 1.0, Early Access Draft*. Santa Clara : Sun Microsystems Inc., September 2002

[Müller 99] Klaus Müller: *Studienarbeit: Konzeption und Umsetzung eines Fachwertkonzeptes*. Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, Juli 1999

[Münz 01] Stefan Münz: *SelfHTML*. http://selfhtml.teamone.de/

[Nielsen 99] Jakob Nielsen: *Designing Web Usability : The Practice of Simplicity*. Indianapolis : New Riders Publishing, 1999

[Nielsen 02] Jakob Nielsen, Marie Tahir: *Homepage Usability – 50 Websites Desconstructed*. Indianapolis : New Riders Publishing, 2002

[Otto 00] Michael Otto, Norbert Schuler: *Diplomarbeit: Fachliche Services – Geschäftslogik als Dienstleistung für verschiedene Benutzungsschnittstellen-Typen*. Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, November 2000

[Raskin 00] Jef Raskin: *The Humane Interface*. New York: Addison Wesley ACM Press, 2000

[Sturm 98] Thorsten Sturm: *Studienarbeit: Interaktionsformen für objektorientierte, reaktive Softwaresysteme unter Berücksichtigung der Werkzeug-Material-Metapher*. Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, Dezember 1998

[Stevens 94] W. Richard Stevens: *TCP/IP Illustrated Volume 1*. New York: Addison Wesley, 1994

[Struts-website] Apache Jakarta Project: Struts. http://jakarta.apache.org/struts/

[Szyperski 98] Clemens Szyperski: *Component Software – Beyond Object-Oriented Programming*. New York: Addison Wesley, 1998

[Thiel 02] Olaf Thiel: *Diplomarbeit: Konzeption und Entwicklung eines Formularwesens für das JWAM Rahmenwerk*. Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 14.03.2002

[ulc-website] ULC. http://www.canoo.com/ulc/index.html

[wingS-website] wingS. http://wings.mercatis.de/

[w3c-website] World Wide Web Consortium. http://www.w3.org/

[Wahrig 00] Gerhard Wahrig: *Deutsches Wörterbuch*. Gütersloh/München: Bertelsmann Lexikon Verlag, 2000

[Züllighoven 98] Züllighoven et al.: *Das objektorientierte Konstruktionshandbuch*. Heidelberg: dpunkt Verlag, 1998

C.1 Quellen für die Epigraphen

Orson Scott Card: *Children of the Mind*. New York: Tor Books, 1997

Doris Day *Whatever Will Be, Will Be*.
http://www.elyrics4u.com/w/whatever_will_be_will_be_doris_day.htm

Betrand Meyer *Object-Oriented Software Construction – Second Edition*. Upper Saddle River: Prentice Hall PTR, 1997

Neal Stephenson: *Snow Crash*. New York: Bantam Books, 1992

Neal Stephenson: *In the beginning was ... the command line*. New York: Avon Books, 1999

Larry Wall et al.: *Programming Perl*. Sebastopol: O'Reilly, 2000