PEARSON Education

ADDISON-WESLEY

open source library

Dave Crane, Eric Pascarello, Darren James

Ajax in action

Das Entwicklerbuch für das Web 2.0

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

3 Ordnung schaffen

Inhalt dieses Kapitels

- Umfangreichen clientseitigen Ajax-Code entwickeln und warten
- Refactoring von Ajax-JavaScript-Code
- Häufig in Ajax-Anwendungen eingesetzte Entwurfsmuster
- Modell-Präsentation-Steuerung auf der Serverseite einer Ajax-Anwendung
- Überblick über Ajax-Bibliotheken von Drittanbietern

In Kapitel 2 haben wir alle grundlegenden Technologien abgedeckt, die eine Ajax-Anwendung ausmachen. Mit dem, was wir bis jetzt gelernt haben, ist es schon möglich, die überragende Ajax-Webanwendung zu erstellen, von der Sie schon immer geträumt haben. Es ist jedoch auch möglich, ganz fürchterliche Schwierigkeiten zu bekommen und am Ende ein Wirrwarr aus Code, HTML-Markup und Stilen zu erhalten, dass sich unmöglich warten lässt und eines Tages aus unerfindlichen Gründen nicht mehr funktioniert. Es könnte sogar noch schlimmer sein: Vielleicht haben Sie am Ende eine Anwendung, die gut funktioniert, solange Sie in ihrer Nähe nicht atmen oder ein lautes Geräusch machen. Bei einem privaten Projekt kann so etwas entmutigend sein. Bei der Site des Arbeitgebers oder eines zahlenden Kunden – jemandem der hier und da noch ein paar Verbesserungen haben will – ist eine solche Situation unzweifelhaft erschreckend.

Zum Glück ist dieses Problem schon seit den Anfängen des Computerzeitalters bekannt – und wahrscheinlich schon vorher! Die Menschen haben Möglichkeiten entwickelt, um eine funktionierende Ordnung für komplexe Zusammenhänge und umfangreichen Code zu schaffen. In diesem Kapitel führen wir die Kernwerkzeuge ein, mit deren Hilfe Sie den Überblick über Ihren Code behalten, Ajax-Anwendungen nach den Wünschen Ihrer Auftraggeber schreiben und umschreiben und trotzdem pünktlich nach Hause gehen können.

Ajax bricht mit der früheren Verwendung von DHTML-Technologien, und zwar nicht nur in der Art und Weise, wie diese Technologien integriert werden, sondern auch in dem Maßstab, in dem sie eingesetzt werden. Hier haben wir es mit sehr viel mehr JavaScript zu tun, als in einer klassischen Webanwendung vorkommt, wobei der Code auch sehr viel länger im Browser verbleibt. Daher muss Ajax mit der Komplexität ganz anders umgehen als klassisches DHTML.

In diesem Kapitel geben wir Ihnen einen Überblick über die Werkzeuge und Techniken, mit denen Sie Ihren Code sauber halten können. Nach unserer Erfahrung sind diese Techniken besonders bei der Entwicklung von umfangreichen, komplexen Ajax-Anwendungen nützlich. Wenn Sie nur einfache Ajax-Anwendungen schreiben wollen, sollten Sie zu den beispielorientierten Kapiteln vorblättern, die bei Kapitel 9 beginnen. Wenn Sie mit Refactoring und Entwurfsmustern bereits vertraut sind, können Sie dieses Kapitel überfliegen und mit der Anwendung dieser Techniken auf Ajax in den Kapitel 4 bis 6 fortfahren. Die Grundlagen, die wir hier legen, sind jedoch auch für die Anwendung dieser Ansätze auf JavaScript wichtig, sodass Sie irgendwann zu diesen Abschnitten zurückkehren sollten. Am Ende dieses Kapitels nutzen wir auch die Gelegenheit, um den aktuellen Stand der Bibliotheken von Drittanbietern für Ajax zu untersuchen. Wenn Sie also Frameworks zur Vereinfachung Ihrer Projekte suchen, sollten Sie Abschnitt 3.5 lesen.

3.1 Ordnung in das Chaos

Das wichtigste Werkzeug, das wir anwenden, ist das *Refactoring*, ein Prozess, bei dem Code umgeschrieben wird, um eine größere Klarheit zu erreichen, und nicht, um neue Funktionen hinzuzufügen. Größere Klarheit ist schon an sich sehr zufriedenstellend, aber sie bietet auch einige Vorteile unter dem Strich für etwas handfester orientierte Gemüter.

Neue Funktionen hinzuzufügen und vorhandene Funktionen zu ändern oder zu entfernen, ist bei gut gestaltetem Code gewöhnlich einfacher. Kurz gesagt, solcher Code ist leichter verständlich. Bei schlecht gestaltetem Code funktioniert oft alles so, wie die jeweiligen Anforderungen es verlangen, aber das Programmiererteam ist sich nicht sicher, warum es funktioniert.

Dass sich Anforderungen oft innerhalb eines knappen Zeitrahmens ändern, gehört häufig zur professionellen Kodierarbeit. Mithilfe des Refactorings können Sie Ihren Code sauber und wartungsfreundlich gestalten, sodass Sie geänderten Anforderungen ohne Furcht entgegensehen und sie auch umsetzen können.

In den Beispielen von Kapitel 2 haben wir bereits etwas elementares Refactoring in der Praxis kennen gelernt, als wir JavaScript, HTML und Stylesheets in getrennte Dateien verschoben haben. Das JavaScript wird mit um die 120 Zeilen jedoch relativ lang und vermischt außerdem »niedere« Funktionen (z.B. Anfragen an den Server) mit Code, der sich eigens um unser Listenobjekt kümmert. Wenn wir es mit umfangreicheren Projekten zu tun bekommen, leidet darunter die separate JavaScript-Datei (und, was das angeht, auch das separate Stylesheet). Unser Ziel – kleine, leicht zu lesende und leicht zu ändernde Codeabschnitte zu erstellen, die jeweils ein bestimmtes Problem angehen –, wird häufig als *Trennung der Verantwortlichkeiten* bezeichnet.

Mit Refactoring wird oft das zweite Ziel verfolgt, gebräuchliche Lösungen und Vorgehensweise für bestimmte Dinge zu erkennen und den Code in Richtung auf das jeweilige Muster abzuändern. Auch dies ist schon für sich allein befriedigend, weist auch praktische Vorteile auf. Lassen Sie uns das als Nächstes betrachten.

3.1.1 Muster: Ein gemeinsames Vokabular anlegen

Code, der einem bewährten Muster folgt, arbeitet mit großer Wahrscheinlichkeit zufriedenstellend, ganz einfach, weil er das schon früher getan hat. Viele der Probleme, die es damit gab, wurden bereits bedacht und, wie wir hoffen, beseitigt. Wenn wir Glück haben, hat jemand ein wiederverwendbares Framework für eine bestimmte Vorgehensweise geschrieben.

Diese Vorgehensweise wird als *Entwurfsmuster* bezeichnet. Der Begriff der Muster wurde in den 1970er Jahren geprägt, um Lösungen für architektonische und planerische Probleme zu beschreiben, und er wurde in den letzten zehn Jahren von der Softwareentwicklung übernommen. Serverseitiges Java bringt eine ausgeprägte Kultur von Entwurfsmustern mit, die Microsoft intensiv in das .NET Framework übernommen hat. Der Begriff hat oftmals einen eher abstoßenden akademischen Beigeschmack und wird häufig in dem Bemühen missbraucht, Eindruck zu schinden. Im Grunde genommen jedoch ist ein Entwurfsmuster lediglich die Beschreibung einer wiederholbaren Möglichkeit, ein bestimmtes Problem beim Softwaredesign zu lösen. Mit Entwurfsmustern können wir abstrakten technischen Lösungen einen Namen geben, sodass es einfacher wird, darüber zu sprechen und sie zu verstehen.

Entwurfsmuster können auch für das Refactoring sehr wichtig sein, weil sie uns in die Lage versetzen, unser Ziel kurz zu beschreiben. Zu sagen, dass wir »diese Code-Bits in ein Objekt verschieben, um den Prozess der Benutzeraktion zu kapseln, damit er jederzeit rückgängig gemacht werden kann« ist ziemlich lang – und außerdem ein viel zu wortreiches Ziel, um es beim Umschreiben des Codes im Sinn zu behalten. Wenn wir sagen, dass wir das Muster `Befehl` in den Code einführen, haben wir ein Ziel, das nicht nur viel knapper gefasst ist, sondern über das sich auch leichter sprechen lässt.

Wenn Sie ein erfahrener Java-Serverentwickler oder ein Software-Architekt welcher Richtung auch immer sind, werden Sie sich wahrscheinlich fragen, was daran neu sein soll. Wenn Sie aus der Welt des Webdesigns und der neuen Medien stammen, glauben Sie wahrscheinlich, dass wir zu den sonderbaren Kontrollfreaks gehören, die lieber Diagramme zeichnen, als echten Code zu schreiben. Wahrscheinlich wundern Sie sich in jedem Falle, was das mit Ajax zu tun hat. Unsere kurze Antwort lautet: »Eine ganze Menge.« Lassen Sie uns untersuchen, was der Ajax-Programmierer in der Praxis von Refactoring hat.

3.1.2 Refactoring und Ajax

Wir haben bereits festgestellt, dass Ajax-Anwendungen wahrscheinlich mehr JavaScript-Code verwenden und dass dieser Code länger bestehen bleibt.

In einer klassischen Webanwendung ist der komplexe Code auf dem Server aktiv, wobei Entwurfsmuster routinemäßig auf den PHP-, Java- oder .NET-Code angewendet werden, der dort läuft. Bei Ajax können wir dieselben Techniken für den Clientcode einsetzen.

Es gibt sogar ein Argument dafür, dass JavaScript diese Art der Gliederung stärker benötigt als die fester strukturierten Gegenstücke Java und C#. Trotz seiner C-ähnlichen Syntax ist JavaScript eher mit Sprachen wie Ruby, Python und selbst Common Lisp verwandt als mit Java oder C#. Es bietet eine enorme Flexibilität und Möglichkeiten für die Entwicklung persönlicher Stile und Dialekte. In den Händen eines brillanten Entwicklers kann das wunderbar sein, aber für den Durchschnittsprogrammierer wird das Sicherheitsnetz dadurch durchlässiger. Sprachen für Unternehmensanwendungen wie Java und C# sind für die Verwendung durch Teams aus Durchschnittsprogrammierern und einen häufigen Wechsel der Teammitglieder entworfen; JavaScript nicht.

Die Gefahr, verwirrenden, undurchsichtigen JavaScript-Code zu schreiben, ist relativ hoch, und wenn wir den Umfang von einfachen Tricks für Webseiten zu Ajax-Anwendungen erhöhen, kann sich diese Erkenntnis schmerzhaft bemerkbar machen. Aus diesem Grunde empfehle ich die Verwendung von Refactoring in Ajax viel stärker als für Java oder C#, die »sicheren« Sprachen, in deren Communitys Designmuster zur Blüte kamen.

3.1.3 Maß halten

Bevor wir fortfahren, ist es sinnvoll festzuhalten, dass Refactoring und Entwurfsmuster lediglich Werkzeuge sind, die nur dort eingesetzt werden sollten, wo sie auch tatsächlich sinnvoll sind. Bei übermäßiger Nutzung können sie zur so genannten *Paralyse durch Analyse* führen, bei der die Implementierung einer Anwendung immer wieder hinausgezögert wird, weil eine neues Design auf das nächste folgt, um die Flexibilität der Struktur zu erhöhen oder Vorkehrungen für mögliche zukünftige Änderungen zu schaffen, die wahrscheinlich niemals umgesetzt werden.

Erich Gamma, Experte für Entwurfsmuster, hat dies vor kurzem in einem Interview auf den Punkt gebracht (siehe den Abschnitt *Quellen* am Ende dieses Kapitels). Dabei berichtete er von dem Hilferuf eines Lesers, der es nur geschafft hatte, 21 der 23 Entwurfsmuster aus dem bahnbrechenden Buch *Entwurfsmuster* in seiner Anwendung einzusetzen. Kein Entwickler würde sich darum bemühen, in jedem Stückchen Code, das er schreibt, Integer-Werte, Strings und Arrays zu verwenden. Ebenso sind Entwurfsmuster auch nur jeweils in bestimmten Situationen sinnvoll.

Gamma empfiehlt das Refactoring als beste Möglichkeit, um Entwurfsmuster einzuführen. Schreiben Sie den Code zuerst in der einfachsten funktionierenden Form und führen Sie dann Muster ein, um übliche Probleme zu lösen, wenn Sie darauf stoßen. Wenn Sie bereits eine Menge Code geschrieben haben oder den Auftrag haben, das verworrene Durcheinander eines anderen zu warten, können Sie jetzt vielleicht befürchten, dass die Party ohne Sie stattfindet. Zum Glück aber ist es möglich, Entwurfsmuster auch im Nachhinein auf Code jedweder Qualität anzuwenden. Im nächsten Abschnitt untersuchen wir, welche Art von Refactoring wir auf den in Kapitel 2 entwickelten Rohcode anwenden können.

3.1.4 Refactoring in der Praxis

Refactoring mag sich für Sie vielleicht wie eine gute Idee anhören, aber wenn Sie etwas praktischer veranlagt sind, werden Sie sicher Beispiele dafür sehen wollen, bevor Sie davon überzeugt sind. Nehmen wir uns einige Minuten Zeit, um auf die in Listing 2.11 des vorhergehenden Kapitels entwickelte Ajax-Kernfunktion ein Refactoring anzuwenden. Um die Struktur dieses Codes kurz zu wiederholen: Wir haben eine Funktion `sendRequest()` definiert, die eine Anfrage an den Server auslöst. `sendRequest()` delegiert die Aufgabe, ein geeigneteres XMLHttpRequest-Objekt zu finden, an `initHttpRequest()` und weist die hartkodierte Rückruffunktion `onReadyState()` zu, um die Antwort zu verarbeiten. Das XMLHttpRequest-Objekt ist als globale Variable definiert, was es der Rückruffunktion erlaubt, einen Verweis darauf aufzunehmen. Der Rückrufhandler fragt dann den Zustand des Anfrageobjekts ab und ruft Debuginformationen hervor.

Der Code in Listing 2.11 tut, was von ihm verlangt wird, aber er ist in gewisser Hinsicht schwer wiederzuverwenden. Wenn wir eine Anfrage an einen Server stellen, möchten wir gewöhnlich die Antwort analysieren und die Ergebnisse eigens nach den Anforderungen unserer Anwendung verarbeiten. Um benutzerdefinierte Geschäftslogik in den Code einbauen zu können, müssen wir einige Abschnitte der Funktion `onReadyState()` ändern.

Auch das Vorhandensein der globalen Variable stellt ein Problem dar. Wenn wir gleichzeitige mehrere Aufrufe an den Server ausgeben möchten, müssen wir in der Lage sein, ihnen jeweils unterschiedliche Rückrufhandler zuzuweisen. Wenn wir eine Liste von Ressourcen abrufen, die aktualisiert werden sollen, und einige, die zu verwerfen sind, ist es schließlich sehr wichtig zu wissen, welche Liste welche ist.

Bei der objektorientierten Programmierung (OO) besteht die Standardlösung zu dieser Art von Problem darin, die erforderliche Funktionalität in einem Objekt zu kapseln. JavaScript unterstützt OO-Kodierstile gut genug, um dies zu tun. Wir nennen unser Objekt `ContentLoader`, da es Inhalte vom Server lädt. Wie soll dieses Objekt aber aussehen? Im Idealfall erstellen wir eines und übergeben ihm einen URL, zu dem die Anfrage gesendet werden soll. Wir müssen auch in der Lage sein, einen Verweis auf einen benutzerdefinierten Rückrufhandler zu übergeben, der ausgeführt wird, wenn das Dokument erfolgreich geladen wird, und auf einen anderen, der bei einem Fehler ausgeführt wird. Ein Aufruf dieses Objekts kann wie folgt aussehen:

```
var loader=new net.ContentLoader('mydata.xml',parseMyDate)
```

Dabei ist `parseMyData` eine Rückruffunktion, die aufgerufen wird, wenn das Objekt erfolgreich geladen wird. Listing 3.1 zeigt den erforderlichen Code zur Implementierung des Objekts `ContentLoader`. Es treten einige neue Begriffe auf, die wir als Nächstes erklären.

Listing 3.1: Das Objekt ContentLoader

```
var net=new Object();                          ❶ Namespace-Objekt
net.READY_STATE_UNINITIALIZED=0;
net.READY_STATE_LOADING=1;
net.READY_STATE_LOADED=2;
net.READY_STATE_INTERACTIVE=3;
net.READY_STATE_COMPLETE=4;
net.ContentLoader=function(url,onload,onerror){    ❷ Konstruktorfunktion
  this.url=url;
  this.req=null;
  this.onload=onload;
  this.onerror=(onerror) ? onerror : this.defaultError;
  this.loadXMLDoc(url);
}
net.ContentLoader.prototype={
 loadXMLDoc:function(url){    ❸ Umbenannte initXMLHttpRequest()-Funktion
  if (window.XMLHttpRequest){
    this.req=new XMLHttpRequest();
  } else if (window.ActiveXObject){
    this.req=new ActiveXObject("Microsoft.XMLHTTP");
  }
  if (this.req){
    try{
      var loader=this;
      this.req.onreadystatechange=function(){
        loader.onReadyState.call(loader);
      }
      this.req.open('GET',url,true);
      this.req.send(null);
    }catch (err){
      this.onerror.call(this);
    }
  }
 },
 onReadyState:function(){    ❻ Umgearbeiteter Callback
  var req=this.req;
  var ready=req.readyState;
  if (ready==net.READY_STATE_COMPLETE){
    var httpStatus=req.status;
    if (httpStatus==200 || httpStatus==0){
      this.onload.call(this);
    }else{
      this.onerror.call(this);
    }
  }
 },
 defaultError:function(){
  alert("error fetching data!"
```

❹ Durch Refactoring umgearbeitete loadXML-Funktion

❺ Umgearbeitete sendRequest-Funktion

```
    +"\n\nreadyState:"+this.req.readyState
    +"\nstatus: "+this.req.status
    +"\nheaders: "+this.req.getAllResponseHeaders());
 }
}
```

Was an diesem Code zuerst auffällt, ist, dass wir eine einzelne globale Variable `net` ❶ definiert und alle unsere anderen Verweise daran angefügt haben. Das minimiert das Risiko von Konflikten in Variablennamen und hält den Code für Netzwerkanfragen an einem Ort zusammen.

Für unser Objekt stellen wir eine einzelne Konstruktorfunktion bereit ❷. Sie hat drei Argumente, aber nur die ersten beiden sind erforderlich. Im Falle des Fehlerhandlers führen wir eine Überprüfung auf NULL-Werte hin durch und geben falls nötig einen sinnvollen Standardwert vor. Die Möglichkeit, einer Funktion eine schwankende Zahl von Argumenten zu übergeben, mag für OO-Programmierer merkwürdig erscheinen, ebenso wie die Möglichkeit, Funktionen als Verweise erster Klasse zu übergeben. Dies sind jedoch übliche Merkmale von JavaScript. Die Besonderheiten dieser Sprache besprechen wir im Einzelnen in Anhang B.

Wir haben große Teile der Funktionen `iniXMLHttpRequest()` ❹ und `sendRequest()` ❺ aus Listing 2.11 entnommen und in das Objekt eingefügt. Außerdem haben wir die Funktion umbenannt, um ihren größeren Gültigkeitsbereich anzudeuten: Sie heißt jetzt `loadXMLDoc` ❸. Wir verwenden nach wie vor dieselbe Technik, um ein XMLHttpRequest-Objekt zu finden und eine Anfrage auszulösen, aber der Benutzer des Objekts muss sich nicht mehr darum kümmern. Die Rückruffunktion `onReadyState` ❻ sollte Ihnen von Listing 2.11 her auch zum größten Teil vertraut vorkommen. Wir haben den Aufruf der Debug-Konsole durch Aufrufe der Funktionen `onload` und `onerror` ersetzt. Die Syntax mag etwas merkwürdig erscheinen; nehmen wir Sie also näher in Augenschein. `onload` und `onerror` sind Function-Objekte, und `Function.call()` ist eine Methode dieses Objekts. Das erste Argument von `Function.call()` wird zum Kontext der Funktion, es kann also innerhalb der aufgerufenen Funktion durch das Schlüsselwort `this` darauf verwiesen werden.

Einen Rückrufhandler zu schreiben, der an unseren `ContentLoader` übergeben wird, ist damit recht einfach. Wenn wir auf eine Eigenschaft von `ContentLoader` verweisen müssen, z.B. auf XMLHttpRequest oder `url`, können wir dazu wie folgt einfach `this` verwenden:

```
function myCallBack(){
   alert(
      this.url
      +" loaded! Here's the content:\n\n"
      +this.req.responsteText
   );
}
```

Die notwendigen Grundlagen zu schaffen, erfordert Kenntnis der Eigenheiten von JavaScript, aber sobald das Objekt einmal geschrieben ist, muss sich der Endbenutzer nicht darum kümmern.

Diese Situation ist häufig ein Zeichen eines guten Refactorings. Wir haben die schwierigeren Stellen des Codes ins Innere des Objekts verschoben und präsentieren ein einfach zu verwendendes Äußeres. Der Endbenutzer wird vor einer Menge unnötiger Schwierigkeiten bewahrt, und der Experte, der für die Wartung des schwierigen Codes verantwortlich ist, findet diesen an einem einzigen Ort isoliert. Korrekturen müssen nur einmal angewendet werden, um für den ganzen Code zu gelten.

Damit haben wir die Grundlagen des Refactorings abgearbeitet und gezeigt, wie wir es zu unserem Vorteil in der Praxis einsetzen. Im nächsten Abschnitt schauen wir uns einige der üblichen Probleme bei der Ajax-Programmierung an und erfahren, wie wir dafür das Refactoring einsetzen können. Dabei lernen wir auch einige nützliche Tricks kennen, die wir in den folgenden Kapiteln wiederverwenden und die Sie in Ihren eigenen Projekten ebenfalls einsetzen können.

3.2 Fallstudien zum Refactoring

In den folgenden Abschnitten beschäftigen wir uns mit einigen Problemen bei der Ajax-Entwicklung und schauen uns gebräuchliche Lösungen dafür an. In jedem Fall führen wir ein Refactoring vor, mit dem das Problem gemildert werden kann, und zeigen auf, welche Elemente dieser Lösung anderweitig wiederverwendet werden können.

Im Zuge einer altehrwürdigen Tradition der Literatur über Entwurfsmuster stellen wir die einzelnen Punkte jeweils in Form eines Problems, der technischen Lösung und einer Erörterung der wichtigsten Gesichtspunkte dar.

3.2.1 Browserinkonsistenzen: Die Muster Fassade und Adapter

Fragen Sie beliebige Webentwickler – seien es Programmierer, Designer, Grafiker oder Allround-Talente – nach ihrer meistgehassten Tätigkeit, dann besteht eine große Wahrscheinlichkeit dafür, dass sich »dafür sorgen, dass meine Arbeit auf verschiedenen Browsern korrekt angezeigt wird« ganz oben auf der Liste befindet. Das Web ist voll von technischen Standards, und die meisten Browserhersteller implementieren die meisten dieser Standards meistens ziemlich vollständig. Manchmal sind diese Standards jedoch etwas vage und können verschieden interpretiert werden, manchmal erweitern die Browserhersteller bestehende Standards auf nützliche, aber proprietäre Weise, und manchmal sind die Browser einfach mit einigen altmodischen Bugs behaftet.

JavaScript-Programmierer sind schon von Anfang an darauf ausgewichen, im Code zu überprüfen, welcher Browser verwendet wird oder ob ein bestimmtes Objekt vorhanden ist. Lassen Sie uns ein sehr einfaches Beispiel betrachten.

DOM-Elemente

Wie wir in Kapitel 2 erklärt haben, wird eine Webseite für JavaScript über das DOM (Document Object Model) bereitgestellt, eine baumartige Struktur, deren Element den Tags eines HTML-Dokuments entsprechen. Zu den üblichen Tätigkeiten bei der programmgesteuerten Bearbeitung eines DOM-Baums gehört es, die Position eines Elements auf der Seite herauszufinden. Leider haben die Browserhersteller in den vergangenen Jahren verschiedene nicht standardmäßige Methoden dafür bereitgestellt, was es schwierig macht, browserübergreifend sicheren Code zu schreiben, um diese Aufgabe zu erfüllen. Listing 3.2 – die vereinfachte Version einer Funktion aus der x-Bibliothek von Mike Foster (siehe Abschnitt 3.5) – zeigt eine umfassende Möglichkeit, die Pixelposition der linken Kante des DOM-Elements *e* zu bestimmen, das als Argument übergeben wird.

Listing 3.2: Die Funktion getLeft()

```
function getLeft(e){
  if(!(e=xGetElementById(e))){
    return 0;
  }
  var css=xDef(e.style);
  if (css && xStr(e.style.left)) {
    iX=parseInt(e.style.left);
    if(isNaN(iX)) iX=0;
  }else if(css && xDef(e.style.pixelLeft)) {
    iX=e.style.pixelLeft;
  }
  return iX;
}
```

Die verschiedenen Browser bieten mehrere Möglichkeiten, um die Position eines Knotens über das Stil-Array zu bestimmen, das wir in Kapitel 2 kennen gelernt haben. Der W3C-Standard CSS2 unterstützt eine Eigenschaft namens `style.left`, definiert als String, der den Wert und die Einheit beschreibt, z.B. in der Form `100px`. Auch andere Einheiten als Pixel können unterstützt werden. Dagegen ist `style.pixelLeft` numerisch und setzt voraus, dass alle Werte in Pixel gemessen werden. Diese Eigenschaft wird aber nur von Microsoft Internet Explorer unterstützt. Die hier vorgestellte Methode `getLeft()` schaut zuerst, ob CSS unterstützt wird, und prüft dann beide Werte nach, wobei sie es zuerst mit dem W3C-Standard versucht. Wenn keine Werte gefunden werden, wird standardmäßig der Wert null zurückgegeben. Beachten Sie, dass wir keine explizite Überprüfung auf Browsernamen oder -versionen durchführen, sondern die solidere Objekterkennungstechnik aus Kapitel 2 verwenden.

Funktionen wie diese zu schreiben, um sich an die Eigenheiten verschiedener Browser anzupassen, ist harte Arbeit, aber nachdem sie einmal erledigt ist, kann der Entwickler die Anwendung schreiben, ohne sich über diese Probleme Gedanken zu machen. In gut getesteten Bibliotheken wie x ist der Großteil der harten Arbeit auch

schon für uns erledigt worden. Eine zuverlässige Adapterfunktion zur Ermittlung der Position eines DOM-Elements auf der Seite kann die Entwicklung einer Ajax-Benutzeroberfläche beträchtlich beschleunigen.

Anfragen an den Server senden

In Kapitel 2 haben wir bereits eine andere Browserinkompatibilität kennen gelernt. Die Browserhersteller haben nicht standardmäßige Mechanismen zum Abrufen des XMLHttpRequest-Objekts bereitgestellt, das für asynchrone Anfragen an den Server dient. Wenn wir ein XML-Dokument vom Server laden möchten, müssen wir herausfinden, welche dieser Möglichkeiten wir zu nutzen haben.

Der Internet Explorer arbeitet nur dann richtig, wenn wir nach einer ActiveX-Komponente fragen, während Mozilla und Safari mitspielen, wenn wir nach einem nativen integrierten Objekt suchen.[1] Nur der XML-Ladecode beachtet die Unterschiede. Ist das XMLHttpRequest-Objekt erst einmal an den restlichen Code zurückgegeben worden, verhält es sich in beiden Fällen identisch. Der aufrufende Code muss weder das ActiveX-Subsystem noch das des nativen Objekts kennen, sondern nur den Konstruktor `net.ContentLoader()`.

Das Muster Fassade

Sowohl bei `getLeft()` als auch bei `net.ContentLoader()` ist der Code für die Objektbestimmung hässlich und mühselig. Wenn wir eine Funktion definieren, um diesen Code vor dem restlichen Code zu verbergen, machen wir den Rest leichter zu lesen und isolieren den Objekterkennungscode an einem einzelnen Ort. Dies ist das grundlegende Prinzip des Refactorings: »Wiederholen Sie sich nicht (*Don't Repeat Yourself*, kurz DRY)!« Wenn wir einen Spezialfall entdecken, den unser Objektbestimmungscode nicht richtig handhabt, müssen wir ihn nur einmal korrigieren, woraufhin diese Änderung dann für alle Aufrufe gilt, die die linke Koordinate eines DOM-Elements bestimmen, ein XML-Anfrageobjekt erstellen oder was auch immer wir sonst tun wollen.

In der Sprache der Entwurfsmuster heißt das, dass wir ein Muster namens *Fassade* einsetzen. Es wird verwendet, um einen gemeinsamen Zugriffspunkt für verschiedene Implementierungen eines Dienstes oder einer Teilfunktionalität zu gewähren. Das XMLHttpRequest-Objekt bietet z. B. einen nützlichen Dienst, und unsere Anwendung kümmert es nicht, wie es bereitgestellt wird, solange es funktioniert (siehe Abbildung 3.1).

In vielen Fällen möchten wir auch den Zugriff auf ein Subsystem vereinfachen. Beim Bestimmen der linken Koordinate eines DOM-Elements bietet uns die CSS-Spezifikation eine Menge an Auswahlmöglichkeiten, indem der Wert in Pixeln, Punkten, Ems und anderen Einheiten angegeben werden kann. Diese Freiheit des Ausdrucks kann mehr sein, als wir benötigen. Die Funktion `getLeft()` aus Listing 3.2 funktioniert,

1 *Anm. d. Fachlekt.: Dieses Verhalten ändert sich mit dem IE 7 – ab dieser Version ist XMLHttpRequest nativ dabei.*

solange wir in unserem gesamten Layoutsystem Pixel als Einheiten verwenden. Die Vereinfachung des Subsystems auf diese Weise ist ein weiteres Merkmal des Musters Fassade.

Das Muster Adapter

Adapter ist ein eng verwandtes Muster. Auch hier arbeiten wir mit zwei Subsystemen, die dieselbe Funktion ausüben, wie die Ansätze von Microsoft und Mozilla zum Abruf des XMLHttpRequest-Objekts. Anstatt wie zuvor für jeden dieser Ansätze eine eigene Fassade aufzubauen, fügen wir über einem der Subsysteme eine zusätzliche Schicht ein, die dieselbe API darstellt wie das andere Subsystem. Diese Schicht wird *Adapter* genannt. Die Sarissa XML-Bibliothek für Ajax, die wir in Abschnitt 3.5.1 besprechen, verwendet das Adapter-Muster, damit das ActiveX-Steuerelement von Internet Explorer wie das integrierte XMLHttpRequest-Objekt von Mozilla aussieht. Beide Ansätze sind gültig und hilfreich bei der Integration von veraltetem Code oder Code von Drittherstellern (einschließlich der beiden Browser) in Ihr Ajax-Projekt.

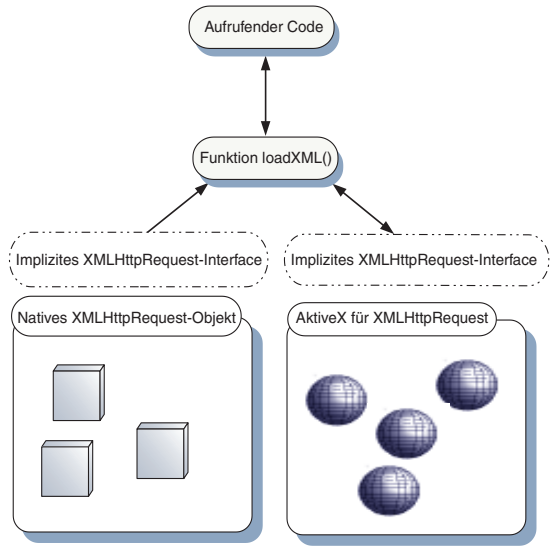

Abbildung 3.1: Schema des Musters Fassade für das XMLHttpRequest-Objekt in verschiedenen Browsern. Die Funktion loadXML() erfordert ein XMLHttpRequest-Objekt, kümmert sich aber nicht um die tatsächliche Implementierung. Die zugrunde liegende Implementierung kann eine deutlich komplexe Semantik der HTTP-Anfrage aufweisen, aber beide Varianten sind hier vereinfacht, um die grundlegende Funktionalität bereitzustellen, die von der aufrufenden Funktion benötigt wird.

Lassen Sie uns in der nächsten Fallstudie Probleme mit dem Ereignisverarbeitungsmodell von JavaScript untersuchen.

3.2.2 Ereignishandler: Das Muster Beobachter

Wir können nicht sehr viel Ajax-Code schreiben, ohne auf ereignisbasierte Programmiertechniken zu stoßen. JavaScript-Benutzeroberflächen sind sehr stark ereignisgesteuert, und die Einführung von asynchronen Anfragen durch Ajax fügt einen weiteren Satz von Rückrufen und Ereignissen hinzu, mit denen unsere Anwendung umgehen können muss. In einer relativ einfachen Anwendung kann ein Ereignis wie ein Mausklick oder die Ankunft der Daten vom Server von einer einzigen Funktion verarbeitet werden. Bei größeren und komplexeren Anwendungen jedoch müssen wir wahrscheinlich mehrere verschiedene Subsysteme benachrichtigen und sogar einen Mechanismus bereitstellen, durch den sich die interessierten Parteien für eine solche Benachrichtigung eintragen können. Lassen Sie uns dazu ein Beispiel anschauen.

Mehrere Ereignishandler verwenden

Es ist übliche Praxis, Skripte für DOM-Knoten mit JavaScript in der Methode `window.onload` zu definieren, die nach dem vollständigen Laden der Seite (und damit des DOM-Baums) ausgeführt wird. Nehmen wir an, dass sich auf unserer Seite ein DOM-Element befindet, das nach dem Laden der Seite in regelmäßigen Abständen vom Server abgerufene, dynamisch generierte Daten anzeigt. Das JavaScript, das den Datenabruf und die Anzeige koordiniert, braucht einen Verweis auf den DOM-Knoten. Den erhält es durch die Definition eines `load`-Ereignisses:

```
window.onload=function(){
   displayDiv=document.getElementById('display');
}
```

Gut und schön. Jetzt wollen wir aber eine zweite Anzeige hinzufügen, die z.B. Benachrichtigungen aus einem News-Feed bereitstellt (wenn Sie an der Implementierung dieser Funktion interessiert sind, lesen Sie Kapitel 13). Der Code, der die Anzeige des News-Feeds steuert, muss beim Start auch Verweise auf DOM-Elemente erhalten. Auch für ihn wird also ein `window.onload`-Ereignishandler definiert:

```
window.onload=function(){
   feedDiv=document.getElementById('feeds');
}
```

Wir testen beide Codeteile auf eigenen Seiten und stellen fest, dass sie hervorragend funktionieren. Wenn wir sie aber kombinieren, überschreibt die zweite `window.onload`-Funktion die erste, sodass die Datenanzeige nicht mehr erscheint, sondern JavaScript-Fehler hervorruft. Das Problem liegt darin, dass es nicht zulässig ist, mit dem window-Objekt mehr als eine `onload`-Funktion zu verknüpfen.

Einschränkungen für zusammengesetzte Ereignishandler

Unser zweiter Ereignishandler überschreibt den ersten. Wir können dieses Problem lösen, indem wir eine einzige, zusammengesetzte Funktion schreiben:

```
window.onload=function(){
   displayDiv=document.getElementById('display');
   feedDiv=document.getElementById('feeds');
}
```

Das funktioniert zwar in unserem Beispiel, aber es vermischt Code aus der News-Feed- und der Datenanzeige, die ansonsten nichts miteinander zu tun haben. Wenn wir es nicht mit zwei, sondern mit zehn oder zwanzig Systemen zu tun haben, und Verweise auf verschiedene DOM-Elemente benötigen, würde ein zusammengesetzter Ereignishandler wie dieser nur schwer zu warten sein. Das Austauschen einzelner Komponenten wäre schwierig und fehleranfällig und würde genau zu der Situation führen, die wir in der Einleitung beschrieben haben – niemand möchte den Code noch anfassen, damit er nicht beschädigt wird. Versuchen wir also ein Refactoring anzuwenden, indem wir eine Ladefunktion für jedes Subsystem definieren:

```
window.onload=function(){
   getDisplayElement();
   getFeedElements();
}
function getDisplayElements(){
   displayDiv=document.getElementById('display');
function getFeedElements(){
   feedDiv=document.getElementById('feeds');
}
```

Dadurch wird eine gewisse Klarheit eingeführt und die zusammengesetzte Funktion `window.onload()` auf eine einzige Zeile pro Subsystem reduziert. Die zusammengesetzte Funktion ist aber immer noch ein schwacher Punkt in diesem Design und wird uns wahrscheinlich Ärger bereiten. Im folgenden Abschnitt schauen wir uns eine etwas komplexere, aber besser skalierbare Lösung für dieses Problem an.

Das Muster Beobachter

Manchmal ist es hilfreich, sich zu fragen, wo die Zuständigkeit für einen Vorgang liegt. Der Ansatz mit der zusammengesetzten Funktion verschiebt die Zuständigkeit für den Abruf von Verweisen auf DOM-Elemente auf das window-Objekt, das daher wissen muss, welche Subsysteme auf der aktuellen Seite vorhanden sind. Im Idealfall sollte aber jedes Subsystem selbst für den Abruf seiner Verweise zuständig sein. Dann würde es die Verweise bekommen, wenn es auf der Seite vorhanden ist, während sie nicht abgerufen werden, wenn es gar nicht da ist.

Um die Aufteilung der Zuständigkeit zu bestimmen, können wir den Systemen erlauben, sich für die Benachrichtigung bei `onload`-Ereignissen zu registrieren, die durch die Übergabe eines Funktionsaufrufs beim Auslösen des Ereignisses `window.onload` erfolgt. Im Folgenden finden Sie eine einfache Implementierung:

```
window.onloadListeners=new Array();
window.addOnLoadListener(listener){
   window.onloadListeners[window.onloadListeners.length]=listener;
}
```

Wenn das Fenster vollständig geladen ist, muss dass window-Objekt lediglich sein Array von Listenern iterativ durchlaufen und diese Listener nacheinander aufrufen:

```
window.onload=function(){
   for(var i=0;i<window.onloadListeners.length;i++){
      var func=windows.onloadListeners[i];
      func.call();
   }
}
```

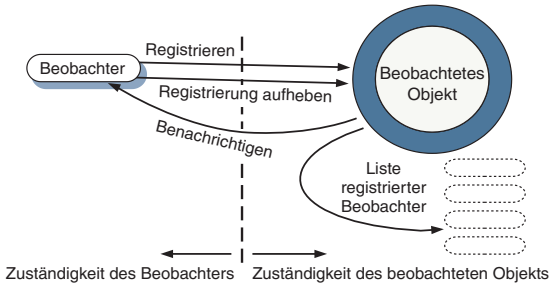

Abbildung 3.2: Aufteilung der Zuständigkeiten im Muster Beobachter. Objekte, die von einem Ereignis benachrichtigt werden sollen, die Beobachter, können sich bei der Ereignisquelle, dem beobachteten Objekt, registrieren (und die Registrierung wieder aufheben). Das beobachtete Objekt benachrichtigt alle registrierten Beobachter, wenn das Ereignis eintritt.

Wenn jedes Subsystem diesen Ansatz verwendet, können wir eine sehr viel sauberere Möglichkeit zur Einrichtung der Subsysteme anbieten, ohne sie miteinander zu vermischen. Natürlich ist nur ein kleines bisschen schlechter Code notwendig, um `window.onload` zu überschreiben und das ganze System zum Zusammenbruch zu bringen. Um dies zu verhindern, müssen wir an irgendeinem Punkt die Verantwortung für unseren Code übernehmen.

Das neuere Ereignismodell des W3C implementiert auch ein System mit mehreren Ereignishandlern. Wir haben uns hier jedoch dazu entschieden, unser eigenes auf der Grundlage des alten JavaScript-Ereignismodells zu konstruieren, da das W3C-Modell nicht browserübergreifend konsistent implementiert ist. Einzelheiten darüber erfahren Sie in Kapitel 4.

Das Entwurfsmuster, das wir in diesem Refactoring einführen, heißt *Beobachter*. Es definiert ein beobachtetes Objekt, in unserem Fall das integrierte window-Objekt, und einen Satz von Beobachtern oder Listenern, die sich dafür registrieren können (siehe Abbildung 3.2).

Beim Muster Beobachter wird die Zuständigkeit sinnvoll zwischen der Ereignisquelle und dem Ereignishandler aufgeteilt. Die Handler sind dafür verantwortlich, sich selbst zu registrieren und die Registrierung wieder aufzuheben, während die Ereignisquelle die Zuständigkeit dafür übernimmt, eine Liste der registrierten Beobachter zu unterhalten und Benachrichtigungen auszugeben, wenn das Ereignis eintritt. Bei der ereignisgesteuerten Programmierung von Benutzeroberflächen wird dieses Muster schon lange eingesetzt, und wir werden darauf zurückkommen, wenn wir uns JavaScript-Ereignisse in Kapitel 4 im Einzelnen ansehen. Wie wir sehen werden, können wir dieses Muster auch in unseren eigenen Codeobjekten für andere Dinge als die Browserverarbeitung von Maus- und Tastenereignissen einsetzen.

3.2.3 Handler von Benutzeraktionen wiederverwenden: Das Muster Befehl

Den meisten Anwendungen sagt der Benutzer (durch Mausklicks oder über die Tastatur), was sie tun sollen, und dann tun sie es. In einem einfachen Programm reichte es aus, dem Benutzer eine Möglichkeit anzubieten, eine Aktion auszuführen, aber in komplexeren Benutzeroberflächen möchten wir dem Benutzer die Möglichkeit geben, eine Aktion auf verschiedenen Wegen auszulösen.

Eine Schaltflächenkomponente implementieren

Nehmen wir an, wir haben ein DOM-Element so gestaltet, dass es wie eine Schaltflächenkomponente aussieht. Wenn der Benutzer auf diese Schaltfläche klickt, führt sie eine Berechnung durch und aktualisiert eine HTML-Tabelle mit dem Ergebnis. Für dieses `button`-Element können wir wie folgt einen Mausklick-Ereignishandler definieren:

```
function buttonOnclickHandler(event){
  var data=new Array();
  data[0]=6;
  data[1]=data[0]/3;
  data[2]=data[0]*data[1]+7;
  var newRow=createTableRow(dataTable);
  for (var i=0;i<data.length;i++){
    createTableCell(newRow,data[i]);
  }
}
```

Hierbei setzen wir voraus, dass die Variable `dataTable` ein Verweis auf eine bestehende Tabelle ist, und dass sich die Funktionen `createTableRow()` und `createTableCell()` für uns um die Einzelheiten der DOM-Bearbeitung kümmern. Interessant ist

hier die Berechnungsphase, die in einer praktischen Anwendung Hunderte von Codezeilen umfassen kann. Wir weisen dem `button`-Element wie folgt einen Ereignishandler zu:

```
buttonDiv.onclick=buttonOnclickHandler;
```

Mehrere Ereignistypen unterstützen

Nehmen wir an, wir haben unsere Anwendung mit Ajax ausgestattet. Wir fragen den Server nach Aktualisierungen ab und möchten die Berechnung durchführen, wenn ein bestimmter Wert vom Server aktualisiert wird, und mit den Daten eine andere Tabelle aktualisieren. Um die Einzelheiten für die wiederholte Abfrage des Servers müssen wir uns hier nicht kümmern. Gehen wir davon aus, dass wir dafür einen Verweis auf ein Abrufobjekt namens `poller` haben. Intern verwendet es ein XMLHttpRequest-Objekt und hat seinen `onreadystatechange`-Handler so eingerichtet, dass ein Aufruf für eine `onload`-Funktion erfolgt, sobald es damit fertig ist, eine Aktualisierung vom Server zu laden. Wir können die Berechnungs- und Anzeigephasen wie folgt in Hilfsfunktionen abstrahieren:

```
function buttonOnclickHandler(event){
  var data=calculate();
  showData(dataTable,data);
}
function ajaxOnloadHandler(){
  var data=calculate();
  showData(otherDataTable,data);
}
function calculate(){
  var data=new Array();
  data[0]=6;
  data[1]=data[0]/3;
  data[2]=data[0]*data[1]+7;
  return data;
}
function showData(table,data){
  var newRow=createTableRow(table);
  for (var i=0;i<data.length;i++){
    createTableCell(newRow,data[i]);
  }
}
buttonDiv.onclick=buttonOnclickHandler;
poller.onload=ajaxOnloadHandler;
```

Ein Großteil der gemeinsamen Funktionalität wurde in die Funktionen `calculate()` und `showData()` abstrahiert. Nur in den `onclick`- und `onload`-Handlern wiederholen wir uns ein bisschen.

Wir haben eine sehr viel bessere Trennung zwischen der Geschäftslogik und der Aktualisierung der Benutzeroberfläche erreicht. Auch dies ist eine nützliche, wieder-

verwendbare Lösung, nämlich das Muster Befehl. Das Befehlsobjekt definiert einige Aktivitäten beliebiger Komplexität und kann im Code leicht übergeben und zwischen den Elementen der Benutzeroberfläche ausgetauscht werden. Beim klassischen Befehl-Muster für objektorientierte Sprachen werden die Benutzerobjekte als Befehlsobjekte zusammengefasst, die gewöhnlich von einer Basisklasse oder einem Interface abgeleitet sind. Wir haben dasselbe Problem hier auch auf eine leicht abgewandelte Art und Weise gelöst. Da JavaScript-Funktionen Objekte erster Klasse sind, können wir sie direkt als Befehlsobjekte behandelt und dabei denselben Grad der Abstraktion beibehalten.

Alles, was der Benutzer tut, als Befehl zusammenzufassen, mag mühselig erscheinen, weist aber auch einen verborgenen Vorteil auf. Wenn alle Benutzeraktionen in Befehlsobjekten zusammengefasst sind, können wir andere Standardfunktionalitäten leicht mit ihnen verknüpfen. Die am häufigsten erörterte Erweiterung ist die Ergänzung um eine `undo()`-Methode. Dadurch wird die Grundlage für eine allgemeine Rückgängig-Funktion in der gesamten Anwendung gelegt. In komplexeren Beispielen sollten Befehle bei der Ausführung in einem Stapel aufgezeichnet werden. Der Benutzer kann dann eine Rückgängig-Schaltfläche verwenden, um sich durch den Stapel rückwärts zu bewegen und die Anwendung wieder in einen früheren Zustand zu versetzen (siehe Abbildung 3.3).

Jeder neue Befehl wird oben auf dem Stapel platziert, der sich elementweise rückgängig machen lässt. Der Benutzer erstellt ein Dokument durch eine Abfolge von Schreibvorgängen. Danach markiert er das gesamte Dokument und drückt versehentlich auf die Löschtaste. Wenn er die `undo`-Funktion aufruft, wird das oberste Element vom Stapel entfernt. Dessen `undo()`-Methode wird aufgerufen und bringt den gelöschten Text zurück. Durch weiteres Rückgängigmachen wird die Markierung des Textes aufgehoben usw.

Die Verwendung von Befehl zum Erstellen eines Rückgängig-Stapels bedeutet für den Entwickler natürlich zusätzliche Arbeit, da er sicherstellen muss, dass die Kombination aus Ausführen und Rückgängigmachen von Befehlen das System in den ursprünglichen Zustand zurückversetzt. Eine funktionierende Rückgängig-Funktion kann jedoch einen deutlichen Unterschied zwischen zwei Produkten ausmachen, vor allem bei Anwendungen, die intensiv oder langfristig eingesetzt werden. Wie wir in Kapitel 1 gesehen haben, ist das genau der Bereich, den Ajax zu erobern beginnt.

Befehlsobjekte sind auch nützlich, wenn wir in einer Anwendung Informationen über die Grenzen der Subsysteme hinweg übergeben müssen. Das Netzwerk ist natürlich eine solche Grenze, und wir werden in Kapitel 5 bei der Erörterung von Client/Server-Interaktionen auf das Muster Befehl zurückkommen.

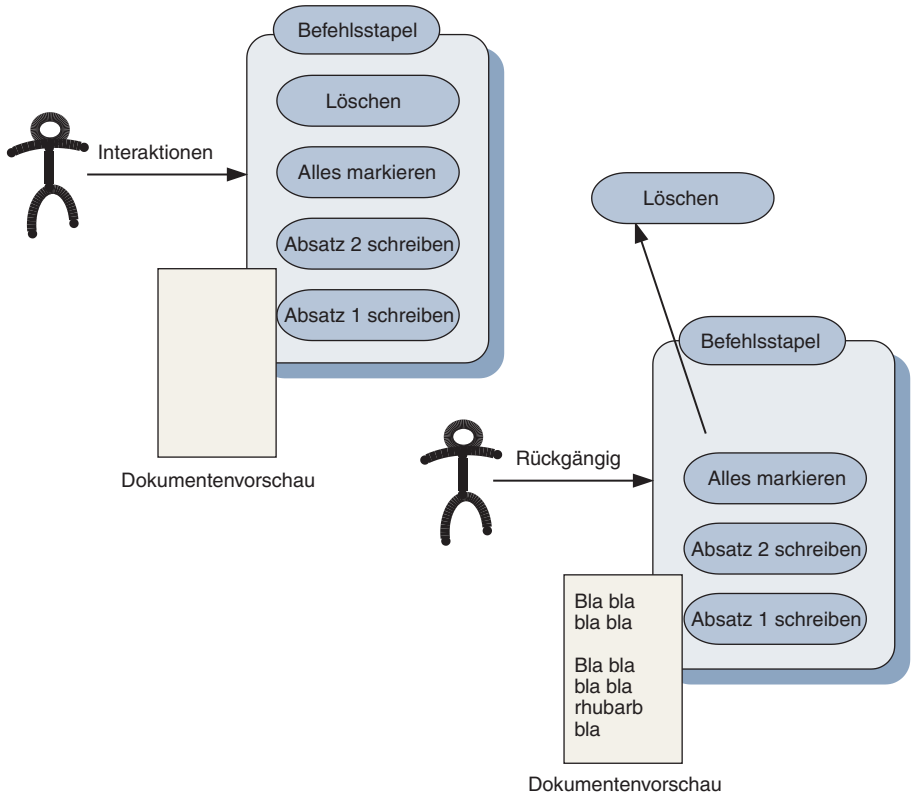

Abbildung 3.3: Das Muster Befehl zur Implementierung eines allgemeinen Rückgängig-Stapels in einer Textverarbeitung. Alle Benutzerinteraktionen werden als Befehle dargestellt, die genauso rückgängig gemacht wie ausgeführt werden können.

3.2.4 Nur jeweils ein Verweis auf eine Ressource: Das Muster Singleton

In manchen Situationen ist es wichtig sicherzustellen, dass es nur einen einzigen Kontaktpunkt mit einer bestimmten Ressource gibt. Auch dies lässt sich am besten an einem konkreten Beispiel erklären.

Ein einfaches Beispiel von der Börse

Stellen wir uns eine Ajax-Anwendung vor, die Börsendaten bearbeitet, sodass wir auf den Märkten Handel treiben, Was-wäre-wenn-Berechnungen durchführen und über ein Netzwerk Simulationsspiele gegen andere Benutzer austragen können. Für diese Anwendung definieren wir drei Modi, die nach dem Ampelsystem benannt sind. Im Echtzeitmodus (*green*) können wir Aktien auf Onlinemärkten kaufen und verkaufen,

wenn sie geöffnet sind, und Was-wäre-wenn-Berechnungen anhand gespeicherter Datenbanken ausführen. Nach Marktschluss gehen wir zum reinen Analysemodus über (*red*), in dem wir immer noch Was-wäre-wenn-Analysen durchführen, aber nicht mehr mit Aktien handeln können. Im Simulationsmodus (*amber*) können wir alle Aktionen durchführen, die im grünen Modus zur Verfügung stehen, handeln dabei aber nicht an einer echten Börse, sondern nur mit einem Testdatensatz.

Unser Clientcode stellt diese Möglichkeiten wie folgt als JavaScript-Objekt dar:

```
var MODE_RED=1;
var MODE_AMBER=2;
var MODE_GREEN=3;
function TradingMode(){
   this.mode=MODE_RED;
}
```

Wir können den durch dieses Objekt repräsentierten Modus abfragen und setzen und werden das in unserem Code auch an vielen Stellen tun. Außerdem können wir die Methoden `getMode()` und `setMode()` bereitstellen, die Bedingungen wie z.B. die überprüfen, dass die Märkte geöffnet sind, aber vorerst wollen wir das Beispiel einfach halten.

Nehmen wir an, dass der Benutzer unter anderem die beiden Optionen hat, Aktien zu kaufen und zu verkaufen sowie die potenziellen Gewinne und Verluste einer Transaktion zu berechnen, bevor er sie ausführt. Je nach Operationsmodus zeigen die Kaufs- und Verkaufsaktionen auf unterschiedliche Webdienste – interne im Bernstein-Modus *(amber)*, den Server des Aktienmaklers im grünen Modus *(green)* – oder werden im roten Modus *(red)* ganz ausgeschaltet. Ebenso gründet sich die Analyse auf den Abruf von Daten-Feeds aktueller und vergangener Preise – simuliert im Bernstein-Modus und an echten Onlinebörsendaten ausgeführt im grünen Modus. Um zu wissen, auf welchen Feed sie zeigen sollen, müssen beide Optionen auf ein im Folgenden definiertes TradingMode-Objekt verweisen (siehe Abbildung 3.4).

Es ist unabdingbar, dass beide Aktivitäten auf dasselbe TradingMode-Objekt zeigen. Wenn ein Benutzer an einer simulierten Börse kauft und verkauft, seine Entscheidungen aber auf die Analyse von echten Daten stützt, wird er das Spiel wahrscheinlich verlieren. Kauft und verkauft er echte Aktien aufgrund der Analyse einer Simulation, verliert er unter Umständen sogar seinen Job.

Ein Objekt, von dem es nur eine einzige Instanz gibt, wird manchmal als *Singleton* bezeichnet. Wir schauen uns zunächst an, wie Singletons in objektorientierten Sprachen gehandhabt werden, und arbeiten dann eine Strategie für ihren Einsatz in JavaScript aus.

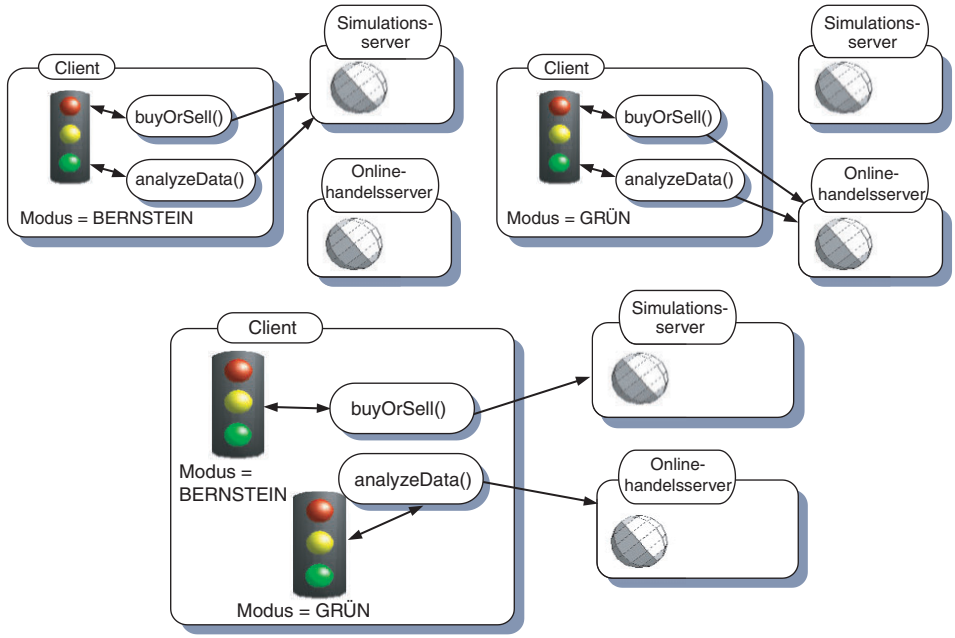

Abbildung 3.4: In unserem Beispiel einer Ajax-Börsenanwendung bestimmen sowohl die Kaufs- und Verkaufs- als auch die Analysefunktionen aufgrund des Status des Trading-Mode-Objekts, ob sie echte oder simulierte Daten verwenden sollen. Im Bernstein-Modus kommunizieren sie mit dem Simulationsserver, im grünen Modus mit dem Onlinehandelsserver. Wenn es im System mehr als ein TradingMode-Objekt gibt, kann es in einen inkonsistenten Zustand geraten.

Singletons in Java

In Java-ähnlichen Sprachen werden Singletons gewöhnlich dadurch implementiert, dass der Objektkonstruktor verborgen und eine GET-Methode bereitstellt wird, wie Listing 3.3 veranschaulicht.

Listing 3.3: Das Singleton-Objekt TradingMode in Java

```
public class TradingMode{
  private static TradingMode instance=null;
  public int mode;
  private TradingMode(){
    mode=MODE_RED;
  }
  public static TradingMode getInstance(){
    if (instance==null){
       instance=new TradingMode();
    }
    return instance;
```

```
  }
  public void setMode(int mode){
    ...
  }
}
```

Die Lösung für Java nutzt die Zugriffsmodifizierer `private` und `public`, um das Singleton-Verhalten durchzusetzen. Der folgende Code wird nicht kompiliert, da der Konstruktor nicht öffentlich zugänglich ist:

```
new TradingMode().setMode(MODE_AMBER);
```

Dagegen wird der folgende Code kompiliert:

```
TradingMode.getInstance().setMode(MODE_AMBER);
```

Dieser Code stellt sicher, dass jeder Aufruf an dasselbe TradingMode-Objekt weitergeleitet wird. Wir haben hierbei verschiedene Merkmale der Sprache genutzt, die in JavaScript nicht zur Verfügung stehen, weshalb wir uns ansehen müssen, wie wir auch ohne sie auskommen.

Singletons in JavaScript

In JavaScript gibt es keine integrierte Unterstützung für Zugriffsmodifizierer, aber wir können den Konstruktor dadurch »verbergen«, dass wir gar keinen bereitstellen. JavaScript gründet sich auf Prototypen, wobei Konstruktoren gewöhnliche Function-Objekte sind (wenn Ihnen das unklar ist, schlagen Sie in Anhang B nach). Wir können ein TradingMode-Objekt auf die gewöhnliche Weise schreiben:

```
function TradingMode(){
   this.mode=MODE_RED;
}
TradingMode.prototype.setMode=function(){
}
```

Dazu können wir dann eine globale Variable als Pseudo-Singleton bereitstellen:

```
TradingMode.instance=new TradingMode();
```

Das kann aber nicht verhindern, dass schlechter Code den Konstruktor aufruft. Wir können das Objekt jedoch auch vollständig manuell ohne Prototyp konstruieren:

```
var TradingMode=new Object();
TradingMode.mode=MODE_RED;
TradingMode.setMode=function(){
   ...
}
```

Dies lässt sich wie folgt auch etwas knapper definieren:

```
var TradingMode={
   mode:MODE_RED,
   setMode:function(){
      ...
   }
};
```

Beide Beispiele erstellen das gleiche Objekt. Die erste Schreibweise ist für Java- und C#-Programmierer wahrscheinlich vertrauter. Den zweiten Ansatz haben wir ebenfalls vorgestellt, da er häufig in der Bibliothek Prototype und in den davon abgeleiteten Frameworks verwendet wird.

Diese Lösung funktioniert nur in den Grenzen eines einzigen Skriptkontexts. Wenn das Skript in einen eigenen Iframe geladen wird, startet es seine eigene Kopie des Singletons. Dies können wir ändern, indem wir ausdrücklich verlangen, dass auf das Singleton-Objekt vom Dokument der obersten Ebene aus zugegriffen werden soll (in JavaScript ist `top` stets der Verweis auf dieses Dokument), wie Listing 3.4 veranschaulicht.

Listing 3.4: Das Singleton-Objekt TradingMode in JavaScript

```
Function getTradingMode(){
  if (!top.TradingMode){
    top.TradingMode=new Object();
    top.TradingMode.mode=MODE_RED;
    top.TradingMode.setMode=function(){
      ...
    }
  }
  return top.TradingMode;
}
```

Dadurch ist es möglich, das Skript sicher in mehrere Iframes einzuschließen und gleichzeitig die Eindeutigkeit des Singleton-Objekts zu bewahren. (Wenn Sie ein Singleton über mehrere Fenster der obersten Ebene hinweg verwenden möchten, müssen Sie `top.opener` verwenden. Aufgrund des beschränkten Platzes überlassen wir diese Aufgabe dem Leser als Übung.)

Wenn Sie Code für eine Benutzeroberfläche schreiben, haben Sie wahrscheinlich keinen großen Bedarf für Singletons, aber sie können bei der Modellierung der Geschäftslogik in JavaScript äußerst nützlich sein. Bei herkömmlichen Webanwendungen wird die Geschäftslogik gewöhnlich nur auf dem Server modelliert, aber durch Ajax ändert sich das, sodass es hilfreich sein kann, über Singletons Bescheid zu wissen.

Damit haben Sie einen ersten Eindruck davon gewonnen, was Refactoring in der Praxis bedeutet. Die Beispiele, die wir uns bis jetzt angeschaut haben, sind alle ziemlich einfach gewesen, aber selbst dabei hat uns das Refactoring geholfen, den Code klarer zu machen und mehrere Schwachpunkte auszumerzen, die uns ansonsten Probleme bereitet hätten, wenn die Anwendung größer wird.

Dabei haben wir einige wenige Entwurfsmuster kennen gelernt. Im folgenden Abschnitt schauen wir uns ein großmaßstäbliches, serverseitiges Muster an, um herauszufinden, wie wir damit verworrenen Code einem Refactoring unterwerfen können, sodass er klarer und flexibler wird.

3.3 Modell-Präsentation-Steuerung

Die kleinen Muster, die wir uns bis jetzt angesehen haben, können für bestimmte Kodieraufgaben sehr nützlich sein. Es sind jedoch auch Muster für die Gliederung ganzer Anwendungen entwickelt worden, die manchmal als Architekturmuster bezeichnet werden. In diesem Abschnitt schauen wir uns ein Architekturmuster an, mit dessen Hilfe wir unsere Ajax-Projekte auf verschiedene Art und Weise gliedern können, sodass sie einfacher zu programmieren und einfacher zu warten sind.

Modell-Präsentation-Steuerung (MPS, engl. Model-View-Controller, MVC) ist eine Möglichkeit, um eine gute Trennung zwischen dem Teil des Programms zu beschreiben, der mit einem Benutzer interagiert, und dem Teil, der die Schwerarbeit, die Zahlenverarbeitung und andere »geschäftliche« Aufgaben der Anwendung durchführt.

MPS wird gewöhnlich in großem Maßstab angewendet und deckt ganze Ebenen einer Anwendung ab oder erstreckt sich gar über mehrere Ebenen. In diesem Kapitel führen wir das Muster ein und zeigen, wie Sie es auf einen Webserver anwenden können, um einer Ajax-Anwendung Daten bereitzustellen. In Kapitel 4 betrachten wir den fortgeschritteneren Fall des Einsatzes für eine JavaScript-Clientanwendung.

Das MPS-Muster bezeichnet drei Rollen, die eine Komponente in dem System ausfüllen kann. Das Modell ist die Darstellung des Aufgabenbereichs der Anwendung, also dessen, mit dem die Anwendung umgehen soll. Eine Textbearbeitung modelliert ein Dokument, eine Kartenanwendung modelliert Punkte in einem Raster, Höhenlinien usw.

Die Präsentation ist der Teil des Programms, der die Dinge für den Benutzer darstellt – Eingabeformulare, Bilder, Text oder Komponenten. Diese Präsentation muss nicht grafisch sein. In einem sprachgesteuerten Programm stellen die gesprochenen Eingabeaufforderungen z. B. die Präsentation dar.

Die goldene Regel von MPS lautet, dass die Präsentation und das Modell nicht miteinander kommunizieren sollten. Zunächst mag das nach einem funktionsunfähigen Programm klingen, aber dies ist der Punkt, an dem die Steuerung ins Spiel kommt. Wenn der Benutzer auf eine Schaltfläche klickt oder ein Formular ausfüllt, teilt die Präsentation dies der Steuerung mit. Die Steuerung bearbeitet dann das Modell und

entscheidet, ob die Änderungen am Modell eine Aktualisierung der Präsentation erfordern. Wenn ja, teilt die Steuerung der Präsentation mit, wie sie sich zu ändern an (siehe Abbildung 3.5).

Der Vorteil liegt darin, dass das Model und die Präsentation lose gekoppelt bleiben, was bedeutet, dass das eine kaum etwas vom anderen weiß. Offensichtlich müssen sie genug wissen, um ihre Aufgabe zu erledigen, aber die Präsentation hat nur sehr allgemeine Kenntnisse über das Modell.

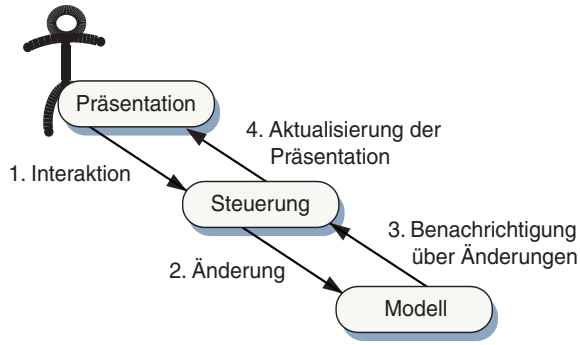

Abbildung 3.5: Die Hauptkomponenten des Musters Modell-Präsentation-Steuerung. Präsentation und Modell interagieren nicht direkt miteinander, sondern stets über die Steuerung. Die Steuerung können Sie sich als dünne Grenzschicht vorstellen, die eine Kommunikation zwischen Modell und Präsentation erlaubt, aber eine klare Trennung des Codes erzwingt, um die Flexibilität und Wartungsfreundlichkeit des Codes im Laufe der Zeit zu verbessern.

Betrachten wir ein Programm zur Verwaltung von Inventarlisten. Die Steuerung kann der Präsentation eine Funktion zur Verfügung stellen, die eine Liste aller Produkte mit einer bestimmten Kategorie-ID zurückgibt, aber die Präsentation weiß nichts darüber, wie diese Liste erstellt wird. Es kann sein, dass Version 1 des Programms die Daten, die zum Aufbau der Liste benötigt werden, in einem Arrays im Arbeitsspeicher ablegt oder aus einer linearen Textdatei liest. Bei der zweiten Version des Programms kann ein Bedarf für die Handhabung sehr viel größerer Datensätze entstanden sein, sodass der Architektur ein relationaler Datenbankserver hinzugefügt wird. Die Auswirkungen dieser Änderungen auf das Modell sind erheblich, weshalb viel Code umgeschrieben werden muss. Solange die Steuerung aber immer noch eine Liste der Produkte einer bestimmten Kategorie ausliefern kann, ist der Einfluss auf den Code der Präsentation gleich null.

Ebenso können die Ingenieure, die an der Präsentation arbeiten, die Nutzbarkeit der Anwendung in völliger Freiheit verbessern, ohne sich darüber Sorgen machen zu müssen, dass sie versteckte Annahmen im Modell hinfällig machen, solange sie sich nur an die grundlegenden Übereinkünfte über die Schnittstellen halten, die die Steuerung ihnen zur Verfügung stellt. Durch die Aufteilung des Systems in Subsysteme

bietet MPS eine Versicherung dagegen, dass kleinere Änderungen im gesamten Code Wellen schlagen. Dadurch können die Teams der einzelnen Subsysteme schnell auf Probleme reagieren, ohne Angst zu haben, anderen auf die Füße zu treten.

Das MPS-Muster wird auf Frameworks klassischer Webanwendungen gewöhnlich auf eine ganz bestimmte Weise angewendet, um die Abfolge statischer Seiten bereitzustellen, aus denen sich die Oberfläche zusammensetzt. Wenn eine Ajax-Anwendung läuft und Daten vom Server anfordert, ähneln die Mechanismen zur Bereitstellung der Daten denen einer klassischen Webanwendung. MPS im Webserver-Stil ist auch für Ajax-Anwendungen von Vorteil, und da diese Variante so gut bekannt ist, beginnen wir hier damit und kümmern uns um Ajax-spezifische Möglichkeiten für den Einsatz von MPS später.

Wenn Ihnen Webframeworks neu sind, sollte dieser Abschnitt Ihnen ausreichend Informationen liefern, um zu verstehen, wie sie dabei helfen können, eine Ajax-Anwendung skalierbarer und stabiler zu machen. Wenn Sie dagegen bereits mit Webschicht-Tools wie Template-Engines und ORM-Werkzeugen (Object-Relational Mapping) oder mit Frameworks wie Struts, Spring oder Tapestry vertraut sind, dürften Ihnen diese Ausführungen zum größten Teil bekannt vorkommen. In diesem Fall können Sie diesen Abschnitt einfach überspringen und in Kapitel 4 fortfahren, wo wir die Verwendung von MPS auf eine andere Art und Weise vorführen.

3.4 Webserver-MPS

Webanwendungen sind MPS nicht fremd, selbst die klassische, auf Seiten aufgebaute Variante, auf die wir in diesem Buch so heftig einschlagen. Die Natur einer Webanwendung an sich erzwingt eine gewisse Trennung zwischen Präsentation und Modell, da sie sich auf unterschiedlichen Computern befinden. Folgt eine Webanwendung daher inhärent dem MPS-Muster? Oder ist es, anders ausgedrückt, möglich, eine Webanwendung zu schreiben, die Modell und Präsentation vermischt?

Leider ist das möglich. Es ist ganz einfach, und die meisten Webentwickler haben dies irgendwann auch schon einmal gemacht, die Autoren eingeschlossen.

Die meisten Befürworter von MPS im Web behandeln die HTML-Seite und den Code, der sie generiert, als Präsentation, und nicht das, was der Benutzer tatsächlich sieht, wenn die Seite dargestellt wird. Im Falle einer Ajax-Anwendung, die Daten für einen JavaScript-Client bereitstellt, ist die Präsentation nach dieser Sichtweise das XML-Dokument, das dem Client in der HTTP-Antwort zurückgegeben wird. Dieses Dokument von der Geschäftslogik zu trennen erfordert daher ein wenig Disziplin.

3.4.1 Die Ajax-Webserverschicht ohne Muster

Lassen Sie uns als Beispiel eine Webserverschicht für eine Ajax-Anwendung entwickeln, um unsere Erklärungen zu veranschaulichen. Wir haben in Kapitel 2 und in Abschnitt 3.1.4 bereits die Grundlagen des clientseitigen Ajax-Codes gesehen und

werden in Kapitel 4 wieder darauf zurückkommen. Jetzt aber konzentrieren wir uns auf das, was auf dem Webserver vor sich geht. Wir beginnen damit, dies auf die einfachste Art und Weise zu kodieren, die möglich ist, und nehmen dann nach und nach Refactorings zum MPS-Muster hin vor, um zu sehen, welche Vorteile dies für unsere Anwendung im Hinblick darauf bringt, sie an Änderungen anpassen zu können. Als Erstes stellen wir die Anwendung vor.

In einem Onlineshop für Bekleidung unterhalten wir eine Liste von Kleidungsstücken, die in einer Datenbank gespeichert ist. Wir möchten diese Datenbank abfragen und die Liste der Artikel dem Benutzer anzeigen, wobei ein Bild, Titel, eine kurze Beschreibung und ein Preis dargestellt werden sollen. Wenn ein Artikel in mehreren Größen und Farben vorhanden ist, soll eine Auswahlmöglichkeit dafür vorhanden sein. Abbildung 3.6 zeigt die Hauptkomponenten des Systems, nämlich die Datenbank, eine Datenstruktur, die für ein einzelnes Produkt steht, und ein XML-Dokument, das an den Ajax-Client übermittelt wird und alle Produkte aufführt, die mit den Suchkriterien übereinstimmen.

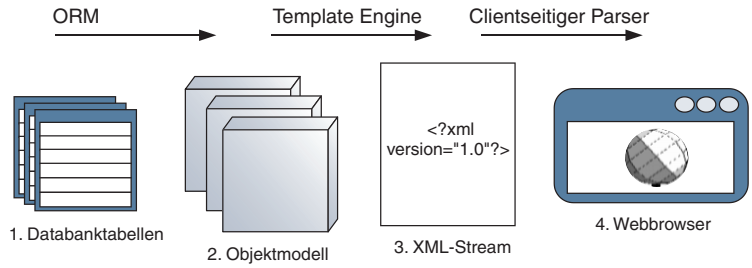

Abbildung 3.6: Die Hauptkomponenten zum Erstellen eines XML-Feeds aus Produktdaten in unserem Beispiel eines Onlineshops. Um die Sicht zu erstellen, extrahieren wir einen Satz von Ergebnissen aus der Datenbank, füllen damit Datenstrukturen, die für die einzelnen Kleidungsstücke stehen, und übertragen diese Daten als XML-Stream an den Client.

Stellen Sie sich vor, dass der Benutzer gerade den Shop betreten hat und die Auswahl zwischen Herren-, Damen- und Kinderbekleidung sieht. Durch die Spalte `Category` der Datenbanktabelle `Garments` wird jedes Produkt einer dieser Kategorien zugeordnet. Alle entsprechenden Artikel für eine Suche unter Herrenbekleidung (»Menswear«) können wie folgt mit ein wenig einfachem SQL-Code abgerufen werden:

```
SELECT * FROM garments WHERE CATEGORY = 'Menswear';
```

Wir müssen die Ergebnisse dieser Abfrage abrufen und dann in Form von XML an die Ajax-Anwendung senden. Im nächsten Abschnitt werden wir sehen, wie das gemacht wird.

XML-Daten für den Client erzeugen

Listing 3.5 zeigt eine einfache, unsaubere Lösung dafür. In diesem Beispiel wird PHP mit einer MySQL-Datenbank verwendet; wichtig ist aber die allgemeine Struktur. Auch eine ASP- oder JSP-Seite oder ein Ruby-Skript kann ähnlich konstruiert werden.

Listing 3.5: Einfache, unsaubere Erzeugung eines XML-Streams aus einer Datenbankabfrage

```
<?php
header("Content-type: application/xml");  ← Teilt dem Client mit, dass wir XML zurückgeben
echo "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n";
$db=mysql_connect("my_db_server","mysql_user");
mysql_select_db("mydb",$db);
$sql="SELECT id,title,description,price,colors,sizes"
  ."FROM garments WHERE category=\"{$cat}\"";
$result=mysql_query($sql,$db);
echo "<garments>\n";
while ($myrow = mysql_fetch_row($result)) {    ← Iteriert durch den Ergebnissatz
    printf("<garment id=\"%s\" title=\"%s\">\n"
      ."<description>%s</description>\n<price>%s</price>\n",
      $myrow["id"],
      $myrow["title"],
      $myrow["description"],
      $myrow["price"]);
    if (!is_null($myrow["colors"])){
      echo "<colors>{$myrow['colors']}</colors>\n";
    }
    if (!is_null($myrow["sizes"])){
      echo "<sizes>{$myrow['sizes']}</sizes>\n";
    }
    echo "</garment>\n";
}
echo "</garments>\n";
?>
```

Ruft Ergebnisse von der Datenbank ab

Das PHP-Skript in Listing 3.5 erstellt eine XML-Seite für uns, die in etwa wie in Listing 3.6 aussieht, wenn es zwei passende Produkte in der Datenbank gibt. Zur besseren Lesbarkeit wurden Zeilen eingerückt. Wir haben XML als Kommunikationsmedium zwischen Client und Server gewählt, da es gewöhnlich für diesen Zweck eingesetzt wird und da wir in Kapitel 2 bereits gesehen haben, wie ein vom Server erstelltes XML-Dokument mithilfe des XMLHttpRequest-Objekts verarbeitet wird. In Kapitel 5 schauen wir uns die verschiedenen anderen Möglichkeiten im Einzelnen an.

Listing 3.6: XML-Beispielausgabe von Listing 3.5

```
<garments>
  <garment id="SCK001" title="Golfers' Socks">
    <description>Garish diamond patterned socks. Real wool.
```

```
      Real itchy.</description>
    <price>$5.99</price>
    <colors>heather combo,hawaiian medley,wild turkey</colors>
  </garment>
  <garment id="HAT056" title="Deerstalker Cap">
    <description>Complete with big flappy bits.
    As worn by the great detective Sherlock Holmes.
    Pipe is model's own.</description>
    <price>$79.99</price>
    <sizes>S, M, L, XL, egghead</sizes>
  </garment>
</garments>
```

Damit haben wir eine Art Webserveranwendung, vorausgesetzt, es gibt ein freundliches Ajax-Front-End, das unser XML-Dokument verarbeitet. Werfen wir einen Blick in die Zukunft. Stellen Sie sich vor, dass wir unsere Produktpalette erweitern und Unterkategorien hinzufügen möchten (z.B. Smart, Casual, Outdoor). Außerdem möchten wir eine Funktion zur »Suche nach Jahreszeiten« anbieten, z.B. in Form einer Schlüsselwortsuche, sowie einen Link zum Bezahlen. Alle diese Funktionen können in einem ähnlichen XML-Stream bereitgestellt werden. Schauen wir uns an, wie wir den bisherigen Code für diese Zwecke wiederverwenden können und welche Hindernisse sich uns dabei stellen.

Schwierigkeiten bei der Wiederverwertung

Es gibt verschiedene Probleme bei der Wiederverwendung unseres bisherigen Skripts. Wenn wir erneut nach der Kategorie oder nach einem Schlüsselwort suchen möchten, müssen wir die Erzeugung des SQL-Codes ändern. Das kann letzten Endes zu einer hässlichen Anhäufung von `if`-Anweisungen führen, die sich mit der Zeit ansammeln, während wir immer mehr Suchoptionen hinzufügen, sowie zu einer wachsenden Liste von optionalen Suchparametern.

Es gibt eine Alternative, die sogar noch schlechter ist, nämlich einfach eine frei gestaltete `WHERE`-Klausel in den CGI-Parametern zu akzeptieren:

```
$sql="SELECT id,title,descripton,price,colors,sizes"
   ."FROM garments" WHERE ".$sqlWhere;
```

Dies können wir direkt aus dem URL heraus aufrufen, z.B. wie folgt:

```
garments.php?sqlWhere=CATEGORY="Menswear"
```

Diese Lösung bringt Modell und Präsentation sogar noch stärker durcheinander und führt rohen SQL-Code in den Darstellungscode ein. Außerdem werden hierdurch Tür und Tor für SQL-Injection-Angriffe geöffnet. Obwohl moderne Versionen von PHP integrierte Verteidigungseinrichtungen dagegen haben, wäre es unsinnig, sich darauf zu verlassen.

Zweitens haben wir das XML-Datenformat in die Seite fest eingefügt – es liegt dort irgendwo zwischen den `printf()`- und `echo`-Anweisungen vergraben. Es kann aber verschiedene Gründe geben, aus denen wir das Datenformat ändern möchten. Vielleicht möchten wir neben dem Verkaufspreis noch den ursprünglichen Preis anzeigen, um irgendeinen armen Zeitgenossen dazu zu überreden, sämtliche der von uns eingekauften kratzigen Golfsocken zu kaufen.

Drittens wird der Ergebnissatz der Datenbank selbst genutzt, um das XML-Dokument zu erstellen. Das mag zu Anfang sehr effizient wirken, weist aber zwei potenzielle Probleme auf. Wir halten die Datenverbindung die ganze Zeit über offen, die wir zum Erstellen des XML-Dokuments brauchen. In diesem Fall tun wir während der `while()`-Schleife zwar nichts übermäßig Schwieriges, sodass die Verbindung nicht zu lange besteht, aber letzten Endes kann sich dies als Engpass erweisen. Außerdem funktioniert dies nur, wenn wir unsere Datenbank als lineare Datenstruktur behandeln.

3.4.2 Refactoring des Modells für den Geschäftsbereich

Zurzeit handhaben wir unsere Liste von Farben und Größen noch auf ziemlich ineffiziente Art und Weise, indem wir kommagetrennte Listen in Feldern der Tabelle `Garments` speichern. Wenn wir unsere Daten normalisieren, um ein gutes relationales Modell aufzubauen, erhalten wir eine eigene Tabelle aller verfügbaren Farben und eine Brückentabelle, die die Kleidungsstücke mit den Farben verknüpft (was Datenbankspezialisten eine m:n-Beziehung nennen). Abbildung 3.7 zeigt die Verwendung einer m:n-Beziehung dieser Art.

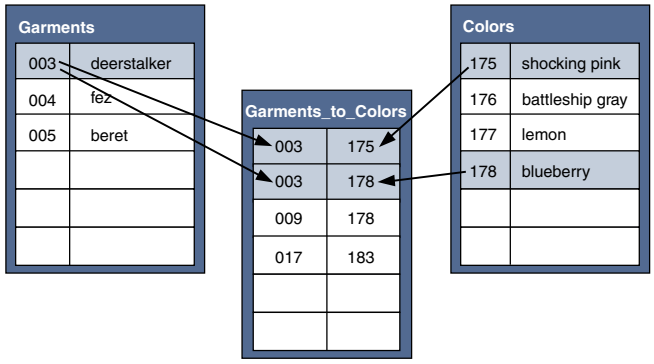

Abbildung 3.7: Eine m:n-Beziehung in einem Datenbankmodell. Die Tabelle Colors führt alle verfügbaren Farben für alle Kleidungstücke auf, während die Tabelle Garments keine Informationen über Farben mehr enthält.

Um die verfügbaren Farben für den Deerstalker-Hut zu ermitteln, schlagen wir in der Tabelle `Garments_to_Colors` unter dem Fremdschlüssel `garment_id` nach. Wenn wir den Inhalt der Spalte `color_id` zum Primärschlüssel in der Tabelle `Colors` zurückverfolgen, können wir erkennen, dass der Hut in schreiendem Rosa und in Blaubeere erhältlich

ist, aber nicht in Schlachtschiffgrau. Wenn wir die Abfrage andersherum ausführen, können wir die Tabelle `Garments_to_Colors` auch verwenden, um alle Kleidungsstücke aufzulisten, die es in einer bestimmten Farbe gibt.

Damit nutzen wir unsere Datenbank schon besser, aber der erforderliche SQL-Code zum Abrufen aller Informationen wird langsam ein bisschen kompliziert. Anstatt ausgefeilte Verknüpfungsabfragen per Hand zu konstruieren, wäre es praktisch, wenn wir unsere Kleidungsstücke als Objekte behandeln könnten, die ein Array mit Farben und Größen enthalten.

Werkzeuge für die objektrelationale Zuordnung

Zum Glück gibt es Werkzeuge und Bibliotheken, die das für uns tun können, nämlich die so genannten ORM-Tools (Object-Relational Mapping, objektrelationale Zuordnung). Eine ORM führt automatisch eine Übersetzung zwischen Datenbankdaten und Objekten im Speicher durch und nimmt dem Entwickler die Arbeit ab, rohen SQL-Code zu schreiben. PHP-Programmierer können dazu auf `PEAR::DB_DataObject`, Easy PHP Data Objects (EZPDO) oder Metastorage zurückgreifen. Java-Programmierer sind mit Möglichkeiten verwöhnt, wobei Hibernate (auch auf die .NET-Plattform portiert) zurzeit sehr populär ist. ORM-Werkzeuge sind ein umfangreiches Thema, das wir vorerst hintanstellen.

Wenn wir uns unsere Anwendung im Sinne von MPS ansehen, können wir erkennen, dass der Einsatz von ORM eine schöne Nebenwirkung hat, denn damit haben wir die Anfänge eines echten Modells zur Hand. Wir können jetzt unsere XML-Generator-Routine für die Kommunikation mit dem Garment-Objekt schreiben und der ORM den Umgang mit der Datenbank überlassen. Damit sind wir auch nicht mehr an die API (und die Eigenheiten) einer bestimmten Datenbank gebunden. Listing 3.7 zeigt den geänderten Code nach dem Wechsel zu ORM.

In diesem Fall definieren wir das Geschäftsobjekt (also das Modell) für unser Beispiel in PHP unter Verwendung von `PEAR::DB_DataObject`, wozu unsere Klassen eine `DB_DataObject`-Basisklasse erweitern müssen. Verschiedene ORMs gehen hierbei unterschiedlich vor. Wichtig ist jedoch, dass wir einen Satz von Objekten erstellen, mit denen wir wie mit normalem Code kommunizieren können, während wir die Komplexität von SQL-Anweisungen herausabstrahieren.

Listing 3.7: Objektmodell für das Online-Bekleidungsgeschäft

```
require_once "DB/DataObject.php";
class GarmentColor extends DB_DataObject {
  var $id;
  var $garment_id;
  var $color_id;
}
class Color extends DB_DataObject {
  var $id;
  var $name;
```

```
}
class Garment extends  DB_DataObject {
  var $id;
  var $title;
  var $description;
  var $price;
  var $colors;
  var $category;
  function getColors(){
    if (!isset($this->colors)){
      $linkObject=new GarmentColor();
      $linkObject->garment_id = $this->id;
      $linkObject->find();
      $colors=array();
      while ($linkObject->fetch()){
        $colorObject=new Color();
        $colorObject->id=$linkObject->color_id;
        $colorObject->find();
        while ($colorObject->fetch()){
          $colors[] = clone($colorObject);
        }
      }
    }
    return $colors;
  }
}
```

Neben dem zentralen Garment-Objekt definieren wir auch ein Color-Objekt und eine Methode für `Garment`, um alle Farben abzurufen, in denen das Kleidungsstück erhältlich ist. Die Größen lassen sich ebenso implementieren, worauf wir hier aber aus Gründen der Kürze verzichtet haben. Da diese Bibliothek m:n-Beziehungen nicht direkt unterstützt, müssen wir einen Objekttyp für die Verknüpfungstabelle definieren und in der Methode `getColors()` dort deren Einträge iterieren. Dennoch ist dies ein ziemlich vollständiges und lesbares Objektmodell. Lassen Sie uns dieses Modell jetzt für unsere Seite anwenden.

Das überarbeitete Modell verwenden

Wir haben aus unserer bereinigten Datenbankstruktur ein Datenmodell erstellt, das wir jetzt innerhalb unseres PHP-Skripts verwenden müssen. Listing 3.8 zeigt die überarbeitete Hauptseite, die die ORM-Objekte verwendet.

Listing 3.8: Überarbeitete Seite mit ORM zur Kommunikation mit der Datenbank

```
<?php
header("Content-type: application/xml");
echo "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n";
include "garment_business_objects.inc"
$garment=new Garment;
```

```
$garment->category = $_GET["cat"];
$number_of_rows = $garment->find();
echo "<garments>\n";
while ($garment->fetch()) {
    printf("<garment id=\"%s\" title=\"%s\">\n"
      ."<description>%s</description>\n<price>%s</price>\n",
      $garment->id,
      $garment->title,
      $garment->description,
      $garment->price);
    $colors=$garment->getColors();
    if (count($colors)>0){
      echo "<colors>\n";
      for($i=0;$i<count($colors);$i++){
        echo "<color>{$colors[$i]}</color>\n";
      }
      echo "</colors>\n";
    }
    echo "</garment>\n";
}
echo "</garments>\n";
?>
```

Wir fügen die Objektmodelldefinitionen ein und kommunizieren dann im Rahmen dieses Objektmodells. Anstatt Ad-hoc-SQL-Code zu konstruieren, erstellen wir ein leeres Garment-Objekt und füllen es teilweise mit unseren Suchkriterien. Da das Objektmodell aus einer anderen Datei eingefügt wird, können wir es auch für andere Suchvorgänge wiederverwenden. Die XML-Präsentation wird nun ebenfalls anhand des Objektmodells erstellt. Unser nächster Refactoring-Schritt besteht darin, das Format des XML-Dokuments von dem Erstellungsvorgang zu trennen.

3.4.3 Inhalt und Darstellung trennen

Unser Präsentationscode ist immer noch mit dem Objekt vermischt, da das XML-Format in den Objektanalysecode eingebunden ist. Wenn wir mehrere Seiten warten, möchten wir das XML-Format nur an einer Stelle ändern, um die Änderungen überall wirksam werden zu lassen. In komplexeren Fällen, in denen wir mehr als ein Format zu warten haben, z.B. eines für kurze und ausführliche Listen, die dem Kunden angezeigt werden, und ein anderes für die Inventur, möchten wir jedes dieser Formate nur einmal definieren und eine zentrale Zuordnung dafür vorsehen.

Template-Systeme

Ein üblicher Ansatz dafür besteht in einer Template-Sprache, also in einem System, das ein Textdokument mit besonderem Markup entgegennimmt, wobei dieses Markup während der Ausführung als Platzhalter für echte Variablen dient. PHP, ASP und JSP sind selbst in gewisser Hinsicht solche Template-Sprachen – geschrieben als

Webseiteninhalt mit eingebettetem Code anstatt als Code mit eingebettetem Inhalt wie in einem Java-Servlet oder einem herkömmlichen CGI-Skript. Dennoch werden durch sie alle Möglichkeiten einer Skriptsprache auf eine Seite angewendet, sodass es sehr einfach ist, Geschäftslogik und Darstellung zu vermischen.

Im Gegensatz dazu bieten zweckgebundene Template-Sprachen wie PHP Smarty und Apache Velocity (ein System auf Java-Grundlage, das als NVelocity auf :NET portiert wurde) eingeschränkte Fähigkeiten für den Code; gewöhnlich begrenzen Sie die Flusssteuerung auf einfache Verzweigungen (z.B. `if`) und Schleifen (z.B. `for` und `while`). Listing 3.9 zeigt ein PHP Smarty-Template zum Erstellen unseres XML-Dokuments.

Listing 3.9: PHP Smarty-Template für die XML-Ausgabe

```
<?xml version="1.0" encoding="UTF-8" ?>
<garments>
{section name=garment loop=$garments}
  <garment id="{$garment.id}" title="{$garment.title}">
    <description>{$garment.description}</description>
    <price>{$garment.price}</price>
{if count($garment.getColors())>0}
    <colors>
{section name=color loop=$garment.getColors()}
      <color>$color->name</color>
{/section}
    </colors>
{/if}
  </garment>
{/section}
</garments>
```

Das Template erwartet die Array-Variable `garment` mit Garment-Objekten als Eingabe. Der Großteil des Templates wird von der Engine Wort für Wort übertragen, aber die Abschnitte innerhalb der geschweiften Klammern werden als Anweisungen interpretiert und entweder durch Variablennamen ersetzt oder als einfache `branch`- oder `loop`-Anweisungen behandelt. Die Struktur des XML-Ausgabedokuments ist im Template besser lesbar als im Rumpf von Listing 3.7, wo sie mit dem Code vermischt war. Schauen wir uns nun an, wie wir das Template für unsere Seite einsetzen.

Die überarbeitete Präsentation verwenden

Wir haben die Definition unseres XML-Formats aus der Hauptseite in unser Smarty-Template verschoben. Daher muss die Hauptseite jetzt nur noch die Template-Engine einrichten und die passenden Daten übergeben. Listing 3.10 zeigt die dafür erforderlichen Änderungen.

Listing 3.10: Smarty zum Erstellen des XML-Ausgabeformats

```
<?php
header("Content-type: application/xml");
include "garment_business_objects.inc";
include "smarty.class.php";
$garment=new DataObjects_Garment;
$garment->category = $_GET["cat"];
$number_of_rows = $garment->find();
$smarty=new Smarty;
$smarty->assign('garments',$garments);
$smarty->display('garments_xml.tpl');
?>
```

Der Einsatz von Smarty erfolgt kurz und knapp in einem Drei-Stufen-Vorgang. Als Erstes erstellen wir eine Smarty-Engine, dann füllen wir sie mit Variablen. In diesem Fall gibt es nur eine, aber wir können so viele hinzufügen, wie wir möchten – so können wir z.B. die Angaben über den Benutzer übergeben, falls sie in der Sitzung gespeichert werden, um über das Template eine personalisierte Begrüßung zu erreichen. Schließlich rufen wir `display()` auf und übergeben den Namen der Template-Datei.

Wir haben jetzt den glücklichen Zustand erreicht, in dem die Präsentation von unserer Seite mit Suchergebnissen getrennt ist. Das XML-Format wird nur einmal definiert und kann mit wenigen Zeilen Code aufgerufen werden. Die Seite mit den Suchergebnissen ist eng gefasst und enthält nur ihre eigenen Informationen, nämlich das Auffüllen der Suchparameter und die Definition des Ausgabeformats. Erinnern Sie sich noch daran, dass wir zuvor davon geträumt haben, im laufenden Betrieb zu anderen XML-Formaten zu wechseln? Mit Smarty ist das einfach: Wir müssen lediglich ein zusätzliches Format definieren. Wenn wir beim Erstellen leichter Abwandlungen besonders strukturiert vorgehen können, können wir es uns auch zunutze machen, dass Smarty Templates innerhalb von Templates unterstützt.

Wenn wir an die Einführung des Musters Modell-Präsentation-Steuerung zurückdenken, können wir feststellen, dass wir es jetzt sehr schön implementiert haben. Abbildung 3.8 bietet einen optischen Überblick über das, was wir bis jetzt erreicht haben.

Das Modell ist unsere Sammlung von Objekten des Geschäftsbereichs, die mithilfe von ORM automatisch dauerhaft in der Datenbank gespeichert werden. Die Präsentation ist das Template, das das XML-Format definiert. Die Steuerung ist die Seite mit der Suche nach Kategorien und jede andere Seite, die wir noch definieren. Sie hält das Modell und die Präsentation zusammen.

Dies ist die klassische Abbildung von MPS auf eine Webanwendung. Wir haben sie hier in der Webserverschicht einer Ajax-Anwendung durchgearbeitet, die XML-Dokumente bereitstellt, aber es lässt sich leicht erkennen, wie man dieses Muster auch für eine klassische Webanwendung einsetzen kann, die HTML-Seiten bereitstellt.

Je nachdem, mit welcher Technologie Sie arbeiten, werden Ihnen verschiedene Varianten dieses Musters begegnen, aber das Prinzip ist immer dasselbe. J2EE-Enterprise-Beans abstrahieren das Modell und die Steuerung so weit, dass sie sich auf verschiedenen Servern befinden können. »Code behind«-Klassen der .NET-Plattform delegieren die Rolle der Steuerung an seitenspezifische Objekte, wogegen Frameworks wie Struts als Steuerung einen »Front Controller« definieren, der alle Anfragen abfängt und an die Anwendung weiterleitet. In Frameworks wie Apache Struts ist dies zu einer eigenen Kunst entwickelt worden, wobei die Rolle der Steuerung nicht nur für einzelne Seiten angewendet wird, sondern auch so verfeinert wurde, dass sie die Benutzer von einer Seite zur anderen weiterleitet. (In einer Ajax-Anwendung können wir das mit JavaScript machen.) In allen Fällen ist die Zuordnung jedoch im Grunde dieselbe, und dies ist die Bedeutung, die MPS im Allgemeinen im Rahmen von Webanwendungen hat.

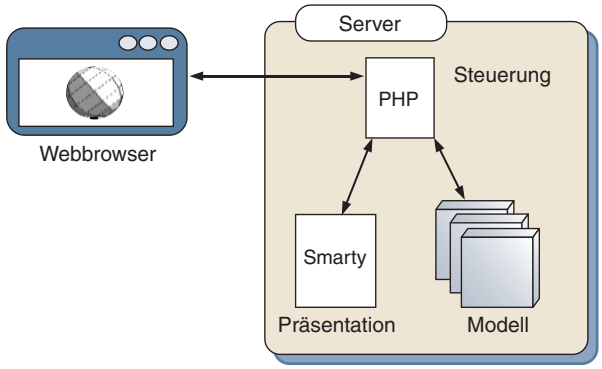

Abbildung 3.8: Das Muster MPS, wie es gewöhnlich für Webanwendungen eingesetzt wird. Die Webseite/das Servlet fungiert als Steuerung und fragt zuerst das Modell ab, um relevante Daten zu erhalten. Dann übergibt es diese Daten der Template-Datei (der Präsentation), die dann den Inhalt erzeugt, der an den Benutzer weitergeleitet wird. Beachten Sie, dass dies eine schreibgeschützte Situation ist. Wenn wir das Modell ändern, hätten wir einen anderen Ereignisfluss. Die Rollen blieben allerdings dieselben.

Unsere Webarchitektur mit MPS zu beschreiben, ist ein nützlicher Ansatz und wird uns auch in Zukunft dabei helfen, von klassischen zu Ajax-Anwendungen überzugehen. Dies ist aber nicht die einzige Möglichkeit, um MPS in Ajax einzusetzen. In Kapitel 4 untersuchen wir eine Variante des Musters, die es uns erlaubt, die Vorteile strukturierten Designs in der gesamten Anwendung zu genießen. Bevor wir dies tun, schauen wir uns jedoch eine andere Möglichkeit an, Ordnung in unsere Ajax-Anwendungen einzuführen.

Wir können den Code nicht nur durch Refactoring verbessern, sondern auch durch die Nutzung von Frameworks und Bibliotheken von Drittanbietern. Angesichts des wachsenden Interesses an Ajax sind eine Reihe von nützlichen Frameworks aufgekommen. Wir schließen dieses Kapitel mit einer kurzen Übersicht über einige der bekannteren ab.

3.5 Bibliotheken und Frameworks von Drittanbietern

Ein Ziel beim Refactoring besteht meistens auch darin, die Menge an Wiederholungen im Code zu verringern, indem Einzelheiten in eine gemeinsame Funktion oder ein Objekt ausgelagert werden. Wenn wir dies logisch bis zum Ende führen, können wir gemeinsam genutzte Funktionalitäten in Bibliotheken oder Frameworks zusammenfassen, die in verschiedenen Projekten wiederverwendet werden können. Das verringert die Menge an neuem Code, der für ein Projekt geschrieben werden muss, und erhöht die Produktivität. Da der Bibliothekscode bereits in früheren Projekten getestet worden ist, ist darüber hinaus auch eine höhere Qualität zu erwarten.

In diesem Buch entwickeln wir einige kleine JavaScript-Frameworks, die Sie in Ihren eigenen Projekten wiederverwenden können, z. B. `ObjectBrowser` in den Kapiteln 4 und 5, `CommandQueue` in Kapitel 5, das Benachrichtigungsframework in Kapitel 6, die `StopWatch`-Profilwerkzeuge in Kapitel 8 und die Debuggingkonsole in Anhang A. Außerdem führen wir jeweils am Ende der Kapitel 9 bis 13 ein Refactoring an den Lehrbeispielen durch, um Ihnen wiederverwendbare Komponenten an die Hand zu geben.

Natürlich sind wir nicht die einzigen, die so vorgehen, weshalb auch im Internet eine Menge JavaScript- und Ajax-Frameworks zur Verfügung stehen. Die namhafteren unter ihnen haben den Vorteil, dass sie bereits von vielen Entwicklern einer eingehenden Prüfung unterzogen wurden.

In diesem Abschnitt schauen wir uns einige Bibliotheken und Frameworks von Drittanbietern an, die der Ajax-Community zur Verfügung stehen. In der Welt der Ajax-Frameworks geht es zurzeit sehr emsig zu, sodass wir nicht alle Beiträge im Einzelnen vorstellen können, aber wir versuchen, Ihnen einen Eindruck davon zu verschaffen, welche Arten von Frameworks es gibt und wie Sie mit deren Hilfe Ordnung in Ihre Projekte bringen können.

3.5.1 Browserübergreifende Bibliotheken

Wie wir in Abschnitt 3.2.1 festgestellt haben, sind Browserinkonsistenzen beim Schreiben von Ajax-Anwendungen alles andere als unbekannt. Es gibt eine Reihe von Bibliotheken, die dem Zweck dienen, solche Inkonsistenzen zu überbrücken, indem sie eine allgemeine Fassade zur Verfügung stehen, die der Entwickler zum Kodieren heranziehen kann. Einige konzentrieren sich auf bestimmte Arten von Funktionalität, andere dienen dazu, eine umfassendere Programmierumgebung zur Verfügung zu stellen. Im Folgenden finden Sie eine Liste der Bibliotheken dieser Art, die wir beim Schreiben von Ajax-Code als hilfreich empfunden haben.

Die x-Bibliothek

Die x-Bibliothek ist eine ausgereifte Allzweckbibliothek zum Schreiben von DHTML-Anwendungen. Sie wurde 2001 als Ersatz für die frühere CBE-Bibliothek (Cross-Browser Extensions) des Autors veröffentlicht und wies einen sehr viel einfacheren

Programmierstil auf. Diese Bibliothek enthält browserübergreifende Funktionen zum Bearbeiten und Formatieren von DOM-Elementen und zur Arbeit mit dem Browser-Ereignismodell und darüber hinaus integrierte Bibliotheken zur Unterstützung von Animationen und Drag & Drop. Sie unterstützt Internet Explorer ab Version 4 sowie die jüngeren Versionen von Opera und Mozilla.

x verwendet einen einfachen, funktionsbasierten Kodierungsstil und nutzt die Vorteile der variablen Argumentlisten und losen Typisierung von JavaScript. So verwendet es für die gemeinsam nutzbare Methode `document.getElementById()`, die nur Strings als Eingabe akzeptiert, eine Funktion als Wrapper, die sowohl Strings als auch DOM-Elemente annimmt. Wird ein String übergeben, so wird er in die Element-ID aufgelöst, während ein DOM-Element unverändert zurückgegeben wird. So kann `xGetElementByID()` aufgerufen werden, um sicherzustellen, dass ein ID-Argument zu einem DOM-Knoten aufgelöst wird, ohne erst zu prüfen, ob dieses Argument bereits aufgelöst ist. Die Text-ID eines DOM-Elements durch das Element zu ersetzen, ist vor allem für dynamisch generierten Code nützlich, z.B. wenn der Methode `setTimeout()` oder einem Rückrufhandler ein String übergeben wird.

Ein ähnlich knapper Stil wird für die Methoden zur Bearbeitung der Stile von DOM-Elementen verwendet. Dabei dient ein und dieselbe Funktion sowohl als Get- als auch als Set-Methode. Die folgende Anweisung gibt z.B. die Breite des DOM-Elements `myElement` zurück, wobei es sich sowohl um ein DOM-Element selbst als auch um seine ID handeln kann:

```
xWidth(myElement)
```

Durch Hinzufügen eines weiteren Arguments können wir wie folgt die Breite des Elements festlegen:

```
xWidth(myElement,420)
```

Um die Breite eines Elements auf den gleichen Wert zu setzen wie die eines anderen, können wir Folgendes schreiben:

```
xWidth(zweitesElement,xWidth(erstesElement))
```

x enthält keinen Code für Netzwerkanfragen, ist aber dennoch eine nützliche Bibliothek für den Aufbau von Benutzeroberflächen in Ajax-Anwendungen. Sie ist in einem klaren, verständlichen Stil geschrieben.

Sarissa

Sarissa ist eine spezialisiertere Bibliothek als x und beschäftigt sich hauptsächlich mit der XML-Bearbeitung in JavaScript. Sie unterstützt die Grundfunktionen von Mozilla, Opera, Konqueror und Safari sowie die der ActiveX-Komponente MSXML von Internet Explorer (Version 3 und höher), wobei einige der erweiterten Funktionen wie XPath und XSLT nicht von allen Browsern unterstützt werden.

Die wichtigste Funktionalität für Ajax-Entwickler ist die browserübergreifende Unterstützung für das XMLHttpRequest-Objekt. Anstatt ein eigenes Fassade-Objekt zu erstellen, verwendet Sarissa das Muster Adapter, um ein XMLHttpRequest-Objekt auf der Grundlage von JavaScript für Browser zu erstellen, die kein natives Objekt dieses Namens anbieten (vor allem Internet Explorer). Intern nutzt dieses Objekt die ActiveX-Objekte, die wir in Kapitel 2 beschrieben haben, aber für Entwickler ist von Bedeutung, dass der folgende Code auf jedem Browser funktioniert, sobald Sarissa importiert wurde:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "myData.xml");
xhr.onreadystatechange = function(){
  if(xhr.readyState == 4){
    alert(xhr.responseXML);
  }
}
xhr.send(null);
```

Vergleichen Sie diesen Code mit Listing 2.11. Beachten Sie, dass die API-Aufrufe mit denen des nativen XMLHttpRequest-Objekts der Browser Mozilla und Safari identisch sind.

Wie bereits erwähnt bietet Sarissa auch eine Reihe allgemeiner Unterstützungsmechanismen für die Arbeit mit XML-Dokumenten, z. B. die Möglichkeit, beliebige JavaScript-Objekte für XML zu serialisieren. Diese Mechanismen können bei der Verarbeitung von XML-Dokumenten sehr nützlich sein, die als Antwort auf eine Ajax-Anfrage vom Server zurückgegeben wurden (falls Sie in Ihrem Projekt XML als Markup für Antwortdaten verwenden). Diese Möglichkeit und die Alternativen besprechen wir in Kapitel 5.

Prototype

Prototype ist eine Allzweckbibliothek für die JavaScript-Programmierung, deren Hauptgewicht auf einer Erweiterung der Sprache JavaScript selbst liegt, um einen stärker objektorientierten Programmierstil zu erlauben. Prototyp weist einen eigenen Stil der JavaScript-Kodierung auf, der auf diesen zusätzlichen Sprachfunktionen beruht. Prototype-Code, der entfernt vom Java-/C#-Stil abgeleitet ist, kann schwierig zu lesen sein, die Anwendung von Prototype und der darauf aufbauenden Bibliotheken ist jedoch einfach. Sie können sich Prototype als eine Bibliothek für Bibliotheksentwickler vorstellen. Autoren von Ajax-Anwendungen werden wahrscheinlich eher Bibliotheken auf der Grundlage von Prototype nutzen als Prototype selbst. Einige dieser Bibliotheken schauen wir uns in den folgenden Abschnitten an. Zuvor jedoch wollen wir Sie mit einer kurzen Erläuterung der Kernfunktionen von Prototype in diesen Kodierstil einführen, was für die Besprechung von Scriptaculous, Rico und Ruby on Rails hilfreich sein wird.

In Prototype ist es möglich, ein Objekt durch ein anderes zu »erweitern«, indem alle Eigenschaften und Methoden des übergeordneten Objekts in das untergeordnete kopiert werden. Dies lässt sich am besten anhand eines Beispiels erklären. Nehmen wir an, wir wollen die übergeordnete Klasse `Vehicle` definieren:

```
function Vehicle(numWheels,maxSpeed){
   this.numWheels=numWheels;
   this.maxSpeed=maxSpeed;
}
```

Eine Instanz dieser Klasse soll für einen Personenzug stehen. In dieser untergeordneten Klasse möchten wir die Anzahl der Wagons darstellen und einen Mechanismus bereitstellen, um Wagons hinzuzufügen und zu entfernen. In normalem JavaScript können wir das wie folgt schreiben:

```
var passTrain=new Vehicle(24,100);
passTrain.carriageCount=12;
passTrain.addCarriage=function(){
  this.carriageCount++;
}
passTrain.removeCarriage=function(){
  this.carriageCount--;
}
```

Dadurch stellen wir die erforderliche Funktionalität für unser passTrain-Objekt zur Verfügung. Wenn wir uns den Code aus dem Blickwinkel des Designs anschauen, stellen wir jedoch fest, dass er kaum etwas tut, um die erweiterte Funktionalität in eine geschlossene Einheit zu verpacken. Prototype hilft uns dabei, da wir damit das erweiterte Verhalten als Objekt definieren und das Basisobjekt damit erweitern können. Als Erstes definieren wir die erweiterte Funktionalität als Objekt:

```
function CarriagePuller(carriageCount){
  this.carriageCount=carriageCount;
  this.addCarriage=function(){
    this.carriageCount++;
  }
  this.removeCarriage=function(){
    this.carriageCount--;
  }
}
```

Anschließend führen wir die beiden zusammen, um ein einzelnes Objekt bereitzustellen, das das gesamte erforderliche Verhalten einschließt:

```
var parent=new Vehicle(24,100);
var extension=new CarriagePuller(12);
var passTrain=Object.extend(parent,extension);
```

Beachten Sie, dass wir das übergeordnete und das Erweiterungsobjekt zunächst getrennt voneinander definiert und dann zusammengeführt haben. Die Beziehung eines über- und untergeordneten Objekts besteht zwischen diesen Instanzen, nicht zwischen den Klassen `Vehicle` und `CarriagePuller`. Das ist zwar keine klassische Objektorientierung, aber es erlaubt uns, den gesamten Code für eine bestimmte Funktion, in diesem Fall das Ziehen von Wagons, an einem Ort zu konzentrieren, sodass er bequem wiederverwendet werden kann. Bei einem kleinen Beispiel wie diesem mag das unnötig erscheinen, aber in größeren Projekten ist es äußerst hilfreich, Funktionalität auf diese Weise zu kapseln.

Prototype bietet auch eine Ajax-Unterstützung, und zwar in Form eines Ajax-Objekts, das ein browserübergreifendes XMLHttpRequest-Objekt auflösen kann. Ajax wird durch den Typ `Ajax.Request` erweitert, der mithilfe von XMLHttpRequest wie folgt Anfragen an den Server stellen kann:

```
var req=new Ajax.Request('myData.xml');
```

Der Konstruktor verwendet einen Stil, den wir auch in vielen Bibliotheken auf der Grundlage von Prototype finden. Er nimmt ein assoziatives Array als zusätzliches Argument entgegen, wodurch nach Bedarf ein breites Spektrum an Optionen konfiguriert werden kann. Für jede Option werden sinnvolle Standardwerte bereitgestellt, sodass wir nur die Objekte übergeben müssen, die wir überschreiben möchten. Beim Konstruktor `Ajax.Request` ermöglicht das Optionen-Array, bereitzustellende Daten, Anfrageparameter, HTTP-Methoden und Rückrufhandler zu definieren. Ein angepasster Aufruf von `Ajax.Request` kann wie folgt aussehen:

```
var req=new Ajax.Request(
  'myData.xml',
  {
    method: 'get',
    parameters: { name:'dave',likes:'chocolate,rhubarb' },
    onLoaded: function(){ alert('loaded!'); },
    onComplete: function(){
      alert('done!\n\n'+req.transport.responseText);
    }
  }
);
```

Das Optionen-Array hat hier vier Parameter übergeben. Als HTTP-Methode ist `get` festgelegt, da Prototype standardmäßig `post` verwendet. Das Parameter-Array wird nach unten an den Abfragestring übergeben, da wir `get` verwenden. Bei `post` würde das Array zum Rumpf der Anfrage übergeben. `onLoaded` und `onComplete` sind Rückruf-Ereignishandler, die ausgelöst werden, wenn sich der `readyState` des zugrunde liegenden XMLHttpRequest-Objekts ändert. Die Variable `req.transport` der Funktion `onComplete` ist ein Verweis auf das zugrunde liegende XMLHttpRequest-Objekt.

Oberhalb von `Ajax.Request` definiert Prototype des Weiteren den Objekttyp `Ajax.Updater`, der auf dem Server erstellte Skriptfragmente abruft und auswertet. Dies folgt dem Muster, das wir in Kapitel 5 als »skriptzentriert« bezeichnet haben, und würde den Rahmen dieses Überblicks sprengen.

Damit schließen wir unseren kurzen Überblick über browserübergreifende Bibliotheken ab. Unsere Auswahl ist in gewisser Weise willkürlich und unvollständig. Wie wir bereits erwähnt haben, tut sich auf diesem Gebiet zurzeit sehr viel, weshalb wir uns auf einige der beliebteren oder bewährten Angebote beschränkt haben. Im nächsten Abschnitt schauen wir uns einige der Komponenten-Frameworks an, die auf der Grundlage dieser und anderer Bibliotheken erstellt wurden.

3.5.2 Komponenten und Komponentensammlungen

Die bis jetzt besprochenen Bibliotheken bieten eine browserübergreifende Unterstützung für einige eher »niedere« Funktionen wie die Bearbeitung von DOM-Elementen und den Abruf von Ressourcen vom Server. Mit diesen Werkzeugen zur Hand ist es sicherlich einfacher, funktionale Benutzeroberflächen und die Anwendungslogik zu erstellen, aber wir müssen immer noch weit mehr Arbeit aufwenden als die Kollegen, die z. B. Swing, MFC oder Qt einsetzen.

In letzter Zeit sind vorkonstruierte Komponenten, ja selbst ganze Komponentensätze für Ajax-Entwickler aufgekommen. In diesem Abschnitt schauen wir uns einige davon an. Auch dies ist kein umfassender Überblick, sondern dient nur dazu, Ihnen zu zeigen, welche verschiedenen Möglichkeiten sich Ihnen bieten.

Scriptaculous

Bei den Scriptaculous-Bibliotheken handelt es sich um Komponenten für Benutzeroberflächen, die auf Prototype aufbauen (siehe den vorherigen Abschnitt). In der jetzigen Form bietet Scriptaculous zwei Hauptfunktionalitäten. Es wird jedoch weiterentwickelt, und mehrere andere Funktionen sind geplant.

Die Effects-Bibliothek definiert eine Reihe animierter visueller Effekte, die auf DOM-Elemente angewendet werden können, sodass sie ihre Größe, Position und Transparenz ändern. Diese Effekte lassen sich leicht kombinieren. Außerdem stehen mehrere vordefinierte Sekundäreffekte zur Verfügung, etwa `Puff()`, bei dem ein Element immer größer und durchsichtiger wird, bevor es ganz verschwindet. Ein weiterer nützlicher Haupteffekt namens `Parallel()` ermöglicht die gleichzeitige Ausführung mehrerer Effekte. Wie wir in Kapitel 6 sehen werden, bilden solche Effekte eine nützliche Möglichkeit, um dem Benutzer einer Ajax-Oberfläche eine schnelle, optische Rückmeldung zu geben.

Der Aufruf eines vordefinierten Effekts erfolgt einfach durch den Aufruf seines Konstruktors und die Übergabe der DOM-Zielelemente oder seiner ID als Argument:

```
new Effect.SlideDown(myDOMElement);
```

Diese Effekte gründen sich auf das Prinzip eines Übergangsobjekts, das sich, was die Dauer und die am Ende des Übergangs aufzurufenden Ereignishandler angeht, parametrisieren lässt. Es stehen verschiedene Grundtypen von Übergängen zur Verfügung, z. B. linear, sinusförmig, schwankend und pulsierend. Um einen benutzerdefinierten Effekt zu erstellen, müssen Sie lediglich die vorgegebenen Effekte kombinieren und passende Parameter übergeben. Eine ausführliche Beschreibung würde jedoch den Rahmen dieses Überblicks sprengen. In Kapitel 6 schauen wir uns Scriptaculous-Effekte noch einmal genauer an, wenn wir ein Benachrichtigungssystem entwickeln.

Die zweite Funktion von Scriptaculous ist eine Drag & Drop-Bibliothek in Form der Klasse `Sortable`. Diese Klasse nimmt ein übergeordnetes DOM-Element als Argument entgegen und aktiviert die Drag & Drop-Funktionalität für alle untergeordneten Objekte. Mit Optionen, die dem Konstruktor übergeben werden, können Rückrufhandler für den Fall angegeben werden, dass ein Objekt verschoben und wieder losgelassen wird, sowie die Arten verschiebbarer untergeordneter Elemente und eine Liste gültiger Ziele für das Loslassen (also von Elementen, die das verschobene Element akzeptieren, wenn der Benutzer die Maustaste über ihnen loslässt). Auch Effektobjekte können als Optionen übergeben werden. Sie können zu Anfang der Verschiebeaktion ausgeführt werden, beim Übergang oder beim Loslassen.

Rico

Rico beruht wie Scriptaculous auf der Prototype-Bibliothek und bietet ebenfalls anpassungsfähige Effekte und Drag & Drop-Funktionen. Außerdem sind Verhaltensobjekte möglich, also Codefragmente, die sich auf einen Teil eines DOM-Baums anwenden lassen, um ihm interaktive Funktionen hinzuzufügen. Einige Verhaltensbeispiele stehen zur Verfügung, darunter die `Accordion`-Komponente, die einen Satz von DOM-Elementen innerhalb eines bestimmten Raums verschachtelt und jeweils eines davon erweitert. (Diese Art von Komponente wird oft als Outlook-Leiste bezeichnet, da sie durch Microsoft Outlook populär gemacht wurde.)

Lassen Sie uns eine einfache `Accordion`-Komponente erstellen. Zunächst brauchen wir ein übergeordnetes DOM-Element. Jedes untergeordnete Objekt wird ein Bereich innerhalb dieser »Ziehharmonika« sein. Wir definieren ein `div`-Element für jeden Bereich mit zwei weiteren `div`s darin, die jeweils den Header und den Rumpf des Bereichs darstellen:

```
<div id='myAccordion'>
  <div>
    <div>Dictionary Definition</div>
    <div>
      <ul>
      <li><b>n.</b>A portable wind instrument with a small
      keyboard and free metal reeds that sound when air is
      forced past them by pleated bellows operated by the
      player.</li>
      <li><b>adj.</b>Having folds or bends like the bellows
      of an accordion: accordion pleats; accordion blinds.</li>
```

```
      </ul>
    </div>
  </div>
  <div>
    <div>A picture</div>
    <div>
    <img src='monkey-accordion.jpg'></img>
    </div>
  </div>
</div>
```

Der erste Bereich enthält eine Definition des Wortes *accordion*, der zweite das Bild eines Affen, der eine Ziehharmonika spielt (siehe Abbildung 3.9). So dargestellt, würden diese Elemente einfach übereinander ausgegeben werden. Wir haben dem `div`-Element der obersten Ebene aber eine ID zugewiesen, sodass wir dem im Folgenden aufgeführten Accordion-Objekt einen Verweis darauf übergeben können:

```
var outer=$('myAccordion');
outer.style.width='320px';
new Rico.Accordion(
  outer,
  { panelHeight:400,
    expandedBg:'#909090',
    collapsedBg:'#404040',
  }
);
```

Die erste Zeile sieht merkwürdig aus. Bei `$` handelt es sich jedoch tatsächlich um einen gültigen JavaScript-Variablennamen, der einfach auf eine Funktion in der Kernbibliothek von Prototype verweist. `$()` löst DOM-Knoten ähnlich auf wie die Funktion `xGetElementById()` der x-Bibliothek, die wir im vorhergehenden Abschnitt besprochen haben. Dem Konstruktor des Accordion-Objekts übergeben wir einen Verweis auf das aufgelöste DOM-Element sowie ein Array von Optionen im Standarddialekt der von Prototype abgeleiteten Bibliotheken. In diesem Fall enthalten die Optionen lediglich die Formatierung der optischen Elemente der `Accordion`-Komponente. Es können hier jedoch auch Rückrufhandler-Funktionen übergeben werden, die beim Öffnen oder Schließen der Bereiche ausgelöst werden sollen. Abbildung 3.9 zeigt die Auswirkung der Formatierung dieser DOM-Elemente mithilfe des Accordion-Objekts. Die Verhalten von Rico bieten eine einfache Möglichkeit, um wiederverwendbare Komponenten aus gewöhnlichem Markup zu erstellen und den Inhalt von der Interaktivität zu trennen. Die Anwendung guter Designprinzipien auf JavaScript-Benutzeroberflächen ist Thema von Kapitel 4.

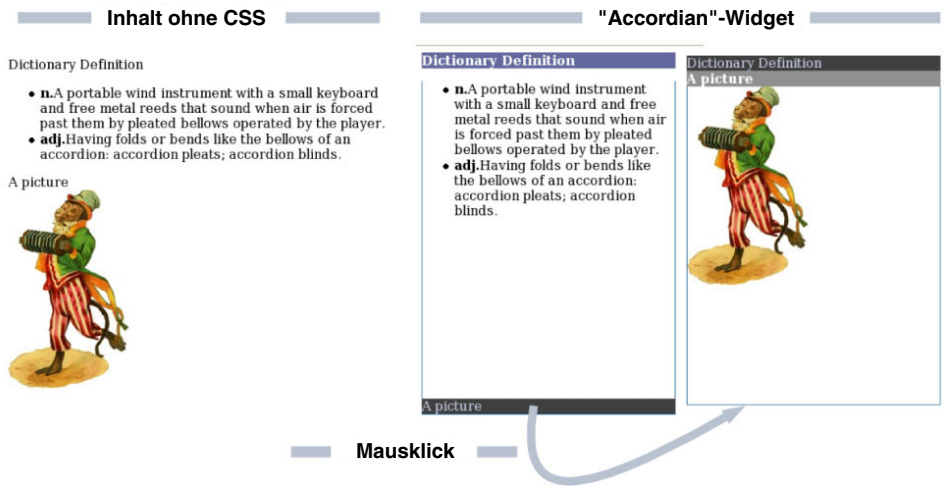

Abbildung 3.9: Die Verhalten des Rico-Frameworks erlauben es, einfache DOM-Knoten als interaktive Komponenten zu formatieren, indem einfach ein Verweis auf den obersten Knoten des Konstruktors für das Verhaltensobjekt übergeben wird. In diesem Fall wurde das Accordion-Objekt auf einen Satz von div-Elementen angewendet (links), um eine interaktive Menü-Komponente zu erstellen (rechts), in dem die einzelnen Bereiche per Mausklick geöffnet und geschlossen werden.

Das letzte Merkmal des Rico-Frameworks, das wir hier erwähnen möchten, ist die gute Unterstützung für Ajax-artige Anfragen an den Server durch das globale Rico-Objekt `AjaxEngine`. Dieses Objekt ist mehr als nur ein browserübergreifender Wrapper um das XMLHttpRequest-Objekt. Es definiert ein XML-Antwortformat, das aus einer Reihe von `<response>`-Elementen besteht. Die Engine dekodiert sie automatisch und verfügt über eine eingebaute Unterstützung für zwei Arten von Antworten: solche, die DOM-Elemente direkt aktualisieren, und solche, die JavaScript-Objekte aktualisieren. In Abschnitt 5.5.3 schauen wir uns ähnliche Mechanismen etwas genauer an, wenn wir Client/Server-Interaktionen eingehend erklären. Jetzt aber fahren wir mit dem nächsten Typ von Framework fort: einem, das sowohl Client als auch Server umfasst.

3.5.3 Anwendungs-Frameworks

Die Frameworks, die wir bis jetzt besprochen haben, werden ausschließlich im Browser ausgeführt und können als statische JavaScript-Dateien von jedem Webserver bereitgestellt werden. Die letzte Kategorie von Frameworks, die wir hier vorstellen, ist jedoch für den Einsatz auf dem Server gedacht und erstellt zumindest einen Teil des JavaScript-Codes und HTML-Markups dynamisch.

Hierbei handelt es sich um die komplexesten Frameworks, die wir an dieser Stelle besprechen. Wir können Sie nicht in allen Einzelheiten darstellen, sondern geben nur einen kurzen Überblick über ihre Funktionen. Das Thema serverseitiger Frameworks sprechen wir in Kapitel 5 erneut an.

DWR, JSON-RPC und SAJAX

Zu Beginn schauen wir uns drei kleine serverseitige Frameworks zusammen an, da sie einen gemeinsamen Ansatz verfolgen, obwohl sie für verschiedene serverseitige Sprachen geschrieben sind. SAJAX funktioniert mit mehreren Sprachen, darunter PHP, Python, Perl und Ruby. DWR (Direct Web Remoting) ist ein Framework auf Java-Basis mit einem ähnlichen Ansatz, das Methoden von Objekten bereitstellt statt eigenständiger Funktionen. Auch der Entwurf von JSON-RPC (Java Script Object Notation-based Remote Procedure Calls) ist ähnlich. Dieses Framework bietet Unterstützung für serverseitiges JavaScript, Python, Ruby, Perl und Java.

Alle drei ermöglichen es, Objekte auf dem Server zu definieren, um deren Methoden direkt als Ajax-Anfragen bereitzustellen. Wir begegnen häufig serverseitigen Funktionen, die ein Ergebnis zurückgeben, das auf dem Server berechnet werden muss, da sie z.B. einen Wert einer Datenbank nachschlagen. Diese Frameworks bieten eine bequeme Möglichkeit, um vom Webbrowser aus auf solche Funktionen und Methoden zuzugreifen, und sind auch dazu geeignet, das serverseitige Modell des Geschäftsbereichs dem Webbrowsercode gegenüber bereitzustellen.

Schauen wir uns ein AJAX-Beispiel an, bei dem auf dem Server definierte Funktionen in PHP bereitgestellt werden. Wir betrachten die folgende einfache Beispielfunktion, die lediglich einen Textstring zurückgibt:

```
<?php
function sayHello(name){
   $name = htmlspecialchars($name);
   return("Hello! ($name) Ajax in Action!!!!");
?>
```

Um diese Funktion in die JavaScript-Schicht zu exportieren, können wir einfach die SAJAX-Engine in PHP importieren und die Funktion `sajax_export` aufrufen:

```
<?php
require 'Sajax.php';
sajax_init();
sajax_export("sayHello");
?>
```

Wenn wir unsere dynamische Webseite schreiben, verwenden wir SAJAX, um JavaScript-Wrapper für die exportierten Funktionen zu erstellen. Der generierte Code erstellt eine lokale JavaScript-Funktion mit identischen Signaturen wie die serverseitigen Funktionen:

```
<script type='text/javascript'>
<?php
   sajax_show_javascript();
?>
...
alert(sayHello("Dave"));
...
</script>
```

Wenn wir `sayHello("Dave")` im Browser aufrufen, nimmt der generierte JavaScript-Code eine Ajax-Anfrage an den Server vor, führt die serverseitige Funktion aus und gibt das Ergebnis in der HTML-Antwort zurück. Die Antwort wird analysiert, der Rückgabewert in das JavaScript extrahiert. Der Entwickler hat mit keiner dieser Ajax-Technologien etwas zu tun; alles wird hinter den Kulissen von den SAJAX-Bibliotheken erledigt.

Diese drei Frameworks bieten eine ziemlich »niedere« Abbildung der serverseitigen Funktionen und Objekte auf die clientseitigen Ajax-Aufrufe. Sie automatisieren, was anderenfalls eine sehr mühselige Arbeit wäre, aber sie bringen auch die Gefahr mit sich, zu viel serverseitige Logik im Internet bereitzustellen. Dieses Problem besprechen wir ausführlicher in Kapitel 5.

Die restlichen Frameworks, die wir uns in diesem Abschnitt anschauen, verfolgen einen anspruchsvolleren Ansatz, indem sie ganze Schichten von Benutzeroberflächen aus den auf dem Server deklarierten Modellen erstellen. Sie verwenden intern zwar Ajax-Technologien, stellen aber im Grunde genommen ihr eigenes Programmiermodell bereit. Daher ist die Arbeit mit diesen Frameworks ganz anders als das Schreiben von allgemeinem Ajax-Code, sodass wir hier nur einen groben Überblick geben können.

Backbase

Der Backbase Presentation Server bietet eine reichhaltige Sammlung von Komponenten, die zur Laufzeit an XML-Tags in den vom Server generierten HTML-Dokumenten gebunden werden. Dies ist ein ähnliches Prinzip wie bei den Verhaltenskomponenten von Rico, abgesehen davon, dass Backbase einen eigenen Satz von XHTML-Tags zur Kennzeichnung der Komponenten der Benutzeroberfläche verwendet und nicht die standardmäßigen HTML-Tags.

Backbase stellt serverseitige Implementierungen für sowohl Java als auch .NET zur Verfügung. Es handelt sich um ein kommerzielles Produkt, von dem es aber auch eine kostenlose Community-Ausgabe gibt.

Echo2

Das Framework Echo2 von NextApp ist eine Server-Engine auf Java-Basis, die Komponenten für intelligente Oberflächen aus einem auf dem Server deklarierten Modell dieser Oberfläche erstellt. Nachdem sie im Browser gestartet sind, verhalten sich die Komponenten ziemlich autonom und verarbeiten Benutzerinteraktionen lokal mithilfe von JavaScript oder senden anderenfalls mithilfe einer ähnlichen Anfragewarteschlange wie der von Rico Anfragen in Stapeln an den Server zurück.

Echo2 wird als Ajax-basierte Lösung angepriesen, für die keinerlei Kenntnisse von HTML, JavaScript oder CSS erforderlich sind, es sei denn, Sie wollen den Satz der vorhandenen Komponenten erweitern. In den meisten Fällen erfolgt die Entwicklung der Clientanwendung nur mit Java. Echo2 ist ein Open-Source-Produkt mit einer Mozilla-ähnlichen Lizenz, die die Verwendung in kommerziellen Anwendungen erlaubt.

Ruby on Rails

Ruby on Rails ist ein Webentwicklungs-Framework, das in der Programmiersprache Ruby geschrieben worden ist. Es bündelt Lösungen für die Zuordnung serverseitiger Objekte zu Datenbanken und zur Darstellung von Inhalten mithilfe von Templates in einem sehr ähnlichen Stil wie dem des serverseitigen MPS-Musters aus Abschnitt 3.4. Mit Ruby on Rails soll eine sehr schnelle Entwicklung von einfachen und mittleren Websites möglich sein, da es Techniken zur Codeerzeugung einsetzt, mit der eine Menge allgemein nutzbarer Code erstellt werden kann. Außerdem soll es den Konfigurationsaufwand dafür verringern, eine Anwendung zum Laufen zu bringen.

In den jüngsten Versionen bietet Rails eine starke Unterstützung für Ajax durch die Prototype-Bibliothek. Prototype und Rails passen von Natur aus gut zusammen, da der JavaScript-Code für Prototype mit einem Ruby-Programm erstellt wurde und sich die Programmierstile ähneln. Ajax mit Rails erfordert ebenso wie Echo2 kein umfassendes Wissen über Ajax-Technologien wie JavaScript, aber ein Entwickler, der JavaScript versteht, kann die Ajax-Unterstützung in ganz neue Richtungen erweitern.

Damit beenden wir unseren Überblick über Ajax-Frameworks von Drittanbieten. Wie wir bereits festgestellt haben, herrscht auf diesem Gebiet viel Bewegung, und viele der hier besprochenen Frameworks werden noch aktiv weiterentwickelt.

Viele der Bibliotheken und Frameworks weisen ihre eigenen Kodierdialekte und Stile auf. Bei den Codebeispielen in diesem Buch haben wir versucht, ein Gefühl für die breite Streuung von Ajax-Technologien und -Techniken zu vermitteln, und vermieden, uns zu sehr auf ein bestimmtes Framework zu stützten. Dennoch werden Ihnen hier und da in diesem Buch verstreut einige der Produkte begegnen, die wir hier vorgestellt haben.

3.6 Zusammenfassung

In diesem Kapitel haben wir das Prinzip des Refactorings als eine Möglichkeit eingeführt, die Qualität und Flexibilität des Codes zu erhöhen. Unsere erste Begegnung mit dem Refactoring bestand darin, das XMLHttpRequest-Objekt – den Kern des Ajax-Stapels – zu einem einfachen, wiederverwendbaren Objekt zu machen.

Wir haben uns eine Reihe Entwurfsmuster angesehen, die wir anwenden können, um häufig auftretende Probleme bei der Arbeit mit Ajax zu lösen. Entwurfsmuster bieten einen halb formellen Weg, um die Erkenntnisse von Programmierern festzuhalten, und können uns helfen, ein Refactoring zu einem bestimmten Ziel vorzunehmen.

Die Muster Fassade und Adapter bieten nützliche Möglichkeiten, um die Unterschiede zwischen verschiedenen Implementierungen auszugleichen. In Ajax sind diese Muster vor allem dazu hilfreich, um eine isolierende Schicht gegen Browserinkompatibilitäten anzulegen, ein größere und ständige Quelle der Sorge für JavaScript-Entwickler.

Beobachter ist ein flexibles Muster für den Umgang mit ereignisgesteuerten Systemen. Wir werden in Kapitel 4 darauf zurückkommen, wenn wir uns die Schichten der Benutzeroberfläche für unsere Anwendung ansehen. Zusammen mit dem Muster Befehl, das eine gute Möglichkeit zur Kapselung von Benutzeraktionen bietet, ist es damit möglich, ein solides Framework für die Verarbeitung von Benutzerangaben und zur Bereitstellung einer Rückgängig-Funktion zu erstellen. Das Muster Befehl wird auch für die Gliederung von Client/Server-Interaktionen eingesetzt, wie wir in Kapitel 5 sehen werden.

Das Muster Singleton bietet einen einfachen Weg, um den Zugriff auf bestimmte Ressourcen zu steuern. In Ajax können wir Singleton sinnvoll einsetzen, um den Zugriff auf das Netzwerk zu regeln, wie wir ebenfalls in Kapitel 5 sehen werden.

Schließlich haben wir das Muster Modell-Präsentation-Steuerung (MPS) eingeführt, ein Architekturmuster, dessen Einsatz für Webanwendungen schon eine lange Geschichte hat (zumindest im zeitlichen Maßstab des Internets). Wir haben erfahren, wie der Einsatz von MPS die Flexibilität einer serverseitigen Anwendung durch eine abstrahierte Datenschicht und ein Template-System erhöhen kann.

Unser Beispiel eines Bekleidungsgeschäfts hat auch veranschaulicht, in welcher Weise Entwurfsmuster und Refactoring Hand in Hand gehen. Code gleich beim ersten Versuch perfekt zu gestalten, ist sehr schwer, aber es ist möglich, Code wie dem aus Listing 3.4, der hässlich ist, aber funktioniert, mithilfe von Refactorings nach und nach die Vorteile von Entwurfsmustern angedeihen zu lassen, bis das Endergebnis in allen Einzelheiten gut ist.

Zum Abschluss haben wir uns Bibliotheken und Frameworks von Drittanbietern als eine weitere Möglichkeit angeschaut, Ordnung in ein Ajax-Projekt einzuführen. Zurzeit werden eine Reihe von Bibliotheken und Frameworks entwickelt, wobei die Palette von einfachen browserübergreifenden Wrappern über vollständige Komponentensammlungen bis hin zu durchgängigen Lösungen für sowohl den Client als auch den Server reicht. Wir haben einige der bekannteren Frameworks kurz vorgestellt und werden auf einige davon in den folgenden Kapiteln zurückkommen.

In den nächsten beiden Kapiteln wenden wir unsere Kenntnisse über Refactoring und Entwurfsmuster auf den Ajax-Client und anschließend auf das Client/Server-Kommunikationssystem an. Dadurch können wir ein Vokabular und eine Reihe von Vorgehensweisen entwickeln, mit deren Hilfe es einfacher wird, stabile und funktionsreiche Webanwendungen zu erstellen.

3.7 Quellen

Martin Fowler hat (zusammen mit den Co-Autoren Kent Beck, John Brant, William Opdyke und Don Roberts) die bahnbrechende Einführung in das Refactoring geschrieben: *Refactoring, oder: Wie Sie das Design vorhandener Software verbessern* (Addison-Wesley, 2005).

Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides – die so genannte »Viererbande« (»Gang of Four«) – sind die Autoren des einflussreichen Buchs *Entwurfsmuster* (Addison-Wesley, 2004).

Gamma wurde später Architekt der Eclipse IDE-Plattform (siehe Anhang A) und beschreibt sowohl Eclipse als auch Entwurfsmuster in folgendem Interview: *www.artima.com/lejava/articles/gammadp.html*.

Michael Mehmoff hat vor kurzem eine Website zur Katalogisierung von Ajax-Entwurfsmustern gestartet: *www.ajaxpatterns.org*.