Kapitel **3**

Entwurfsbetrachtungen zur Präsentationsschicht

- Entwurf der Präsentationsschicht
- Schlechte Praktiken für die Präsentationsschicht

Wenn Entwickler die Präsentationsmuster anwenden, die im Katalog dieses Buchs erscheinen, müssen sie auch die damit in Verbindung stehenden Entwurfsprobleme berücksichtigen. Diese Probleme beziehen sich auf den Entwurf mit Mustern auf verschiedenen Ebenen und können zahlreiche Aspekte eines Systems beeinflussen, einschließlich Sicherheit, Datenintegrität, Verwaltbarkeit und Skalierbarkeit. Auf diese Punkte gehen wir in diesem Kapitel ein.

Obwohl sich viele dieser Entwurfsthemen in ein Musterformat fassen ließen, haben wir uns für eine andere Darstellung entschieden, weil sich diese Fragen eher auf niedrigere Abstraktionsebenen beziehen als auf die Präsentationsmuster im Katalog. Statt die Themen als Muster zu dokumentieren, haben wir eine eher zwanglose Form gewählt: Wir beschreiben sie einfach als Themen, die man berücksichtigen muss, wenn man auf dem Musterkatalog basierende Systeme implementiert.

3.1 Entwurf der Präsentationsschicht

3.1.1 Sitzungsverwaltung

Der Begriff *Benutzersitzung* beschreibt eine Konversation, die sich über viele Anforderungen zwischen einem Client und einem Server erstreckt. Auf das Konzept der Benutzersitzung stützen wir uns in den folgenden Abschnitten.

Sitzungszustand auf Client

Um den Sitzungszustand auf dem Client zu sichern, ist es erforderlich, den Sitzungszustand zu serialisieren und in den Code der HTML-Seite einzubetten, die an den Client zurückgegeben wird.

Das persistente Speichern des Sitzungszustands auf der Clientseite

- ist relativ leicht zu implementieren und
- genügt den Anforderungen, wenn man einen nicht zu umfassenden Zustand sichern muss.

Darüber hinaus vermeidet diese Strategie scheinbar das Problem, den Zustand über Server hinweg zu replizieren, wenn eine Lastverteilung mit mehreren Computern implementiert ist.

Für die Sicherung des Sitzungszustands auf dem Client sind zwei Strategien gebräuchlich – versteckte HTML-Felder und HTTP-Cookies. Darauf gehen wir nachfolgend ein. Bei einer dritten Strategie bettet man den Sitzungszustand direkt in die URLs ein, auf die in jeder Seite verwiesen wird (z.B. `<form action=someServlet?var1=x&var2=y method=GET>`). Auch wenn diese dritte Strategie weniger gebräuchlich ist, hat sie viele Beschränkungen mit den beiden folgenden Methoden gemein.

Versteckte HTML-Felder

Obwohl es relativ leicht ist, diese Strategie zu implementieren, gibt es einige Nachteile, wenn man den Sitzungszustand auf dem Client in versteckten HTML-Feldern speichert. Diese Nachteile treten besonders dann zutage, wenn ein umfangreicher Zustand zu sichern ist, weil sich dann ein negativer Einfluss auf den Durchsatz bemerkbar macht. Da jetzt jeder HTML-Code den Zustand einbettet oder enthält, muss er bei jeder Anforderung und jeder Antwort über das Netzwerk übertragen werden.

Bei versteckten Feldern zum Speichern des Sitzungszustands ist der dauerhaft zu speichernde Zustand außerdem auf Stringwerte beschränkt, sodass jeder Objektverweis in eine Stringdarstellung zu überführen ist. Außerdem liegt der Zustand in der generierten HTML-Quelle als reiner Text offen, sofern die Daten nicht speziell verschlüsselt sind.

HTML-Cookies

Ähnlich der Strategie mit versteckten Feldern ist es relativ einfach, die Strategie mit HTTP-Cookies zu implementieren. Diese Strategie hat leider fast die gleichen Nachteile. Insbesondere leidet der Durchsatz beim Speichern umfangreicher Zustandsinformationen, weil der gesamte Sitzungszustand bei jeder Anforderung und jeder Antwort über das Netzwerk zu übertragen ist.

Außerdem sehen wir uns Größen- und Typbeschränkungen gegenüber, wenn wir den Sitzungszustand auf dem Client speichern. Es gibt Beschränkungen hinsichtlich der Header-Größe von Cookies und das schränkt wiederum den Umfang der dauerhaft zu speichernden Daten ein. Darüber hinaus ist der mit HTTP-Cookies zu speichernde Zustand analog zu versteckten HTML-Feldern auf Werte im Stringformat beschränkt.

Sicherheitsbelange des clientseitigen Sitzungszustands

Wenn man den Sitzungszustand auf dem Client speichert, sind auch Sicherheitsprobleme zu berücksichtigen. Wenn Sie nicht möchten, dass Ihre Daten für den Client offen gelegt werden, müssen Sie sie mit bestimmten Verschlüsselungsverfahren sichern.

Auch wenn das Speichern des Sitzungszustands auf dem Client anfänglich relativ einfach zu implementieren ist, weist das Verfahren zahlreiche Klippen auf, die Zeit und Überlegung kosten, um sie zu umschiffen. Bei Projekten, die mit einem großen Datenvolumen umgehen, wie es bei Unternehmenssystemen in der Regel der Fall ist, übersteigen die Nachteile bei weitem den erzielbaren Nutzen.

Sitzungszustand in der Präsentationsschicht

Verwaltet der Server den Sitzungszustand, wird er mithilfe einer Sitzungs-ID abgerufen und bleibt normalerweise verfügbar, bis eines der folgenden Ereignisse auftritt:

- Eine für die Sitzung vordefinierte Zeitüberschreitung wird überschritten.
- Die Sitzung wird manuell beendet.
- Der Zustand wird aus der Sitzung gelöscht.

Beachten Sie, dass nach einem Herunterfahren des Servers bestimmte Mechanismen der Sitzungsverwaltung im Hauptspeicher möglicherweise nicht wiederherstellbar sind.

Wenn Anwendungen einen umfangreichen Sitzungszustand benötigen, sollte man ihn vorzugsweise auf dem Server speichern. Hier ist man nicht durch die Größe oder die Typbeschränkungen der clientseitigen Sitzungsverwaltung eingeschränkt. Außerdem umgeht man die Sicherheitsprobleme, die mit der Offenlegung des Sitzungszustands beim Client verbunden sind, und man hat keine Leistungseinbußen durch die Übertragung des Sitzungszustands über das Netzwerk bei jeder Anforderung zu verzeichnen.

Außerdem profitiert man von der Flexibilität, die diese Strategie bietet. Durch dauerhaftes Speichern des Sitzungszustands auf dem Server kann man flexibel zwischen Einfachheit und Komplexität abwägen und sich mit Skalierbarkeit und Performanz befassen.

Speichert man den Sitzungszustand auf dem Server, muss man entscheiden, wie man diesen Zustand für jeden Server verfügbar macht, von dem aus man die Anwendung ausführt. Diese Fragen betreffen unter anderem die Replikation des Sitzungszustands zwischen gruppierten Softwareinstanzen auf mehrere Computer mit Lastverteilung – ein mehrdimensionales Problem. Allerdings bieten zahlreiche moderne Applikationsserver eine Vielzahl von vorgefertigten Lösungen. Es sind Lösungen verfügbar, die oberhalb der Applikationsserver-Ebene ansetzen. Eine solche Lösung ist die Verwaltung so genannter »klebriger Benutzer« (Sticky User Experience). Beispielsweise bietet die Firma Resonate [Resonate] ein Traffic-Management-System an, um Benutzer zum selben Server weiterzuleiten, der so alle Anforderungen in deren Sitzung behandelt. Man bezeichnet dieses Vorgehen auch als *Serveraffinität*.

Eine andere Alternative ist es, den Sitzungszustand entweder in der Geschäftsschicht oder in der Ressourcenschicht zu speichern. Mit JavaBean-Komponenten kann man den Sitzungszustand in der Geschäftsschicht halten und in der Ressourcenschicht lässt sich eine relationale Datenbank verwenden. Weitere Informationen zur Geschäftsschichtoption finden Sie im Abschnitt »Session Beans verwenden« in Kapitel 4.

3.1.2 Den Clientzugriff steuern

Es gibt verschiedene Gründe, den Clientzugriff auf bestimmte Anwendungsressourcen einzuschränken oder zu steuern. In diesem Abschnitt untersuchen wir zwei dieser Szenarios.

Ein Grund liegt vor, wenn man eine Ansicht oder Teile einer Ansicht gegenüber dem direkten Zugriff durch einen Client schützen muss. Das kann zum Beispiel notwendig sein, wenn nur registrierte oder angemeldete Benutzer auf eine bestimmte Ansicht zugreifen dürfen oder wenn der Zugriff auf Teile einer Ansicht nur für Benutzer verfügbar sein soll, die zu einer bestimmten Rolle gehören.

Nach der Beschreibung dieser Themen behandeln wir ein zweites Szenario, das sich auf die Steuerung der Navigation eines Benutzers durch die Anwendung bezieht. Dieses zweite Szenario betont Fragen, die sich auf mehrfaches Versenden eines Formulars beziehen, da hierdurch unerwünschte Mehrfachtransaktionen auftreten können.

Eine Ansicht schützen

In bestimmten Fällen wird eine Ressource in ihrer Gesamtheit gegenüber dem Zugriff durch bestimmte Benutzer eingeschränkt. Dieses Ziel lässt sich mit verschiedenen Strategien erreichen. Beispielsweise kann man mit Anwendungslogik, die bei Verarbeitung des Controllers oder der Ansicht ausgeführt wird, den Zugriff verweigern. Eine zweite Strategie besteht darin, das Laufzeitsystem zu konfigurieren, um den Zugriff auf bestimmte Ressourcen nur über einen internen Aufruf von einer anderen Anwendungsressource zu erlauben. In diesem Fall ist der Zugriff auf diese Ressourcen über eine andere Anwendungsressource der Präsentationsschicht umzuleiten, beispiels-

weise einen Servlet-Controller. Der Zugriff auf diese eingeschränkten Ressourcen ist nicht über einen direkten Browseraufruf verfügbar.

In der Regel setzt man dazu einen Controller als Delegationspunkt für diesen Typ der Zugriffssteuerung ein. Es ist auch üblich, den Schutz direkt in eine Ansicht einzubetten. Den Controller-basierten Ressourcenschutz behandeln wir im Abschnitt »Refaktorisierungen der Präsentationsschicht« in Kapitel 5 und im Musterkatalog, sodass wir uns hier auf Ansicht-basierte Steuerungsstrategien konzentrieren. Wir beschreiben diese Strategien zuerst, bevor wir uns der alternativen Strategie, den Zugriff über Konfiguration zu steuern, zuwenden.

Schutz in eine Ansicht einbetten

Um den Schutz in die Verarbeitungslogik einer Ansicht einzubetten sind zwei Varianten gebräuchlich. Eine Variante blockiert den Zugriff für die gesamte Ressource, während die andere nur den Zugriff auf einen Teil dieser Ressource sperrt.

Einen Alles-oder-Nichts-Schutz pro Ansicht einbinden

In bestimmten Fällen erlaubt oder verweigert der in den Verarbeitungscode der Ansicht eingebettete Code den Zugriff nach dem Alles-oder-Nichts-Prinzip. Mit anderen Worten hindert diese Logik einen Benutzer am Zugriff auf eine bestimmte Ansicht in ihrer Gesamtheit. Normalerweise wird ein derartiger Schutz vorzugsweise in einen zentralisierten Controller eingebettet, sodass die Logik nicht über den gesamten Code verteilt ist. Diese Strategie ist sinnvoll, wenn nur ein kleiner Teil von Seiten zu schützen ist. Normalerweise tritt dieses Szenario auf, wenn sich ein technisch nicht versierter Benutzer zyklisch durch eine kleine Anzahl von statischen Seiten auf einer Site bewegt. Wenn der Client trotzdem noch an dieser Site angemeldet sein muss, um diese Seiten zu betrachten, dann fügt man wie in Beispiel 3.1 gezeigt eine benutzerdefinierte Hilfsmarkierung am Beginn jeder Seite ein, um den Zugriffstest zu vervollständigen.

Beispiel 3.1: Einbinden eines Alles-oder-Nichts-Schutzes pro Ansicht

```
<%@ taglib uri="/WEB-INF/corej2eetaglibrary.tld"
  prefix="corePatterns" %>

<corePatterns:guard/>
<HTML>
.
.
.
</HTML>
```

Einen Schutz für einen Teil einer Ansicht einbinden

In anderen Fällen verweigert einfach die in den Verarbeitungscode der Ansicht eingebettete Logik den Zugriff auf Teile einer Ansicht. Diese zweite Strategie lässt sich mit der vorher erwähnten Alles-oder-Nichts-Strategie verbinden. Eine Analogie soll das verdeutlichen: Wir nehmen einen Raum in einem Gebäude an, zu dem wir den Zugang steuern. Der Alles-oder-Nichts-Schutz teilt den

Besuchern mit, ob sie durch den Raum gehen können oder nicht, während die zweite Schutzlogik den Besuchern sagt, was sie sich ansehen dürfen, nachdem sie sich erst einmal im Raum befinden. Die folgenden Beispiele zeigen, warum man diese Strategie einsetzt.

Teile einer Ansicht basierend auf Benutzerrolle nicht anzeigen
Ein Teil einer Ansicht soll möglicherweise abhängig von einer Benutzerrolle nicht angezeigt werden. Zum Beispiel hat ein Manager beim Betrachten seiner Firmeninformationen Zugriff auf eine Unteransicht, die sich mit einem Verwaltungsüberblick für seine Mitarbeiter beschäftigt. Ein Mitarbeiter kann nur seine eigenen Firmeninformationen sehen und er ist von den Teilen der Benutzeroberfläche ausgeschlossen, die den Zugriff auf alle Überblicksinformationen bieten (siehe Beispiel 3.2).

Beispiel 3.2: Teile der Ansicht basierend auf Benutzerrolle nicht anzeigen

```
<%@ taglib uri="/WEB-INF/corej2eetaglibrary.tld"
  prefix="corePatterns" %>

<HTML>
.
.
.
<corePatterns:guard role="manager">
<b>Nur für Manager zugänglich!</b>
<corePatterns:guard/>
.
.
.
</HTML>
```

Teile einer Ansicht basierend auf Systemzustand oder Fehlerbedingungen nicht anzeigen
Das Layout der Anzeige lässt sich je nach Systemumgebung modifizieren. Wenn zum Beispiel eine Benutzeroberfläche dazu dient, Prozessoren zu verwalten, und der Benutzer ein Gerät mit nur einem Prozessor wählt, kann man die Teile der Anzeige unterdrücken, die sich ausschließlich auf Mehrprozessorsysteme beziehen.

Schutz durch Konfiguration

Um den Client vom direkten Zugriff auf bestimmte Ansichten auszuschließen, kann man das Präsentationsmodul so konfigurieren, dass es Zugriff auf diese Ressourcen nur über andere interne Ressourcen erlaubt, beispielsweise einen Servlet-Controller, der einen RequestDispatcher (Anforderungsverteiler) verwendet. Zusätzlich können Sie die Sicherheitsmechanismen nutzen, die in den Webcontainer integriert sind. Diese basieren auf der Servlet-Spezifikation ab Version 2.2. Sicherheitseinschränkungen sind im Deployment-Deskriptor namens `web.xml` definiert.

Die ebenfalls in der Servlet-Spezifikation beschriebenen *Basic*- und *Formular-basierten* Authentifizierungsverfahren stützen sich auf diese Sicherheitsinformationen. Statt die Spezifikation an dieser Stelle zu wiederholen, verweisen wir hinsichtlich der Details auf die aktuelle Spezifikation, die unter *http://java.sun.com/products/servlet/index.html* auch zum Download verfügbar ist.

Damit Sie wissen, was Sie erwartet, wenn Sie deklarative Sicherheitseinschränkungen in Ihre Umgebung aufnehmen, behandeln wir kurz dieses Thema und wie sich dieses Verfahren zum Alles-oder-Nichts-Schutz durch Konfiguration verhält. Schließlich beschreiben wir eine einfache und generische Alternative für einen Alles-oder-Nichts-Schutz einer Ressource.

Ressourcenschutz über Standardsicherheitseinschränkungen
Anwendungen lassen sich mit einer Sicherheitseinschränkung konfigurieren. Mit dieser deklarativen Sicherheit kann man per Programm den Zugriff basierend auf Benutzerrollen steuern. Ressourcen kann man für bestimmte Benutzerrollen verfügbar machen und anderen Benutzern verweigern. Wie der Abschnitt »Schutz in eine Ansicht einbetten« weiter vorn in diesem Kapitel erläutert hat, kann man darüber hinaus auch Teile einer Ansicht basierend auf diesen Benutzerrollen einschränken. Wenn es bestimmte Ressourcen gibt, die in ihrer Gesamtheit für alle direkten Browseranforderungen gesperrt werden sollen, wie es bei dem im vorherigen Abschnitt beschriebenen Alles-oder-Nichts-Szenario der Fall ist, dann kann man die betreffenden Ressourcen auf eine Sicherheitsrolle einschränken, die keinem Benutzer zugeordnet ist. Die auf diese Weise konfigurierten Ressourcen bleiben für alle direkten Browseranforderungen unzugänglich, solange man die Sicherheitsrolle nicht zuweist. Die auszugsweise in Beispiel 3.3 angegebene `web.xml`-Konfigurationsdatei definiert eine Sicherheitsrolle, um den direkten Browserzugriff zu unterbinden.

Der Rollenname lautet »sensitive« (vertraulich) und die eingeschränkten Ressourcen sind mit `sensitive1.jsp`, `sensitive2.jsp` und `sensitive3.jsp` benannt. Solange kein Benutzer oder keine Gruppe der Rolle »sensitive« zugeordnet wird, sind Clients nicht in der Lage, direkt auf diese JSPs zuzugreifen. Da diese Sicherheitseinschränkungen für intern verteilte Anforderungen unwirksam sind, gilt gleichzeitig, dass eine Anforderung, die anfänglich durch einen Servlet-Controller behandelt und dann zu einer dieser drei Ressourcen weitergeleitet wird, den Zugriff auf diese JSPs erhält.

Schließlich sei auf einige Ungereimtheiten hingewiesen, die bei der Implementierung dieses Aspekts der Servlet-Spezifikation Version 2.2 quer durch die einzelnen Anbieterprodukte zu verzeichnen sind. Server, die Servlet 2.3 unterstützen, sollten in dieser Hinsicht konsistent sein.

Beispiel 3.3: Eine nicht zugewiesene Sicherheitsrolle bietet Alles-oder-Nichts-Steuerung

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>SensitiveResources
        </web-resource-name>
        <description>Eine Sammlung vertraulicher Ressourcen
        </description>
        <url-pattern>/trade/jsp/internalaccess/
            sensitive1.jsp</url-pattern>
        <url-pattern>/trade/jsp/internalaccess/
```

```
            sensitive2.jsp</url-pattern>
        <url-pattern>/trade/jsp/internalaccess/
            sensitive3.jsp</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>sensitive</role-name>
    </auth-constraint>
</security-constraint>
```

Ressourcenschutz über einfache und generische Konfiguration
Es gibt eine einfache und generische Möglichkeit, einen Client vom direkten Zugriff auf eine Ressource, beispielsweise eine JSP, fern zu halten. Diese Methode erfordert keine Änderungen an der Konfigurationsdatei, wie sie etwa in Beispiel 3.3 zu sehen sind, sondern setzt einfach die Ressource unter das Verzeichnis `/WEB-INF/` der Webanwendung. Um beispielsweise den direkten Browserzugriff auf eine Ansicht namens `info.jsp` in der Webanwendung `securityissues` zu blockieren, können wir die JSP-Quelldatei im folgenden Verzeichnis platzieren:

```
/securityissues/WEB-INF/internalaccessonly/info.jsp
```

Auf das Verzeichnis `/WEB-INF/`, seine Unterverzeichnisse und folglich auch auf `info.jsp` ist kein direkter öffentlicher Zugriff erlaubt. Andererseits kann ein Controller-Servlet trotzdem noch bei Bedarf zu dieser Ressource weiterleiten. Das ist eine Alles-oder-Nichts-Form der Kontrolle, da ein direkter Browserzugriff auf derartig konfigurierte Ressourcen in ihrer Gesamtheit nicht möglich ist.

Ein diesbezügliches Beispiel finden Sie im Abschnitt »Ressourcen vor einem Client verbergen « in Kapitel 5.

Mehrfaches Absenden von Formularen

Benutzer können in ihrer Browser-Clientumgebung auf die Schaltfläche ZURÜCK klicken und unbeabsichtigt dasselbe Formular, das sie bereits vorher abgesendet haben, noch einmal abschicken und dadurch eine zweite Transaktion auslösen. Analog dazu kann ein Benutzer bevor eine Bestätigungsseite eingetroffen ist, auf die Schaltfläche STOP des Browsers klicken und daraufhin dasselbe Formular noch einmal schicken. In den meisten Fällen wollen wir dieses doppelte Versenden abfangen und deaktivieren. Mit einem steuernden Servlet lässt sich eine Art Kontrollstelle einrichten, um dieses Problem anzugehen.

Synchronisations- oder Déjà vu-Token
Diese Strategie widmet sich dem Problem der doppelten Formularversendung. Ein Synchronisationstoken wird in einer Benutzersitzung gesetzt und in jedes an den Client zurückgegebene Formular eingebunden. Wenn der Benutzer dieses Formular abschickt, wird das Synchronisationstoken im Formular mit dem Synchronisationstoken in der Sitzung verglichen. Beim erstmaligen Versenden des Formulars sollten die Token übereinstimmen. Wenn die Token voneinander abweichen, kann man die Formularversendung deaktivieren und eine Fehlermeldung an den Benutzer zurückgeben.

Ungleiche Token können auftreten, wenn der Benutzer ein Formular abschickt, dann auf die Browser-Schaltfläche ZURÜCK klickt und das Formular erneut abzuschicken versucht.

Wenn jedoch zwei Tokenwerte übereinstimmen, kann man darauf vertrauen, dass der Steuerungsfluss genau wie erwartet verlaufen ist. Zu diesem Zeitpunkt setzt man den Tokenwert in der Sitzung auf einen neuen Wert und akzeptiert das gesendete Formular.

Mit dieser Strategie kann man auch den direkten Browserzugriff auf bestimmte Seiten steuern, wie es die Abschnitte zum Ressourcenschutz beschrieben haben. Nehmen wir beispielsweise an, ein Benutzer setzt ein Lesezeichen für Seite A einer Anwendung, wobei die Seite A nur von den Seiten B und C aus zugänglich sein soll. Wählt dann der Benutzer die Seite A über das Lesezeichen an, wird die Seite außerhalb der vorgesehenen Reihenfolge aufgerufen. Das Synchronisationstoken spiegelt dann einen nicht synchronen Zustand wider oder ist überhaupt nicht vorhanden. In beiden Fällen kann man falls gewünscht den Zugriff auf die Seite unterbinden.

Ein Beispiel für diese Strategie finden Sie im Abschnitt »Ein Synchronisationstoken einführen« in Kapitel 5.

3.1.3 Gültigkeitsprüfung

Häufig ist es wünschenswert, eine Gültigkeitsprüfung sowohl auf der Client- als auch auf der Serverseite durchzuführen. Obwohl die Gültigkeitsprüfung auf der Clientseite normalerweise weniger anspruchsvoll ist als die Serverprüfung, bietet sie Prüfungen auf höherer Stufe, zum Beispiel, ob ein Formularfeld leer ist. Die serverseitige Gültigkeitsprüfung ist oftmals wesentlich umfassender. Obwohl beide Arten der Verarbeitung in einer Anwendung geeignet sind, empfiehlt es sich nicht, nur mit clientseitiger Gültigkeitsprüfung zu arbeiten. Das ist vor allem darin begründet, dass die clientseitigen Scriptsprachen vom Benutzer konfigurierbar sind und deshalb jederzeit deaktiviert werden können.

Auch wenn die detaillierte Behandlung der Strategien zur Gültigkeitsprüfung über den Rahmen dieses Buchs hinausgeht, möchten wir diesen Punkt anreißen, da man ihn beim Systementwurf beachten muss. Für weitergehende Informationen sei auf die einschlägige Literatur verwiesen.

Gültigkeitsprüfung auf Clientseite

Die Eingabeprüfung findet auf dem Client statt. Normalerweise bettet man dazu Scriptcode – etwa JavaScript – in die Clientansicht ein. Wie oben erwähnt, ist die clientseitige Gültigkeitsprüfung eine gute Ergänzung zur serverseitigen Gültigkeitsprüfung, sollte aber nicht allein verwendet werden.

Gültigkeitsprüfung auf Serverseite

Die Eingabeprüfung findet auf dem Server statt. Hierfür gibt es mehrere typische Strategien, wozu die Formularorientierte und die auf abstrakten Typen basierende Gültigkeitsprüfung gehören.

Formularorientierte Gültigkeitsprüfung

Bei der Formularorientierten Gültigkeitsprüfung muss eine Anwendung jede Menge Methoden einbinden, die verschiedene Bestandteile des Zustands für jedes abgeschickte Formular prüfen. Normalerweise überlappen sich diese Methoden im Hinblick auf die Logik, die sie umfassen, was auf

Kosten der Wiederverwendbarkeit und Modularität geht. Da es für jedes abgeschickte Webformular eine spezielle Methode gibt, ist kein zentraler Code vorhanden, der obligatorische Felder oder rein numerische Felder behandelt. Obwohl es auf mehreren verschiedenen Formularen ein obligatorisches Feld geben kann, behandelt es die Anwendung an verschiedenen Stellen separat und damit redundant. Diese Strategie ist zwar relativ leicht zu implementieren und auch effektiv, führt aber bei umfangreicheren Anwendungen unvermeidlich zu doppeltem Code.

Um eine flexiblere, wiederverwendbare und leichter zu wartende Lösung zu erreichen, kann man die Modelldaten auf einer anderen Abstraktionsebene betrachten. Diesen Ansatz verfolgt die alternative Strategie, die der nächste Abschnitt beschreibt. Beispiel 3.4 zeigt ein Listing, das die Formularorientierte Gültigkeitsprüfung realisiert.

Beispiel 3.4: Formularorientierte Gültigkeitsprüfung

```
/** Wenn die Felder für Vor- oder Nachname (firstname, lastname)
leer bleiben, erhält der Client eine Fehlermeldung zurück. Bei
dieser Strategie sind die Prüfungen auf ein obligatorisches Feld
doppelt vorhanden. Wenn man diese Testlogik in eine separate
Komponente abstrahiert, lässt sie sich für mehrere Formulare
wiederverwenden (siehe den Abschnitt "Gültigkeitsprüfung basierend
auf abstrakten Typen")**/
public Vector validate()
{
Vector errorCollection = new Vector();
    if ((firstname == null) ||
   (firstname.trim().length() < 1))
     errorCollection.addElement("Vorname obligatorisch");
    if ((lastname == null) || (lastname.trim().length() < 1))
     errorCollection.addElement("Nachname obligatorisch");
return errorCollection;}
}
```

Gültigkeitsprüfung basierend auf abstrakten Typen

Diese Strategie lässt sich sowohl auf dem Client als auch auf dem Server nutzen, wird aber in einer Browser-basierten oder Thin Client-Umgebung auf dem Server bevorzugt.

Die Typ- und Einschränkungsinformationen werden aus dem Modellzustand herausgezogen und in einem generischen Framework untergebracht. Dieses trennt die Gültigkeitsprüfung des Modells von der Anwendungslogik, in der das Modell verwendet wird, und lockert somit deren Kopplung.

Zur Modellgültigkeitsprüfung vergleicht man die Metadaten und Einschränkungen mit dem Modellzustand. Die zum Modell gehörenden Metadaten und Einschränkungen lassen sich in der Regel aus einem einfachen Datenspeicher abrufen, beispielsweise aus einer Eigenschaftsdatei. Diese Lösung hat den Vorteil, dass sie das System generischer macht, weil sie die Typ- und Einschränkungsinformationen des Zustands aus der Anwendungslogik ausklammert.

Ein Beispiel ist eine Komponente oder ein Subsystem, das die Logik der Gültigkeitsprüfung kapselt – etwa um zu entscheiden, ob eine Zeichenfolge leer ist, ob eine bestimmte Zahl innerhalb eines gültigen Bereichs liegt oder ob ein String in einer bestimmten Weise formatiert ist. Wollen mehrere unterschiedliche Anwendungskomponenten unterschiedliche Aspekte eines Modells auf Gültigkeit prüfen, erhält nicht jede Komponente ihren eigenen Code zur Gültigkeitsprüfung. Stattdessen verwendet man den Mechanismus der zentralisierten Gültigkeitsprüfung. Diesen konfiguriert man in der Regel entweder per Programm, über irgendeine Art von Fabrik oder deklarativ mithilfe von Konfigurationsdateien.

Somit ist der Mechanismus zur Gültigkeitsprüfung generischer und konzentriert sich unabhängig von anderen Teilen der Anwendung auf den Modellzustand und dessen Anforderungen. Nachteilig bei dieser Strategie ist, dass Effizienz und Performanz leiden können. Außerdem sind allgemeinere Lösungen zwar oftmals recht leistungsfähig, manchmal aber schwerer zu überblicken und nicht so leicht zu warten.

Ein Beispielszenario: Eine XML-basierte Konfigurationsdatei beschreibt eine Vielfalt von Gültigkeitsprüfungen, wie zum Beispiel »obligatorisches Feld« oder »rein numerisches Feld«. Zusätzlich kann man Handler-Klassen für jede dieser Gültigkeitsprüfungen einrichten. Schließlich verknüpft eine Zuordnung die HTML-Formularwerte mit einem spezifischen Typ der Gültigkeitsprüfung. Der Code für die Gültigkeitsprüfung eines bestimmten Formularfelds sieht dann etwa wie das Codefragment in Beispiel 3.5 aus.

Beispiel 3.5: Gültigkeitsprüfung basierend auf abstrakten Typen

```
//firstNameString="Dan"
//formFieldName="form1.firstname"
Validator.getInstance().validate(firstNameString,
  formFieldName);
```

3.1.4 Hilfseigenschaften – Integrität und Konsistenz

JavaBean-Hilfsklassen setzt man normalerweise ein, um einen Zwischenzustand zu speichern, wenn er mit einer Clientanforderung übergeben wird. JSP-Laufzeitmodule bieten einen Mechanismus, der automatisch die Parameterwerte aus einem Servlet-Anforderungsobjekt in Eigenschaften dieser JavaBean-Hilfsobjekte kopiert. Die JSP-Syntax sieht folgendermaßen aus:

```
<jsp:setProperty name="helper" property="*"/>
```

Damit ergeht an das JSP-Modul die Aufforderung, alle *übereinstimmenden* Parameterwerte in die jeweiligen Eigenschaften einer JavaBean namens »helper« zu kopieren, wie es Beispiel 3.6 zeigt.

Beispiel 3.6: Hilfseigenschaften – Eine einfache JavaBean-Hilfsklasse

```
public class Helper
{
  private String first;
  private String last;

  public String getFirst()
  {
    return first;
  }

  public void setFirst(String aString)
  {
    first=aString;
  }

  public String getLast()
  {
    return last;
  }


  public void setLast(String aString)
  {
    last=aString;
  }

}
```

Wie ist aber nun eine Übereinstimmung definiert? Wenn ein Anforderungsparameter mit demselben Namen und demselben Typ wie die Eigenschaft der Hilfs-Bean existiert, gilt das als Übereinstimmung. Praktisch wird jeder Parameter mit jedem Eigenschaftsnamen und dem Typ der Set-Methode für die Eigenschaft der Bean verglichen.

Obwohl dieser Mechanismus einfach ist, kann er zu Unklarheiten und unerwünschten Nebeneffekten führen. Zuallererst ist zu klären, was passiert, wenn ein Anforderungsparameter einen leeren Wert enthält. Viele Entwickler gehen davon aus, dass ein Anforderungsparameter mit einem leeren Zeichenfolgenwert zu einer leeren Zeichenfolge oder Null in der korrespondierenden Bean-Eigenschaft führen sollte. Das der Spezifikation entsprechende Verhalten legt aber fest, dass in diesem Fall keine Änderungen an der korrespondierenden Bean-Eigenschaft stattfinden. Da aber auch Instanzen von JavaBean-Hilfsobjekten normalerweise über Anforderungen hinweg wiederverwendet werden, kann eine derartige Unklarheit zu inkonsistenten und falschen Datenwerten führen. Abbildung 3.1 zeigt ein derartiges Problem, das die geschilderten Effekte verursachen kann.

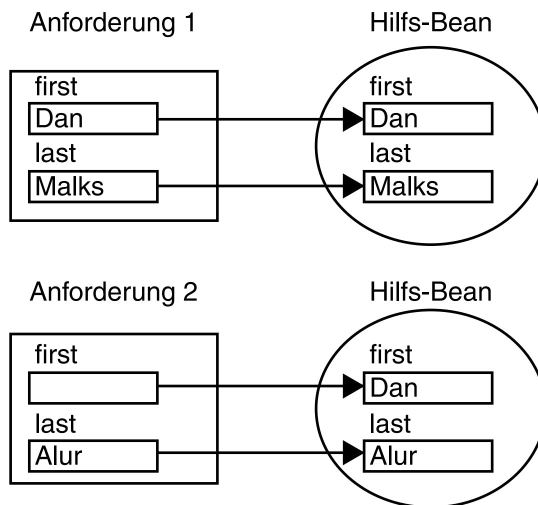

Abbildung 3.1: Helper-Eigenschaften

Anforderung 1 bindet Werte für den Parameter namens »first« und den Parameter namens »last« ein; die beiden korrespondierenden Bean-Eigenschaften werden gesetzt. *Anforderung 2* enthält nur für den Parameter »last« einen Wert und bewirkt damit, dass nur eine Eigenschaft in der Bean zu setzen ist. Der Wert für den Parameter »first« bleibt unverändert. Er wird weder auf eine leere Zeichenfolge noch auf Null gesetzt, einfach deshalb, weil es keinen Wert im Anforderungsparameter gibt. Wie Abbildung 3.1 zeigt, kann das zu fehlerhaften Daten führen, wenn die Bean-Werte zwischen den Anforderungen nicht manuell zurückgesetzt werden.

Beim Anwendungsentwurf ist als verwandtes Problem das Verhalten der HTML-Formularschnittstellen zu berücksichtigen, wenn Steuerelemente des Formulars nicht ausgewählt sind. Hat zum Beispiel ein Formular mehrere Kontrollkästchen, erscheint es logisch anzunehmen, dass das Ausschalten jedes Kontrollkästchen zum Löschen dieser Werte auf dem Server führt. Im Fall des Anforderungsobjekts, das basierend auf dieser Schnittstelle erzeugt wird, gibt es jedoch einfach keinen Parameter, der für irgendeinen der Kontrollkästchenwerte in dieses Anforderungsobjekt eingebunden ist. Folglich werden auch keine Parameterwerte, die sich auf diese Kontrollkästchen beziehen, an den Server gesendet (eine vollständige HTML-Spezifikation finden Sie bei *http://www.w3.org*).

Da kein Parameter zum Server gelangt, bleibt die korrespondierende Bean-Eigenschaft unverändert, wenn man die Aktion `<jsp:setProperty>` wie beschrieben verwendet. Sofern der Entwickler diese Werte nicht manuell modifiziert, besteht hier also die Möglichkeit, dass in der Anwendung widersprüchliche und falsche Datenwerte entstehen können. Wie erwähnt löst man dieses Problem einfach dadurch, dass man den gesamten Zustand in der JavaBean zwischen den Anforderungen zurücksetzt.

3.2 Schlechte Praktiken für die Präsentationsschicht

Schlechte Praktiken sind Lösungen, die nicht optimal sind und vielen Empfehlungen der Muster entgegenlaufen. Als wir die Muster und guten Praktiken dokumentiert haben, sind natürlich diejenigen Praktiken unter den Tisch gefallen, die kein Optimum darstellen.

In diesem Teil des Buches beleuchten wir, was wir als schlechte Praktik in der Präsentationsschicht ansehen.

In jedem Abschnitt beschreiben wir kurz die schlechte Praktik und geben zahlreiche Verweise auf Entwurfsprobleme, Refaktorisierungen und Muster an, die weitere Informationen und zu bevorzugende Alternativen aufzeigen. Wir gehen nicht allzu tief auf jede schlechte Praktik ein, sondern geben eine kurze Zusammenfassung als Ausgangspunkt für weitere Untersuchungen an.

Der Abschnitt »Problemzusammenfassung« bietet eine kurze Beschreibung einer weniger optimalen Situation und der Abschnitt »Lösungsverweis« enthält Referenzen auf:

- *Muster*, die Informationen zum Kontext und zu Kompromissen angeben,
- *Entwurfsbetrachtungen*, die verwandte Details angeben, und
- *Refaktorisierungen*, die den Weg von der weniger optimalen Situation (schlechte Praktik) zu einer eher optimalen, einer Vorzugslösung oder einem Muster beschreiben.

Betrachten Sie diesen Teil des Buchs als Wegweiser, der Sie auf weitere Details und Beschreibungen in anderen Teilen des Buchs aufmerksam macht.

3.2.1 Steuerungscode in Ansichten

Problemzusammenfassung

Am Beginn einer JSP-Ansicht lassen sich benutzerdefinierte Hilfs-Tags einbinden, um die Zugriffssteuerung und andere Arten von Prüfungen durchzuführen. Wenn eine große Zahl von Ansichten ähnliche Hilfsverweise enthält, wird die Verwaltung dieses Codes schwierig, da Änderungen an mehreren Stellen durchzuführen sind.

Lösungsverweise

Fassen Sie den Steuercode in einem Controller und zugeordneten Command-Hilfsobjekten zusammen.

- **Refaktorisierung** siehe »Einen Controller einführen« in Kapitel 5
- **Refaktorisierung** siehe »Unterschiedliche Logik lokalisieren« in Kapitel 5
- **Muster** siehe »Front Controller – Command- und Controller-Strategie« in Kapitel 7

Muss man ähnlichen Steuercode an mehreren Stellen einbinden, wenn beispielsweise nur ein Teil einer JSP-Ansicht gegenüber einem bestimmten Benutzer eingeschränkt ist, delegiert man die Arbeit an eine wiederverwendbare Hilfsklasse.

- **Muster** siehe »View Helper« in Kapitel 7
- **Entwurf** siehe »Eine Ansicht schützen« weiter vorn in diesem Kapitel.

3.2.2 Datenstrukturen der Präsentationsschicht für die Geschäftsschicht offen legen

Problemzusammenfassung

Datenstrukturen der Präsentationsschicht wie `HttpServletRequest` sollte man auf die Präsentationsschicht begrenzen. Wenn man derartige Elemente mit der Geschäftsschicht oder irgendeiner anderen Schicht gemeinsam nutzt, koppelt man diese Schichten enger und verringert die Wiederverwendbarkeit der verfügbaren Dienste damit drastisch. Wenn die Methodensignatur im Geschäftsdienst einen Parameter vom Typ `HttpServletRequest` akzeptiert, dann müssen alle anderen Clients für diesen Dienst (selbst diejenigen außerhalb des Webbereichs) ihren Anforderungszustand in ein `HttpServletRequest`-Objekt einhüllen. In diesem Fall müssen die Dienste der Geschäftsschicht zusätzlich wissen, wie sie mit diesen Präsentationsschicht-spezifischen Datenstrukturen interagieren können. Dadurch bläht sich der Code der Geschäftsschicht unnötig auf und verstärkt die Kopplung zwischen den Schichten.

Lösungsverweis

Statt die Präsentationsschicht-spezifischen Datenstrukturen mit der Geschäftsschicht gemeinsam zu nutzen, kopiert man den relevanten Zustand in generischere Datenstrukturen und nutzt diese gemeinsam. Alternativ kann man den relevanten Zustand aus der Präsentationsschicht-spezifischen Datenstruktur in Form individueller Parameter herausziehen und diese gemeinsam nutzen.

- **Refaktorisierung** siehe »Präsentationsschicht-spezifische Details vor der Geschäftsschicht verbergen« in Kapitel 5

3.2.3 Datenstrukturen der Präsentationsschicht für Domänenobjekte offen legen

Problemzusammenfassung

Wenn man Datenstrukturen wie zum Beispiel `HttpServletRequest` mit Domänenobjekten gemeinsam nutzt, um Anforderungen zu behandeln, erhöht man unnötigerweise die Kopplung zwischen zwei abgegrenzten Aspekten der Anwendung. Domänenobjekte sollten wiederverwendbare Komponenten sein und wenn sich ihre Implementierung auf Protokoll- oder Schicht-spezifische Details stützt, verringern sich ihre Möglichkeiten der Wiederverwendung. Darüber hinaus ist es wesentlich schwieriger, eng gekoppelte Anwendungen zu warten und auf Fehler zu untersuchen.

Lösungsverweis

Statt ein `HttpServletRequest`-Objekt als Parameter zu übergeben, kopiert man den Zustand aus dem Anforderungsobjekt in eine generischere Datenstruktur und nutzt dieses Objekt gemeinsam mit dem Domänenobjekt. Alternativ zieht man den relevanten Zustand aus dem `HttpServletRequest`-Objekt heraus und stellt jeden Teil des Zustands als individuellen Parameter für das Domänenobjekt bereit.

- **Refaktorisierung** siehe »Präsentationsschicht-spezifische Details vor der Geschäftsschicht verbergen« in Kapitel 5

3.2.4 Mehrfaches Formularversenden zulassen

Problemzusammenfassung

Eine der Beschränkungen der Browser-Clientumgebung besteht darin, dass eine Anwendung keine Kontrolle über die Clientnavigation hat. Ein Benutzer kann ein Auftragsformular absenden und damit eine Transaktion veranlassen, die ein Kreditkartenkonto belastet und die Lieferung eines Produkts an seinen Wohnsitz auslöst. Wenn der Benutzer nach Erhalt der Bestätigungsseite auf die Schaltfläche ZURÜCK klickt, könnte dasselbe Formular erneut abgeschickt werden.

Lösungsverweise

Um dieses Problem zu beseitigen, überwacht und steuert man den Anforderungsfluss.

- **Refaktorisierung** siehe »Ein Synchronisationstoken einführen« in Kapitel 5
- **Refaktorisierung** siehe »Den Clientzugriff steuern« weiter vorn in diesem Kapitel
- **Entwurf** siehe »Synchronisations- (oder Déjà vu-) Token« weiter vorn in diesem Kapitel

3.2.5 Vertrauliche Ressourcen für direkten Clientzugriff offen legen

Problemzusammenfassung

Sicherheit gehört zu den wichtigsten Angelegenheiten in Unternehmensumgebungen. Wenn es keinen triftigen Grund gibt, dass ein Client auf bestimmte Informationen direkt zugreift, muss man diese Informationen schützen. Sind bestimmte Konfigurationsdateien, Eigenschaftsdateien, JSPs und Klassendateien nicht richtig geschützt, dann können Clients versehentlich oder böswillig vertrauliche Informationen abrufen.

Lösungsverweise

Schützen Sie vertrauliche Ressourcen, indem sie den direkten Clientzugriff unterbinden.

- **Refaktorisierung** siehe »Ressourcen vor einem Client verbergen« in Kapitel 5
- **Refaktorisierung** siehe »Den Clientzugriff steuern« weiter vorn in diesem Kapitel

3.2.6 Annehmen, dass Bean-Eigenschaften durch <jsp:setProperty> zurückgesetzt werden

Problemzusammenfassung

Während das Verhalten des Standardtags `<jsp:setProperty>` darin besteht, die Parameterwerte einer Anforderung in gleichnamige Eigenschaften eines JavaBean-Hilfsobjekts zu kopieren, kann das Verhalten bei Parametern mit leeren Werten oftmals verwirren. Beispielsweise wird ein Parameter mit einem leeren Wert ignoriert, obwohl viele Entwickler fälschlicherweise annehmen, dass die korrespondierende JavaBean-Eigenschaft einen Nullwert oder eine leere Zeichenfolge erhält.

Lösungsverweis

Berücksichtigen Sie die nicht ganz logisch erscheinende Verfahrensweise, wie Eigenschaften mithilfe des Tags `<jsp:setProperty>` gesetzt werden und initialisieren Sie Bean-Eigenschaften vor ihrer Verwendung.

- **Entwurf** siehe »Hilfseigenschaften – Integrität und Konsistenz« weiter vorn in diesem Kapitel

3.2.7 Fette Controller erstellen

Problemzusammenfassung

Steuercode, der in mehreren JSP-Ansichten vorkommt, lässt sich in vielen Fällen in einen Controller auslagern. Wenn man jedoch den Controller mit zu viel Code überlädt, wird er »zu schwer« und lässt sich schwieriger warten, testen und debuggen. Beispielsweise ist es wesentlich schwieriger, einen Servlet-Controller, insbesondere einen »fetten Controller«, en bloc zu testen, als einzelne Hilfsklassen, die unabhängig vom HTTP-Protokoll sind, als abgeschlossene Einheit zu untersuchen.

Lösungsverweise

Ein Controller ist normalerweise der erste Anlaufpunkt für die Behandlung einer Anforderung. Er sollte aber auch ein Delegationspunkt sein, der in Zusammenarbeit mit anderen Steuerungsklassen agiert. Mit Command-Objekten kapselt man Steuerungscode, zu dem der Controller die Verarbeitung delegiert. Es ist einfacher, diese JavaBean-Command-Objekte als selbstständige Einheit unabhängig vom Servlet-Modul zu testen, als weniger modularen Code zu testen.

- **Refaktorisierung** siehe »Einen Controller einführen« in Kapitel 5
- **Muster** siehe »Front Controller – Command- und Controller-Strategie« in Kapitel 7
- **Refaktorisierung** siehe »Unterschiedliche Logik lokalisieren« in Kapitel 5
- **Muster** siehe »View Helper« in Kapitel 7