Master Thesis

Konzeption und Realisierung eines graphischen Editors für Report-Vorlagen

Tilman Schneider

1. Oktober 2005

Zusammenfassung

Bei der Karlsruher PTV AG wird ein vom STZ-IDA entwickeltes J2EE-Report-Werkzeug eingesetzt. Damit lassen sich Daten aus verschiedenen Quellen aufbereiten und in HTML (später evtl. auch als PDF) ausgeben.

Die Definition der Reports erfolgt in Form von Vorlagen (Templates). Als Template-Sprache kommt Velocity [Velocity05] zum Einsatz, d.h. jedes Template ist als ein Skript anzusehen, das für die Report-Erstellung mit Parametern versorgt und ausgeführt wird. Ergebnis ist ein XML-Dokument, das die Bestandteile des Reports in abstrakter, an HTML angelehnter Form enthält: `<p>` für Abschnitte, `<table>` für Tabellen, `<chart>` für Grafiken, etc. Dieses Dokument wird in einem weiteren Schritt per XSLT in das gewünschte Zielformat, z.B. HTML, transformiert.

Ziel der Master Thesis war die Erstellung eines graphischen Editors für die Templates. Der Benutzer sollte Velocity-Anweisungen und Layout-Elemente zusammenstellen und ineinander schachteln können. Die Darstellung sollte dabei, was die Layout-Elemente angeht, weitgehend dem fertigen Report entsprechen, während die normalerweise unsichtbaren Velocity-Anweisungen durch Platzhalter repräsentiert werden sollten. Außerdem sollte ein schnelles Umschalten zwischen Entwurfs- und Laufzeitmodus möglich sein, um den Report zu testen. Bei der Umsetzung sollte darauf geachtet werden, dass sich das System auch für andere Template-Sprachen anpassen lässt, z. B. für JSPs.

Dieser graphische Editor wurde in Form einer Webanwendung umgesetzt, die auf AJAX basiert. AJAX ist eine Kombination verschiedener offener Standards, die es erlaubt, Webanwendungen zu entwickeln, die sich so flüssig bedienen lassen wie eine Desktop-Anwendung, gleichzeitig jedoch nur einen gewöhnlichen Browser ohne Plugins oder Ähnliches auf dem Client voraussetzt. Solche Anwendungen vereinen damit die Vorteile von Thin Clients und Fat Clients: Eine flüssige Bedienung, die Unterstützung moderner Bedienkonzepte wie Drag-and-Drop, reduzierte Serverlast, zentrale Verwaltung der Anwendung auf dem Server und kein Installations- oder Wartungsaufwand auf der Clientseite.

©2005 Tilman Schneider

Abstract

The PTV AG located in Karlsruhe uses a J2EE report tool developed by the STZ-IDA. This tool refines data from various sources and generates HTML reports (possibly PDF reports in future, too).

The reports are defined by templates. Velocity [Velocity05] is used as template language, i.e. every template may be regarded as script that is provided with parameters and executed. The result is a XML document that contains the elements of the report in an abstract way similar to HTML: `<p>` for paragraphs, `<table>` for tables, `<chart>` for charts, etc. In a following step this document is transformed by XSLT to the target format, e.g. HTML.

The goal of this master thesis was the development of a graphical editor for these templates. The user should be facilitated to combine Velocity statements and layout elements and to interleave them. The layout elements should be displayed in a similar way as they are displayed in the finished report, while the normally invisible Velocity statements should be shown using placeholders. Moreover a fast switching between the design view and the runtime view should be possible in order to test the report. The system should also be adaptable to other template languages, e.g. JSP.

This graphical editor has been implemented as web application based on AJAX. AJAX is a combination of several open standards, that allow the development of web applications that can be used as smooth as desktop applications, only requiring an ordinary web browser without any plugins or similar on the client side. Such applications thereby combine the advantages of thin clients and fat clients: A smooth usability, the support of modern UI concepts like drag and drop, a reduced server load, central management of the application on the server and no installation or maintenance expense on the client side.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde zur Erlangung eines akademischen Grades vorgelegt.

Karlsruhe, den 1. Oktober 2005

____

Tilman Schneider

©2005 Tilman Schneider

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**

2.1 JavaScript . . . 3

2.1.1 Objekte in JavaScript . . . 3

2.1.2 Klassen in JavaScript . . . 4

2.1.3 Kapselung in JavaScript . . . 5

2.1.4 Vererbung in JavaScript . . . 6

2.1.5 Pakete in JavaScript . . . 7

2.1.6 Polymorphismus in JavaScript . . . 8

2.1.7 Typprüfung in JavaScript . . . 9

2.1.8 Selbstbeschreibung in JavaScript . . . 9

2.1.9 Was JavaScript nicht kann . . . 10

2.1.10 Tool-Unterstützung . . . 11

2.2 AJAX . . . 12

2.3 Velocity . . . 14

2.4 Reports . . . 15

3 Aufgabenstellung **16**

3.1 Der PTV-Reporter . . . 16

3.2 Anforderungen an den Report-Editor . . . 17

4 Analyse **18**

4.1 Der PTV-Reporter . . . 18

4.2 Der alte Editor . . . 19

4.2.1 Bewertung . . . 20

4.3 Der neue Editor . . . 21

4.4 Use-Cases . . . 22

4.5 Die Benutzer . . . 23

4.6 Ergebnis der Editor-Recherche . . . 25

4.7 Aus der Analyse gewonnene Anforderungen . . . 26

5 Entwurf **28**
5.1 Benutzerführung und Bedienkonzepte 28
5.1.1 Allgemeine Überlegungen 28
5.1.2 Hinzufügen und Verschieben von Report-Knoten 29
5.1.3 Verändern von Report-Knoten 30
5.1.4 Der Papierentwurf 30
5.1.5 Prüfung des Papierentwurfs 32
5.2 Wahl der Basistechnologie 32
5.2.1 Fat Client: Browser als Widget 33
5.2.2 Rich Client: Browser plus Plugin 34
5.2.3 Rich Client: Dynamisches HTML 35
5.2.4 Entscheidung 35
5.3 Architektur 37
5.3.1 Verallgemeinerung des Editors 37
5.3.2 Adapter-Modell 38
5.3.3 Multilevel-Undo 41
5.3.4 Die Eigenschaftenansicht 42
5.3.5 Drag-and-Drop 44
5.3.6 Entwurf eines komplexen Widgets 48

6 Implementierung **50**
6.1 Implementierung eines komplexen Widgets 50
6.2 Drag-and-Drop 52
6.2.1 Die Rolle von Iframes im Report-Editor 53
6.2.2 Die Drag-and-Drop-Bibliothek von Walter Zorn 55
6.2.3 Die selbst entwickelte Drag-and-Drop-Implementierung 55
6.3 EventDispatcherFactory 56
6.4 Die WYSIWYG-Anzeige 58
6.5 Der Velocity-Parser 60
6.6 Server-Seite 63
6.6.1 Client-Server-Kommunikation 63
6.6.2 Erweiterung des PTV-Reporters 66
6.6.3 Service-Servlet 68

7 Ergebnisse **69**
7.1 Vergleich mit der Aufgabenstellung 69
7.2 Allgemeine Erfahrungen mit AJAX 71

8 Ausblick **74**

©2005 Tilman Schneider

Literaturverzeichnis **75**

Glossar **77**

Index **80**

Abbildungsverzeichnis

2.1 Der JavaScript-Debugger von Firefox 12
2.2 Der DOM Inspector von Firefox 13

4.1 Flussdiagramm – Erstellung eines Reports im PTV-Reporter 19
4.2 Das Eclipse-Plugin Velocity Web Edit 21
4.3 Persona – Entwickler bei PTV 24
4.4 Persona – Berater bei PTV 24
4.5 Persona – Entwickler beim Kunden 25
4.6 Persona – Benutzer beim Kunden 25

5.1 Master-Detail-Ansicht bei Eclipse 29
5.2 Entwurfsskizze der Oberfläche 31
5.3 Klassendiagramm der Editor-Abstraktion 39
5.4 Klassendiagramm des Adapter-Modells 40
5.5 Klassendiagramm des Multilevel-Undo 41
5.6 Klassendiagramm der Eigenschaftenansicht 42
5.7 Die Eigenschaftenansicht 43
5.8 Klassendiagramm des Drag-And-Drop 45
5.9 Ein Schiebebalken wird per Drag-and-Drop bewegt 46
5.10 Ein row-Element wird in eine Tabelle gezogen 47
5.11 Klassendiagramm der TreeView 49

6.1 Aufbau der TreeView aus HTML-Elementen 51
6.2 Durch den Iframe bekommt die Gliederungsansicht Scrollbalken 54
6.3 Flussdiagramm – Erstellung der Vorschau im Report-Editor 59
6.4 Klassendiagramm des Parsers 61
6.5 Aufrufstack der Client-Server-Kommunikation 64

7.1 Der Report-Editor 70

Quelltextverzeichnis

2.1 Objektstrukturen in JavaScript 4
2.2 Klassendefinition in JavaScript und Java nach [Jung05] 5
2.3 Kapselung in JavaScript und Java nach [Jung05] 6
2.4 Vererbung in JavaScript und Java nach [Jung05] 7
2.5 Pakete in JavaScript und Java nach [Jung05] 8
2.6 Polymorphismus in JavaScript und Java nach [Jung05] 8
2.7 Typprüfung in JavaScript und Java nach [Jung05] 9
2.8 Selbstbeschreibung unbekannter Objekte 9
2.9 Ermittlung des Typ-Namens in JavaScript und Java nach [Jung05] . . . 10
2.10 Beispiel für ein Velocity-Skript, das ein XML-Dokument erzeugt 15
4.1 Beispiel für eine Report-Vorlage 20
6.1 Ereignisverarbeitung in JavaScript 56
6.2 Ereignisverarbeitung mit Hilfe der `EventDispatcherFactory` 58
6.3 Eine Report-Vorlage eingebettet in einem XML-Containern 67

©2005 Tilman Schneider

Kapitel 1

Einleitung

Web-Applikationen nehmen in der Informatik einen sehr großen Stellenwert ein. Sie erlauben eine recht einfache und schnelle Entwicklung von Anwendungen, die weltweit von vielen Benutzern gleichzeitig genutzt werden können. Die Daten und die Programmlogik liegen dabei auf dem Server, so dass Sicherheitsmaßnahmen, Backups und Programmaktualisierungen an einer zentralen Stelle vorgenommen werden können, die zudem unter der vollen Kontrolle des Dienstanbieters liegt. Die potentiell unsicheren Clients dienen nur dazu, die vom Server generierten Antwortseiten darzustellen.

Dem gegenüber steht der Fat-Client-Ansatz: Ein mächtiges, proprietäres Programm auf der Client-Seite, das versucht, möglichst viel Logik auf dem Client auszuführen. Das hat den Vorteil, dass die Anwendung sehr flüssig zu bedienen ist und dass der Server geschont wird. Angesichts der vielen Vorteile von leichtgewichtigen Web-Applikationen scheinen Fat-Client-Lösungen jedoch zunehmend verdrängt zu werden. Zu groß sind die Probleme bei der Installation, Aktualisierung, Wartung und Sicherheit.

Trotzdem lassen Web-Anwendungen einige Wünsche offen: Die Bedienung ist nicht wirklich flüssig, da der Browser bei jeder Aktion eine Anfrage an den Server sendet, welcher daraufhin eine komplett neue Seite generiert, selbst wenn sich nur ein kleiner Teil der Ansicht verändert hat. Dadurch ist die Bedienung einer Web-Applikation mit ständigen kleinen Unterbrechungen verbunden und erzeugt unnötig Last im Netzwerk und beim Server. Erschwerend kommt hinzu, dass oft gar keine Kommunikation notwendig wäre, weil die eigentliche Information in vielen Fällen bereits früher vom Client abgefragt wurde.

Diese Arbeit zeigt anhand eines komplexen Vorlagen-Editors, dass die heutigen Browser bereits mächtig genug sind, Applikationen auszuführen, die die Vorteile beider Welten

vereinen. Die Skriptsprache JavaScript ist mittlerweile so ausgereift, dass sich moderne OO-Techniken, wie beispielsweise das Entwurfsmuster Model-View-Controller, damit umsetzen lassen. Damit lassen sich komplexe Anwendungen erstellen, die auf einem gängigen Browser laufen – ohne Plugins oder andere besondere Software und unter Verwendung freier Standards des W3C bzw. Quasi-Standards, wie dem XMLHttpRequest.

Auf Client-Seite muss dadurch keine besondere Software installiert und aktuell gehalten werden und die gesamte Applikation kann zentral auf dem Server verwaltet werden. Gleichzeitig ist die Anwendung flüssig bedienbar, bietet moderne Bedienkonzepte, wie z. B. Drag-and-Drop, und kommuniziert nur dann mit dem Server, wenn es nötig ist, wobei auch nur die eigentlichen Daten ausgetauscht werden und keine Layoutinformation.

©2005 Tilman Schneider

Kapitel 2

Grundlagen

2.1 JavaScript

Nach [JScript05] ist JavaScript (kurz: JScript) eine objektbasierte Scriptsprache, die von Netscape entwickelt wurde. Die Grundlagen von JavaScript, wie die Syntax, die Sprachelemente und die grundlegenden Datentypen, sind unter dem Namen ECMAScript standardisiert. JavaScript war zwar von Anfang an sowohl für den Einsatz im Client als auch im Server konzepiert. In der Praxis wird JavaScript jedoch hauptsächlich clientseitig im Web-Browser eingesetzt.

Der Name JavaScript wurde aus reinen Marketinggründen gewählt. JavaScript hat zwar eine ähnliche Syntax wie die Programmiersprache Java, hat ansonsten jedoch nichts mit Java zu tun.

2.1.1 Objekte in JavaScript

In JavaScript ist alles ein Objekt, sogar Funktionen. Jedes Objekt kann dynamisch um Referenzen auf andere Objekte, Eigenschaften[1] genannt, erweitert werden, so dass sich eine Baumstruktur[2] ergibt. Über dieses Prinzip lassen sich einem Objekt Variablen oder

[1] Engl. „property“: Eigenschaft

[2] Da man auch Zyklen bilden kann und da ein Objekt von vielen Objekten referenziert werden kann, handelt es sich genau genommen um einen Objekt-Graph, der jedoch in der Praxis in weiten Teilen einem Baum entspricht.

```
var person = new Object();

// Hinzufügen einer Variablen
person.name = "Karl Birne";

// Hinzufügen einer Methode
person.showName = function() {
  alert("Mein Name ist " + this.name);
}
```

Quelltext 2.1: Objektstrukturen in JavaScript

Methoden hinzufügen, indem man ihm ein Daten- bzw. Funktionsobjekt als Eigenschaften zuweist (siehe Quelltext 2.1).

Wie [Jung05] zeigt, können alle wichtigen OO-Techniken in JavaScript verwendet werden. Die folgenden Quelltexte zeigen die einzelnen Techniken im Vergleich zu Java. Auf der linken Seite befindet sich jeweils der JavaScript-Code, auf der rechten der entsprechende Java-Code.

2.1.2 Klassen in JavaScript

Im Gegensatz zu klassischen objektorientierten Sprachen gibt es in JavaScript keine Klassen – es gibt nur Objekte. JavaScript wird deshalb auch als „objektbasierte“ Sprache bezeichnet.

Es kann jedoch ein Klassenverhalten erreicht werden, indem die Variablen und Methoden einer Klasse einem Funktionsobjekt zugewiesen werden. Die Funktion dient dabei als Konstruktor. Die Definition und Initialisierung von Variablen fällt zusammen. Das Objekt besteht aus dem, was im Konstruktor angelegt wird (siehe Quelltext 2.2 auf der nächsten Seite).

Wenn im weiteren Verlauf dieser Arbeit im Zusammenhang mit JavaScript von Klassen gesprochen wird, dann ist damit die hier vorgestellte Konstruktion gemeint.

```
// Klassendefinition in JavaScript
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}





// Instanziierung in JavaScript
var person = new Person("Karl",
                        "Birne");
```

```
// Klassendefinition in Java
public class Person {
  public String firstName;
  public String lastName;

  public Person(String firstName,
                String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}

// Instanziierung in Java
Person person = new Person("Karl",
                           "Birne");
```

Quelltext 2.2: Klassendefinition in JavaScript und Java nach [Jung05]

2.1.3 Kapselung in JavaScript

Eine wichtige OO-Technik ist die Kapselung. Ein Objekt soll nach außen hin nur die Schnittstelle zeigen, die es zu seiner Benutzung vorsieht. Insbesondere soll es keinen direkten Zugriff auf seine Daten erlauben.

In JavaScript erreicht man dieses Verhalten, indem man eine lokale Variable erzeugt, anstatt dem Objekt eine Eigenschaft zuzuweisen (siehe Quelltext 2.3 auf der nächsten Seite). Normalerweise werden alle lokalen Variablen aufgeräumt, wenn eine Funktion verlassen wird. In unserem Fall wären also die Variablen `_firstName` und `_lastName` nach Verlassen des Konstruktors nicht mehr definiert.

In JavaScript werden lokale Variablen jedoch erst aufgeräumt, nachdem keine Referenzen mehr darauf existieren. Bei einer gewöhnlichen Funktion passiert das, wie gewohnt, sobald sie verlassen wird. Wenn jedoch in einer Funktion eine weitere Funktion definiert wird, die die lokale Variable referenziert, dann wird die Variable so lange nicht gelöscht, wie die Funktion existiert. Dieses Verhalten wird „Closure“ genannt. Da eine lokale Variable außerdem von außen nicht sichtbar ist, verhält sie sich also genauso wie eine `private`-Variable in Java oder C.

```javascript
function Person(firstName, lastName) {
  var _firstName = firstName;
  var _lastName = lastName;

  this.getFirstName = function() {
    return _firstName;
  }

  this.getLastName = function() {
    return _lastName;
  }
}
```

```java
public class Person {
  private String firstName;
  private String lastName;

  public Person(String firstName,
                String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public String getFirstName() {
    return this.firstName;
  }

  public String getLastName() {
    return this.lastName;
  }
}
```

Quelltext 2.3: Kapselung in JavaScript und Java nach [Jung05]

2.1.4 Vererbung in JavaScript

Wie bereits erwähnt, gibt es in JavaScript keine Klassen. Somit gibt es auch keine Klassenhierarchie.

Die Vererbung wird in JavaScript über sogenannte Prototyp-Objekte realisiert. Wird auf eine Variable oder Funktion zugegriffen, die im aktuellen Objekt nicht definiert ist, dann wird die des Prototyp-Objekts verwendet. Ist sie dort auch nicht definiert, dann wird beim Prototyp-Objekt des Prototyp-Objekts gesucht und so weiter. So lassen sich ganze Vererbungshierarchien realisieren (siehe Quelltext 2.4 auf der nächsten Seite).

Das Prototyp-Objekt bleibt dabei stets unangetastet. Wenn der Wert einer Variablen geändert wird, dann wird sie im aktuellen Objekt angelegt, so dass beim nächsten Zugriff der Wert aus dem aktuellen Objekt genommen wird und nicht aus dem Prototyp-Objekt. So kann dasselbe Prototyp-Objekt für alle Objekte einer Klasse verwendet werden.

©2005 Tilman Schneider

```
function Employee(firstName,
                  lastName) {
  // Aufruf des Super-Konstruktors
  Person.call(this, firstName,
              lastName);

  // Definition und Initialisierung
  // einer eigenen Variable
  var _id = this.getFirstName()[0] +
            this.getLastName();

  // Definition einer eigenen Methode
  this.getId = function() {
    return _id;
  }
}
// Vererbung durch Prototyp-Objekt
Employee.prototype = new Person();
Employee.prototype.constructor =
  Employee;
```

```
// Vererbung durch Super-Klasse
public class Employee extends Person {
  // Definition einer eigenen Variable
  private String id;

  public Employee(String firstName,
                  String lastName) {
    // Aufruf des Super-Konstruktors
    super(firstName, lastName);

    // Initialisierung einer eigenen
    // Variable
    this.id =
      getFirstName().charAt(0) +
      getLastName();
  }

  // Definition einer eigenen Methode
  public String getId() {
    return this.id;
  }
}
```

Quelltext 2.4: Vererbung in JavaScript und Java nach [Jung05]

2.1.5 Pakete in JavaScript

Im Gegensatz zu Java kennt JavaScript keine Pakete, durch die man zusammengehörige Klassen gruppieren könnte, um Namenskonflikte zu vermeiden.

Durch die Benutzung von „Paket-Objekten" kann man das Paket-Verhalten jedoch nachbilden: Man erzeugt dazu ein Paket-Objekt und ordnet ihm diejenigen Konstruktoren zu, die man in das jeweilige Paket gruppieren will (siehe Quelltext 2.5 auf der nächsten Seite).

Imports lassen sich realisieren, indem man sich das gewünschte Klassenobjekt in einer lokalen Variablen merkt.

©2005 Tilman Schneider

```
var mypackage = new Object();

mypackage.Person = function(...) {
  ...
}
```

```
package mypackage;

public class Person {
  ...
}
```

```
function SomeClass = function() {
  // Import
  var Person = mypackage.Person;

  // Instanziierung ohne Import
  var hugo =
    new mypackage.Person(...);

  // Instanziierung mit Import
  var otto = new Person(...);
}
```

```
// Import
import mypackage.Person;

public class SomeClass {
  // Instanziierung ohne Import
  mypackage.Person hugo =
    new mypackage.Person(...);

  // Instanziierung mit Import
  Person otto = new Person(...);
}
```

Quelltext 2.5: Pakete in JavaScript und Java nach [Jung05]

2.1.6 Polymorphismus in JavaScript

JavaScript erlaubt auch Polymorphismus. Soll die Funktion einer Unterklasse ein anderes Verhalten zeigen, als die der Oberklasse, so wird sie einfach neu definiert. Will man dabei auf die Funktion der Oberklasse zugreifen, so kann man sich diese vorher in einer lokalen Variable merken, um sie dann in der überschriebenen Funktion aufzurufen (siehe Quelltext 2.6).

```
function Employee(...) {

  ...

  // super-Methode merken
  var _getLastName = this.getLastName;
  // Methode neu definieren
  this.getLastName = function() {
    // Gemerkte super-Methode
    // aufrufen
    return _getLastName.call(this)
                        .toUpperCase();
  }
}
```

```
public class Employee
       extends Person {

  ...

  // Methode überladen
  public String getLastName() {
    // super-Methode aufrufen
    return super.getLastName()
                .toUpperCase();
  }
}
```

Quelltext 2.6: Polymorphismus in JavaScript und Java nach [Jung05]

©2005 Tilman Schneider

2.1.7 Typprüfung in JavaScript

Die Typprüfung funktioniert in JavaScript analog zu Java mit dem Operator `instanceof` (siehe Quelltext 2.7).

```
var emp = new Employee("Karl",
                       "Birne");
var test1 =
  emp instanceof Employee;
var test2 =
  emp instanceof Person;
```

```
Employee emp = new Employee("Karl",
                            "Birne");
boolean test1 =
  emp instanceof Employee;
boolean test2 =
  emp instanceof Person;
```

Quelltext 2.7: Typprüfung in JavaScript und Java nach [Jung05]

2.1.8 Selbstbeschreibung in JavaScript

In Java-Script kann man durch eine einfache for-in-Schleife die Eigenschaften eines Objekts abfragen (siehe Quelltext 2.8). Dadurch ist eine Selbstbeschreibung von Objekten gegeben, die eine generische Verwendung unbekannter Objekte erlaubt.

```
var obj = ...;
var objDump = "";

// Selbstbeschreibung über
// for-in-Schleife in JavaScript
for (var subObj in obj) {
  objDump += subObj + " = "
          + obj[subObj] + "\n";
}
```

```
Object obj = ...;
String objDump = "";

// Selbstbeschreibung über
// Reflection-API in Java
Field[] fArr =
  obj.getClass().getFields();
Method[] mArr =
  obj.getClass().getMethods();
for (int i = 0; i < fArr.length; i++){
  objDump += fArr[i].getName() + " = "
          + fArr[i].get(obj) + "\n";
}
for (int i = 0; i < mArr.length; i++){
  objDump += mArr[i].getName()
          + " = (method)\n";
}
```

Quelltext 2.8: Selbstbeschreibung unbekannter Objekte

Um den Typ-Namen eines Objektes herauszufinden, muss man sich jedoch eines Tricks bedienen: Über einen regulären Ausdruck extrahiert man aus dem Quelltext des Objektes den Typnamen (siehe Quelltext 2.9 auf der nächsten Seite).

```
var emp = new Employee(...);

var typeRegex =
  /\s*function\s*((\w|_)+)\s/;
var typeName =
  emp.constructor.toString()
                .match(typeRegex)[1];
```

```
Employee emp = new Employee(...);

String typeName =
  emp.getClass().getName();
```

Quelltext 2.9: Ermittlung des Typ-Namens in JavaScript und Java nach [Jung05]

2.1.9 Was JavaScript nicht kann

Nach [Jung05] gibt es jedoch auch einige Dinge, die in JavaScript kein Äquivalent finden.

Kein `protected`

In JavaScript ist es nicht möglich, Variablen oder Methoden der Sichtbarkeit `protected` anzulegen. Alle Variablen sind entweder nur innerhalb der eigenen Klasse oder global sichtbar. Variablen, die auch von Unterklassen aus sichtbar sein sollen, muss man global sichtbar machen, bzw. durch global sichtbare Getter zugänglich machen.

Keine statische Typüberprüfung

Da es keinen Compiler gibt, fehlt auch eine statische Typüberprüfung.

In JavaScript hat jedes Objekt einen Typ. Variablen, also Referenzen auf Objekte, jedoch nicht. Man kann derselben Variable einmal eine Integer-Zahl zuweisen und später ein beliebiges anderes Objekt – ohne Type-Casts.

Der Typ wird erst geprüft, wenn versucht wird, auf ein Objekt zuzugreifen. Wenn es die entsprechende Variable oder Methode gibt, dann wird sie ausgelesen bzw. aufgerufen. Wenn nicht, dann kommt es zu einem Laufzeitfehler. Dieses Verhalten wird auch als „Duck Typing“ bezeichnet: Wenn es geht wie eine Ente und quakt wie eine Ente, dann ist es auch eine Ente[3].

[3] Engl.: „If it walks like a duck and quacks like a duck, it must be a duck“

©2005 Tilman Schneider

Teilweise inkompatible Implementierungen

Die JavaScript-Implementierungen der einzelnen Browser verhalten sich weitgehend gleich. Allerdings gibt es Unterschiede im Verhalten mancher APIs, wie beispielsweise dem DOM➦, obwohl auch diese standardisiert sind. Die Abweichungen sind jedoch lange nicht so groß, wie bei HTML zu HTML-3.0-Zeiten, aber es gibt sie.

2.1.10 Tool-Unterstützung

Bei der Entscheidung, ob eine bestimmte Technologie eingesetzt werden soll, spielt auch die Unterstützung, die von diversen Tools geboten wird, eine große Rolle. Denn welchen Nutzen hat die schönste Technologie, wenn es keine Tools dazu gibt, die den Aufwand, die Fehleranfälligkeit und die Fehlersuche in der Entwicklung reduzieren?

JavaScript-Debugger

Für Firefox gibt es einen JavaScript-Debugger, der dem Entwickler alles bietet, was man von einem Debugger erwartet: Schritt-für-Schritt-Ausführung, Breakpoints, Watches und eine Sicht auf lokale Variablen. Sogar Profiling ist möglich (siehe Abbildung 2.1 auf der nächsten Seite).

Für den Internet Explorer gibt es zwar auch einen JavaScript-Debugger, dieser ist jedoch sehr rudimentär und bietet nur das Nötigste – so wenig, dass er praktisch nicht zu gebrauchen ist.

DOM Inspector

Ein weiteres Tool, das bei der Entwicklung von dynamischen Webseiten sehr wertvoll ist, ist der DOM Inspector des Firefox-Browsers.

Mit seiner Hilfe bekommt man sowohl eine gute Übersicht auf das DOM➦, als auch alle möglichen Details zu bestimmten Knoten: Angefangen bei den DOM-Eigenschaften über die CSS-Style-Regeln bis hin zum JavaScript-Objekt, das diesen Knoten repräsentiert (siehe Abbildung 2.2 auf Seite 13).

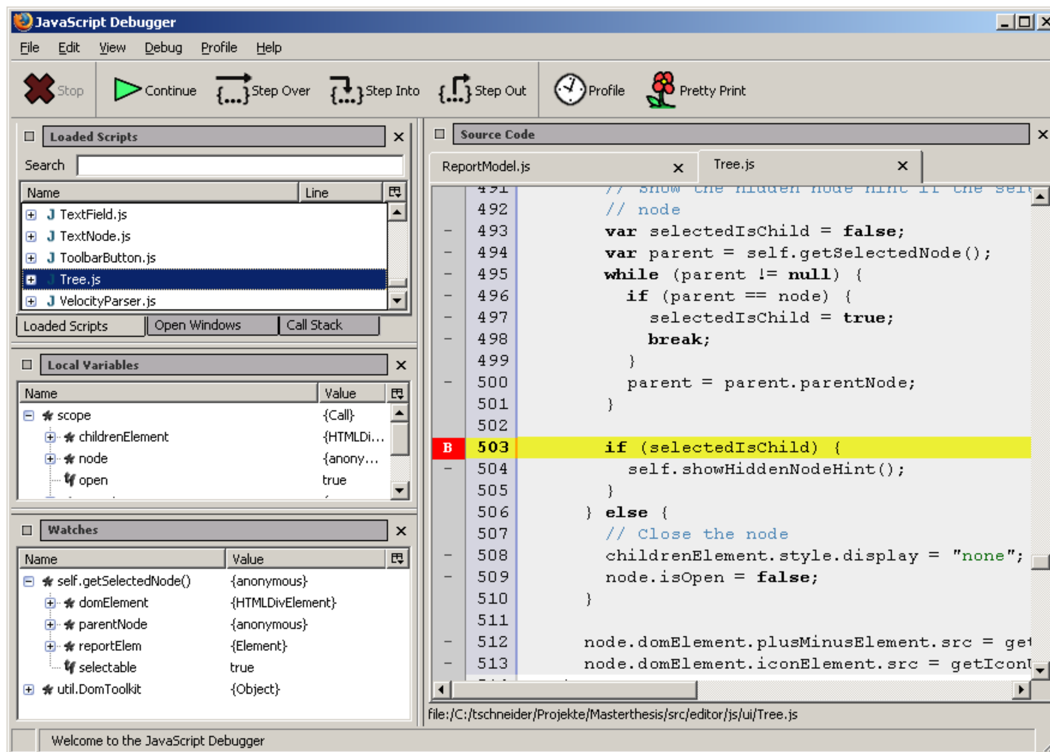

Abbildung 2.1: Der JavaScript-Debugger von Firefox

2.2 AJAX

Moderne Browser bieten folgende Techniken, die im Zusammenspiel die Entwicklung völlig neuartiger Webanwendungen erlauben:

- *JavaScript:* Mit JavaScript bieten Browser eine Scriptsprache, durch die es möglich ist, clientseitig Programmcode auszuführen.
- *DOM:* Durch die DOM-API ist es möglich, eine bereits geladene Webseite zu ändern und auf Benutzeraktionen wie Mausklicks oder Tastatureingaben zu reagieren.
- *XMLHttpRequest:* Der XMLHttpRequest erlaubt es, aus JavaScript heraus für den Benutzer unsichtbar HTTP-Anfragen zum Server zu senden. Dabei kann wahlweise einfacher Text oder XML ausgetauscht werden.

Da das DOM über JavaScript angesprochen werden kann, ist es möglich, desktopähnliche Webanwendungen zu entwickeln. Die Seite wird nur einmal geladen, danach reagiert sie auf Benutzereingaben und baut sich über das DOM selbst um. Es wird also

©2005 Tilman Schneider

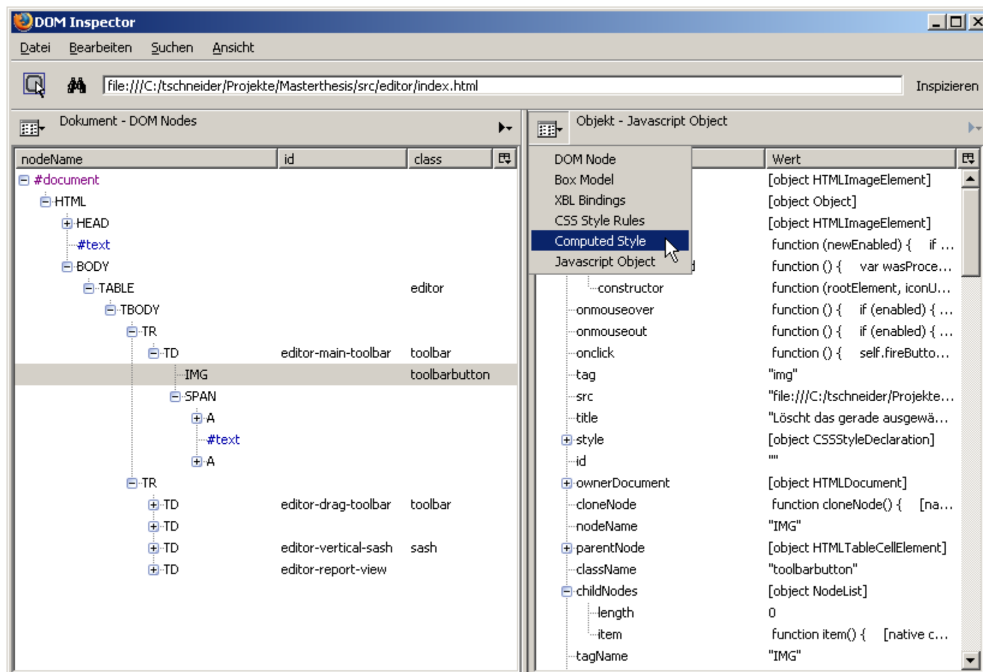

Abbildung 2.2: Der DOM Inspector von Firefox

nicht bei jeder Aktion eine Anfrage an den Server gestellt, wie es bei herkömmlichen Webanwendungen der Fall ist, und es muss auch nicht jedes Mal eine komplett neue Seite generiert werden. Die Webanwendung ist daher wesentlich flüssiger bedienbar und kann auch hoch-interaktive Techniken wie Drag-and-Drop unterstützen. Diese Verbindung von JavaScript und DOM ist auch als „dynamisches HTML“ oder kurz „DHTML“ bekannt.

Sobald Daten vom Server benötigt werden, kommt die dritte Technik ins Spiel, der XMLHttpRequest. Mit seiner Hilfe kann aus dem JavaScript-Code heraus eine HTTP-Anfrage an den Server gesendet werden, mit deren Hilfe die erforderlichen Daten geladen werden können. Da die Antwort nicht einfach vom Browser angezeigt, sondern vom JavaScript-Code interpretiert und entsprechend umgesetzt wird, kann sie völlig frei von Layoutinformationen gehalten werden. Das macht die Anwendung besser wartbar, den Server besser wiederverwendbar und spart nebenbei auch sehr viel Bandbreite.

Solche Anfragen im Hintergrund waren zwar bereits zuvor durch die Verwendung versteckter Iframes möglich, der XMLHttpRequest vereinfacht sie jedoch deutlich. Außerdem erlaubt er das synchrone Senden einer Anfrage und die Kommunikation über XML-Nachrichten. Er stellt auch die einzige Browser-übergreifende Möglichkeit dar,

XML zu parsen, d. h. eine DOM-Repräsentation eines XML-Dokuments zu bekommen[4].

Die Serveranfrage kann auch asynchron erfolgen, so dass man Anwendungen entwickeln kann, die im Hintergrund Daten anfragen, gleichzeitig jedoch noch bedient werden können. Google Maps[5] beispielsweise fordert im Hintergrund Kartenstücke an, die außerhalb des Sichtbereichs liegen. So sind diese Kartenstücke bereits geladen, wenn der Benutzer zur Seite scrollt.

Damit hat man alle Werkzeuge, die man braucht, um Clients zu bauen, die sich an Funktionalität und Bedienbarkeit nicht hinter Fat-Clients zu verstecken brauchen. Das zu dieser Technik gehörende Modewort ist „AJAX“. AJAX steht für Asynchronous Javascript and XML und bezeichnet die gerade vorgestellten desktop-ähnlichen Webanwendungen.

Der Begriff AJAX wurde laut [AJAX05] am 18. Februar 2005 im Essay [Garrett05] eingeführt. Außer dem Begriff selbst ist jedoch nichts wirklich neu daran. Alle drei Techniken, JavaScript, DOM und der XMLHttpRequest sind schon mehrere Jahre alt und wurden auch in dieser Kombination bereits zuvor genutzt.

2.3 Velocity

Velocity [Velocity05] ist eine Java-basierte Template-Engine, entwickelt im Rahmen des Jakarta Apache Projektes. Wie auch andere Template-Engines ist Velocity primär auf den Webbereich ausgerichtet und zielt darauf ab, Code und Design einer Webseite zu trennen. So können im Template Java-Methoden aufgerufen und eine einfache Flusssteuerung vorgenommen werden.

Als Template-Sprache wird die Velocity Template Language (kurz: VTL) eingesetzt. Sie erlaubt den Zugriff auf Java-Objekte, die Definition von Variablen, einfache Ablaufsteuerung mit `foreach` oder `if`, die Definition von Makros, sowie einfache mathematische Ausdrücke. In Quelltext 2.10 auf der nächsten Seite sehen Sie ein Beispiel für ein Velocity-Skript.

[4] Mozilla Firefox und Internet Explorer bieten zwar eine API, über die XML clientseitig geparst werden kann, bei anderen Browsern, wie Opera oder Safari, fehlt eine solche Möglichkeit jedoch. Hier bleibt nur der Umweg über den Server mittels XMLHttpRequest.

[5] Siehe `http://maps.google.com`

```
## Execute a query
#set ($context = $query.makeQueryContext("tutorial"))
#set ($results = $query.executeQuery($context))

<h2>Max values per customer</h2>

## Generate a table
<table>
  <header-row>
    <column>Customer</column>
    <column>Max. values</column>
  </header-row>
  #foreach ($customer in $results)
    <row>
      <column>$customer.name</column>
      <column>$customer.maxValues</column>
    </row>
  #end
</table>
```

Quelltext 2.10: Beispiel für ein Velocity-Skript, das ein XML-Dokument erzeugt

2.4 Reports

Reports sind Dokumente, die dazu dienen, Daten so aufbereitet darzustellen, dass sie zur Entscheidungsfindung herangezogen werden können. Die Daten stammen meist aus Datenbanken, andere Quellen sind jedoch ebenfalls möglich. Reports sind ein gängiges Werkzeug im Bereich der Business Intelligence, werden aber auch bei der Administration großer Unternehmensapplikationen eingesetzt.

Reports haben stets Dokument-Charakter, so dass sie prinzipiell gedruckt werden können. Sie können jedoch auch Funktionen anbieten, die über die reine Anzeige hinaus gehen. So können sie beispielsweise sortierbare Tabellen anbieten oder Hyperlinks zu anderen Reports. Mit Hilfe solcher Hyperlinks lässt sich ein sogenanntes „Drill-Down“ ermöglichen, indem in generelleren Reports an entsprechender Stelle zu detaillierteren Reports verlinkt wird. So wird eine Verbindung von der Übersicht zu den Details hergestellt, über die der Anwender schnell an die Informationen kommt, die er für seine Entscheidung braucht.

Zur Erzeugung von Reports werden meist spezielle Softwaresysteme, sogenannte „Reportgeneratoren“, eingesetzt. Der Reportgenerator „PTV-Reporter“, der Grundlage dieser Arbeit ist, wird in Abschnitt 3.1 auf der nächsten Seite vorgestellt.

Kapitel 3

Aufgabenstellung

3.1 Der PTV-Reporter

Der PTV-Reporter ist das System, für dessen Report-Vorlagen im Rahmen dieser Master Thesis ein graphischer Editor entstand.

Er ist ein recht mächtiges System, mit dessen Hilfe Reports generiert werden können, die zur regelmäßigen Berichterstattung, Überwachung oder Analyse genutzt werden können. Er wurde im Auftrag der PTV AG vom STZ-IDA entwickelt und wird bereits seit einiger Zeit produktiv eingesetzt. Details zur Funktionsweise des PTV-Reporters siehe Abschnitt 4.1 auf Seite 18.

Die Stärken des Systems liegen in den vielfältigen Möglichkeiten, Daten zu veranschaulichen. Sie können bespielsweise in Form von sortierbaren Tabellen oder Diagrammen dargestellt werden. Durch Verweise auf andere Reports, können auch fortgeschrittene Techniken wie Drill-Down umgesetzt werden. Eine weitere wichtige Eigenschaft ist, dass die Reports mit Hilfe von Skripten definiert werden, was viele Möglichkeiten bei der Gestaltung ergibt, insbesondere in Abhängigkeit von den Daten.

Was bisher noch gefehlt hat, war ein graphischer Editor, mit dessen Hilfe schnell und einfach Report-Vorlagen erstellt und verwaltet werden können. Dieser Report-Editor wurde im Rahmen dieser Master Thesis entwickelt. Die PTV AG plant, in Zukunft verstärkt Webanwendungen in AJAX-Technik zu entwickeln. Neben der gerade genannten Vereinfachung der Vorlagen-Erstellung dient der Editor daher auch gleichzeitig als Evaluierung der AJAX-Technologie.

3.2 Anforderungen an den Report-Editor

Folgende Anforderungen werden an den Report-Editor gestellt:

- *Kompatibel zum PTV-Reporter:* Er muss in der Lage sein, Velocity-Templates für den bestehenden PTV-Reporter erstellen zu können.
- *WYSIWYG:* Der Editor soll nach dem WYSIWYG-Prinzip funktionieren.

  Nun handelt es sich nicht um einen Editor für fertige Dokumente, sondern um einen Editor für Dokument-Vorlagen. Als solcher muss er nicht nur die statischen Teile eines Dokuments anzeigen, sondern auch die dynamischen Teile, also den Velocity-Code, durch dessen Ausführung aus einer Vorlage ein konkreter Report wird.

  Die statischen Teile der Report-Vorlage sollen so angezeigt werden, wie sie schließlich im fertigen HTML-Report aussehen. Da die HTML-Version gewonnen wird, indem auf die Ausgabe des Velocity-Templates ein XSLT-Stylesheet angewendet wird, muss der Editor in der Lage sein, das XSLT-Stylesheet auf die statischen Teile des Templates anzuwenden. Die dynamischen Template-Teile sollen dabei von Platzhaltern repräsentiert werden.

Folgende Eigenschaften wären ebenfalls wünschenswert, sind jedoch nicht zwingend erforderlich:

- *Umschalten zwischen XSLT-Stylesheets:* Bei der Erstellung der HTML-Ausgabe kann man beim PTV-Reporter zwischen verschiedenen XSLT-Stylesheets wählen, die verschiedene HTML-Ausgaben produzieren können. Es wäre wünschenswert, wenn auch der Editor in der Lage wäre, auf andere XSLT-Stylesheets umzuschalten.
- *Umschalten zwischen Entwurf und fertiger Ausgabe:* Es sollte ein schnelles Umschalten zwischen Entwurfs- und Laufzeitmodus möglich sein, um den Report zu testen.
- *Universeller Editor:* Der Editor sollte auch für andere Template-Sprachen anpassbar sein, wie z. B. JSPs.

©2005 Tilman Schneider

Kapitel 4

Analyse

4.1 Der PTV-Reporter

Wie in Abschnitt 3.1 auf Seite 16 bereits erwähnt, ist der PTV-Reporter ein System zur Generierung von Reports, für das im Rahmen dieser Master Thesis ein Editor entwickelt wurde. Seine Funktionsweise soll im Folgenden kurz vorgestellt werden.

Die Erzeugung eines Reports findet auf dem Server in einer J2EE-Umgebung statt und erfolgt in zwei Schritten. Im ersten Schritt wird ein Velocity-Skript ausgeführt. Dabei wird ein formatierungsunabhängiges Report-XML-Dokument erstellt. Die durch den Report veranschaulichten Daten werden durch Queries abgefragt und in die XML-Ausgabe eingearbeitet. Im zweiten Schritt wird das Report-XML-Dokument über eine XSLT-Transformation in das Ausgabeformat, z. B. HTML oder PDF, konvertiert (siehe Abbildung 4.1 auf der nächsten Seite).

Das Velocity-Skript dient also als Report-Vorlage. Es bestimmt, welche Daten in den Report gelangen und wie sie dargestellt werden sollen, also ob sie in einer Tabelle, Liste oder in einem Diagramm erscheinen sollen (siehe Quelltext 4.1 auf Seite 20).

Die Queries, die zur Abfrage der eigentlichen Daten dienen, sind in gesonderte Dateien ausgelagert. Dadurch wird die Report-Vorlage lesbarer und dieselbe Query kann in mehreren Reports verwendet werden. Momentan werden sowohl SQL- als auch MVCQL-Queries unterstützt. In der Report-Vorlage wird eine Query durch den Befehl `$results = $query.executeQuery("queryname")` ausgeführt.

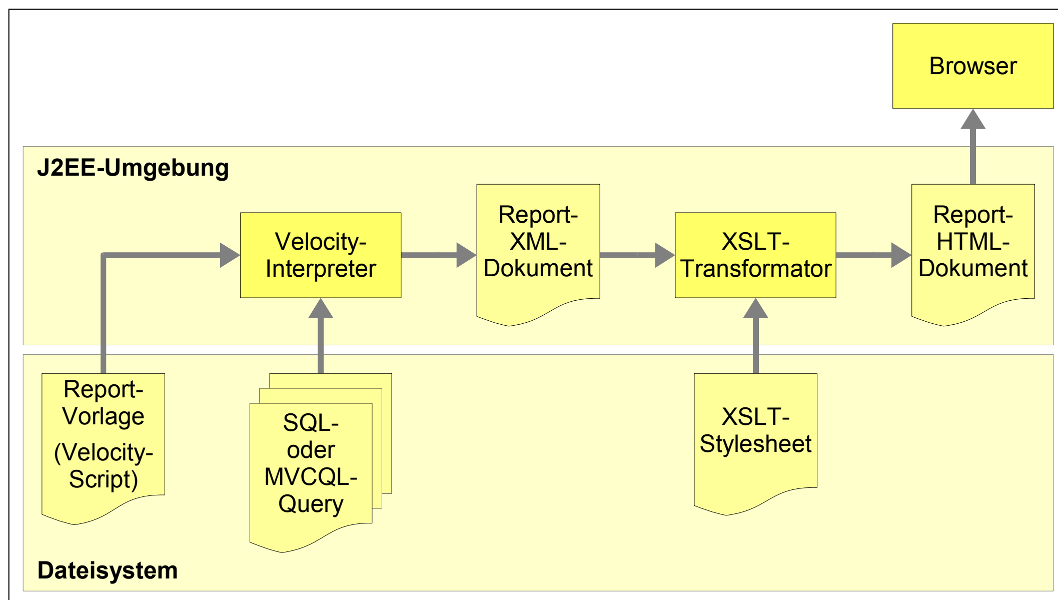

Abbildung 4.1: Flussdiagramm – Erstellung eines Reports im PTV-Reporter

Das bei der XSLT-Transformation eingesetzte XSLT-Stylesheet ist dafür verantwortlich, ein Report-XML-Dokument in ein vom Benutzer lesbares Format zu wandeln. Durch die Verwendung verschiedener XSLT-Stylesheets kann derselbe Report in viele verschiedene Formate gewandelt werden. Beispiele hierfür sind HTML, für den Druck optimiertes HTML oder PDF.

4.2 Der alte Editor

Momentan wird ein Report folgendermaßen erstellt bzw. geändert: Der Entwickler sucht das entsprechende Velocity-Template im Dateisystem auf dem Server und bearbeitet es in einem Texteditor.

Als Editor kommt dabei das Eclipse-Plugin Velocity Web Edit [WebEdit05] zum Einsatz. Dieses bietet Syntaxhervorhebung, eine Gliederungsansicht[1], eine Syntaxprüfung, sowie eine automatische Vervollständigung von Schlüsselwörtern oder bereits genutzten Variablen oder Makros (siehe Abbildung 4.2 auf Seite 21).

[1] Engl. „outline“: Gliederung

```
<h1>Ein Report</h1>

## Ausgabe von Daten in einer Tabelle
#set($results = $query.executeQuery("customers"))
<table class="ptv-table">
  <header-row>
    <column>Name</column>
    <column>Adresse</column>
    <column>Umsatz</column>
  </header-row>
  ## Für jeden Datensatz in der Ergebnismenge der Query
  ## wird eine Tabellenzeile erzeugt
  #foreach ($customer in $results)
  <row>
    <column>$customer.name</column>
    <column>$customer.address</column>
    <column>$customer.country</column>
    <column format="number">$customer.turnover</column>
  </row>
  #end
</table>

## Erzeugung einer Datenstruktur für ein Diagramm
#set($chartData = $chart.createTableDataset())
#set($chartParams = $chart.createChartParameters())

## Füllen der Datenstruktur mit den Ergebnissen einer Query
#set($results = $query.executeQuery("customers"))
#foreach ($customer in $results)
  $chartData.addValue($customer.turnover, $customer.name, "")
#end

## Ausgabe des Diagramms
#set($chartInfo = $chart.createPieChart($chartData, $chartParams))
<chart href="$chartInfo.hRef"
       width="$chartInfo.ImgWidth" height="$chartInfo.ImgHeight"/>
```

Quelltext 4.1: Beispiel für eine Report-Vorlage

4.2.1 Bewertung

Das Eclipse-Plugin hat den Vorteil, dass es die Entwicklung der Code-Teile sehr gut unterstützt. Auf der anderen Seite bietet es jedoch gar keine Unterstützung bei den XML-Report-Tags.

Dieser Umstand führt in der Praxis auch zu Problemen: Die Elemente eines Reports (Tabellen, Tabellenzeilen, Diagramme, usw.) können auch vom Velocity-Code aus über eine API erzeugt werden. Dabei werden die XML-Elemente nicht direkt in der Report-Vorlage definiert, sondern es wird bei einem Velocity-Objekt eine Methode aufgerufen, die den XML-Code zur Laufzeit ausgibt. Da Code-Teile mit dem Eclipse-Plugin leich-

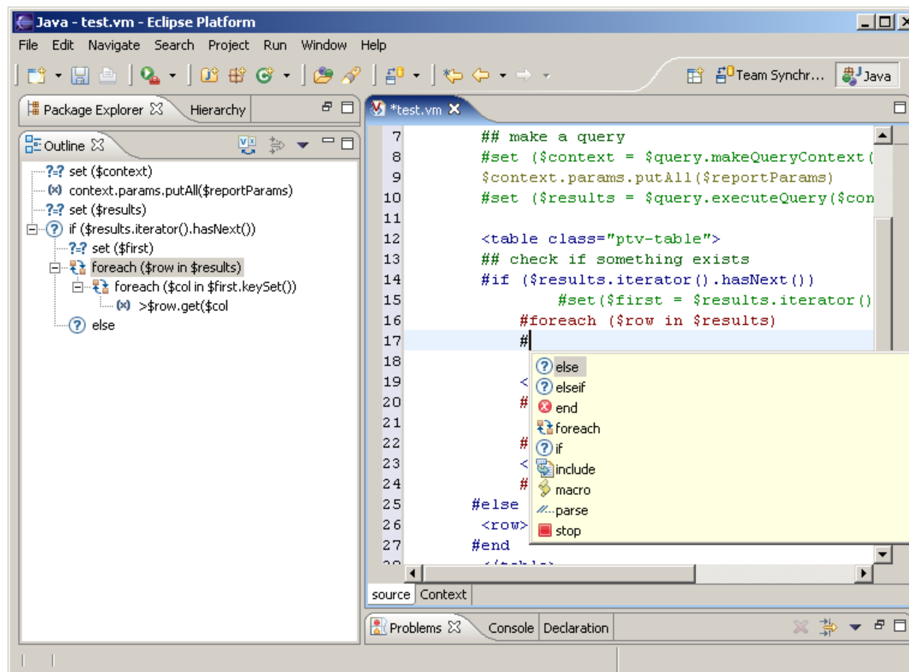

Abbildung 4.2: Das Eclipse-Plugin Velocity Web Edit

ter zu entwickeln sind, nutzen viele Entwickler lieber die API statt der direkten XML-Schreibweise. Dadurch wird jedoch die Lesbarkeit der Report-Vorlage, gerade für weniger geübte Benutzer, wesentlich schlecher.

4.3 Der neue Editor

Um die Erwartungen seitens der PTV AG zum neuen Editor abschätzen zu können, wurde mit den Verantwortlichen auf PTV-Seite ein Interview durchgeführt. Dabei wurde gefragt, in welchem Umfeld der neue Editor eingesetzt werden soll, mit welchem Benutzerprofil zu rechnen ist, wie der derzeitige Ablauf aussieht und was man sich vom neuen Editor verspricht.

Das Benutzerprofil wird in Abschnitt 4.5 auf Seite 23 näher beleuchtet, der derzeitige Ablauf in Abschnitt 4.2 auf Seite 19. Die übrigen Ergebnisse des Interviews sollen nun vorgestellt werden.

Die Tools, die später zusammen mit dem Report-Editor verwendet werden sollen, sind in das jeweilige Intranet-Portal des Kunden bzw. der PTV AG integriert. Der neue Editor

sollte daher selbst in Webtechnologie erstellt sein, damit er in ein Web-Portal integriert werden kann.

Da das Aussehen stets an das jeweilige Portal angepasst wird, gibt es keine Styleguides, welche die Optik der Oberfläche bestimmen. Der neue Editor sollte also Möglichkeiten bieten, das Aussehen leicht anpassbar zu machen.

Es ist damit zu rechnen, dass der neue Editor in etwa einmal monatlich pro Benutzer genutzt werden wird. Bei dieser eher niedrigen Frequenz ist bei der Umsetzung darauf zu achten, dass der Editor eine recht gute Selbstbeschreibungsfähigkeit und Lernförderlichkeit besitzt.

Auf die Frage, aus welchen Gründen der neue Editor am meisten gewünscht ist, wurden folgende Punkte genannt:

- Es fehlt ein einfaches Erstellen. Gerade nicht-technische Anwender haben mit der Erstellung der Velocity-Skripte Probleme. Aber auch Entwickler wünschen eine weniger fehleranfällige und zeitraubende Möglichkeit, Reports zu erstellen.
- Es fehlt eine WYSIWYG-Ansicht, was sehr viel Ausprobieren erfordert. Beim alten System merkt der Benutzer erst bei der Ausführung, wie sein Report aussieht. Das zieht häufige Editier/Ausführ-Zyklen nach sich. Ein Ziel des neuen Editors sollte sein, ein direkteres Feedback zu geben, so dass sich der Benutzer nervenaufreibendes, ständiges Ausprobieren ersparen kann und schneller zu einem befriedigenden und ansehnlichen Ergebnis kommt.
- Der neue Editor soll die Entwicklungsgeschwindigkeit, Qualität und Akzeptanz der Report-Vorlagen bzw. des PTV-Reporters steigern.

4.4 Use-Cases

Für das gesamte System gibt es nur einen Akteur, den Vorlagen-Autor. Er hat die Aufgabe, sich um die Pflege der Report-Vorlagen zu kümmern.

Für das System gibt es folgende Anwendungsfälle:

- *Vorlagenbestand anschauen:* In einer Übersicht wird gezeigt, welche Vorlagen es gibt.

- *Vorlage editieren:* Der Akteur bekommt die Aufgabe, eine Vorlage zu verändern. Dazu öffnet er die Vorlage im Report-Editor, ändert sie entsprechend ab und speichert dann das Ergebnis. Dies ist der Hauptanwendungsfall. Eine genaue Analyse der dazu nötigen Einzelschritte findet in Abschnitt 5.1 auf Seite 28 statt.
- *Vorlage löschen:* Der Akteur wählt eine Vorlage und gibt den Befehl zum Löschen. Nach einer Sicherheitsabfrage wird die Vorlage gelöscht.
- *Vorlage umbenennen:* Der Akteur wählt eine Vorlage und bekommt ein Textfeld angezeigt, in dem er den neuen Namen eingeben kann. Sobald er fertig ist, bestätigt er den neuen Namen oder er bricht ab.
- *Vorlage kopieren:* Der Akteur wählt eine Vorlage und gibt den Befehl zum Kopieren. Daraufhin wird eine Vorlage mit demselben Namen erstellt, plus dem Präfix „Kopie von". So hat dieser Anwendungsfall ein ähnliches Verhalten, wie es z.B. vom Windows Explorer bereits bekannt ist.
- *Neue Vorlage erstellen:* Der Akteur gibt den Befehl, dass eine neue Vorlage erstellt werden soll. Daraufhin wird eine leere Vorlage mit dem Namen „Neue Report-Vorlage" erstellt. Auch dieses Verhalten soll die Erwartungskonfomität erhöhen, indem sie sich ähnlich verhält wie der Windows Explorer.

4.5 Die Benutzer

Um eine benutzerfreundliche Anwendung zu entwerfen, muss man zunächst analysieren, wer die Benutzer sind und welche Aufgaben, Fähigkeiten und Wünsche sie haben. Dabei kristallisieren sich meist verschiedene Benutzergruppen heraus, in die Benutzer mit ähnlichem Profil eingeteilt werden können.

Damit ein Entwickler sich besser in die Sichtweise der Benutzer hineinversetzen kann, hat Alan Cooper das Prinzip der „Personas" – auch „Persona Design" genannt – entwickelt[2]. Eine Persona ist eine Art Steckbrief, der einen fiktiven, repräsentativen Benutzer einer Benutzergruppe beschreibt. Dieser Steckbrief soll möglichst persönlich sein, d. h. zumindest mit einem Namen und einem Foto versehen sein, so dass es leicht fällt, sich mit diesem Beispielbenutzer zu identifizieren. Außerdem enthält der Steckbrief weitere Details zu diesem Benutzer, insbesondere seine Aufgaben, Fähigkeiten

[2] Siehe [Cooper95]

und Wünsche. Man kann die Beschreibung einer Persona sehr detailliert gestalten. Prinzipiell gilt: Je lebensechter die Persona wird, desto leichter kann man sich in ihre Lage versetzen.

Für den Report-Editor wurden vier Benutzergruppen identifiziert, welche durch die folgenden Personas beschrieben werden:

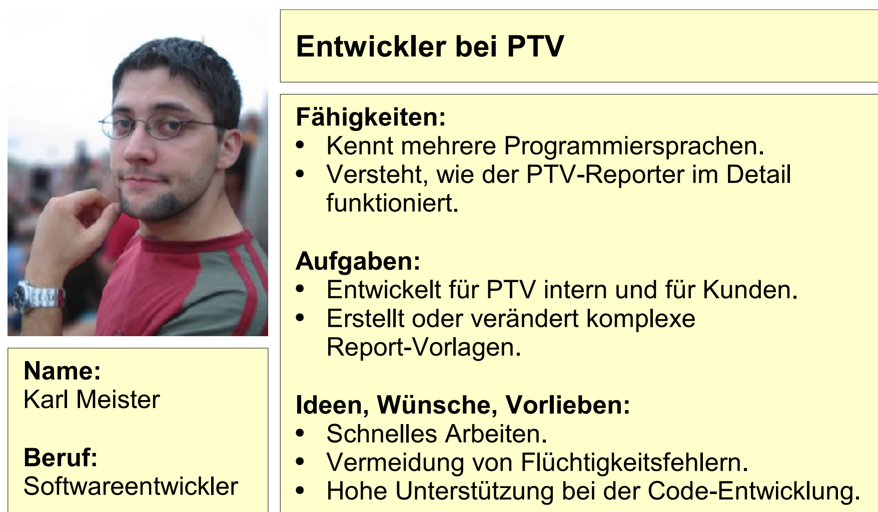

Abbildung 4.3: Persona – Entwickler bei PTV

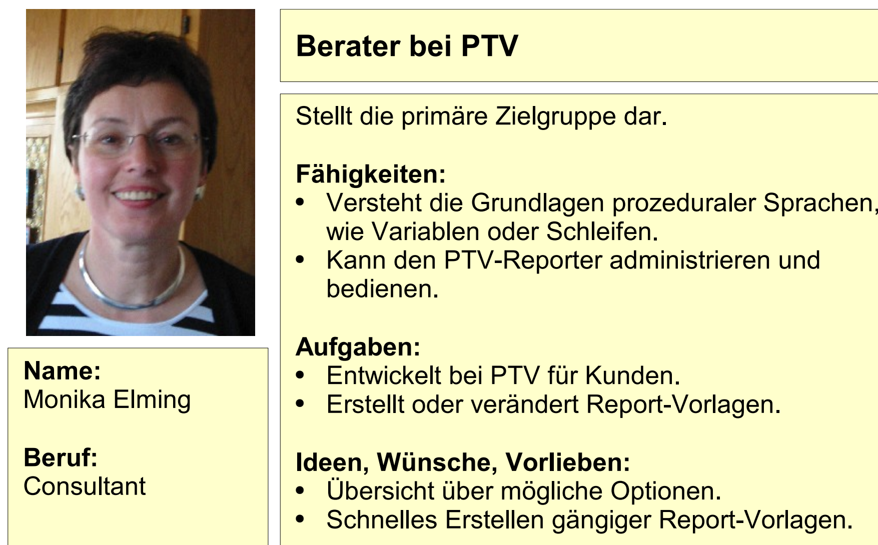

Abbildung 4.4: Persona – Berater bei PTV

Abbildung 4.5: Persona – Entwickler beim Kunden

Abbildung 4.6: Persona – Benutzer beim Kunden

4.6 Ergebnis der Editor-Recherche

Um zu prüfen, ob eine Eigenentwicklung notwendig ist, wurde eine Recherche nach bereits vorhandenen Report-Editoren durchgeführt.

Diese hat ergeben, dass es vier große Produkte im Bereich Reporting gibt:

- „Report Net“ von Cognos, siehe [RepNet05].
- „Crystal Reports“ von Business Objects, siehe [CrysRep05].
- „SAS 9“ von SAS, siehe [SAS05].
- „Microsoft SQL Server 2000 Reporting Services“ von Microsoft, siehe [RepServ05].

Diese Produkte bieten natürlich Oberflächen, mit deren Hilfe man sich Report-Vorlagen zusammenklicken kann. Diese kommen jedoch aus mehreren Gründen nicht in Frage: Zum einen fallen für die Produkte Lizenzkosten an, was für eine Integration in Systeme, die an Kunden ausgeliefert werden sollen, nicht tragbar wäre. Zum anderen sind sie an die entsprechenden Produkte gebunden und alles andere als universell einsetzbar. Sie könnten also technisch gar nicht von ihrem System losgelöst betrieben werden, geschweige denn dazu genutzt werden, Velocity-basierte Vorlagen zu erzeugen. Und es käme für die PTV AG auch nicht in Frage, einen anderen Report-Generator einzusetzen.

An den Report-Editor werden sehr spezielle Anforderungen gestellt. Schon alleine die Tatsache, dass eine Vorschau auf ein Velocity-Script gezeigt werden soll, welches XML erzeugt, das über XSLT in HTML umgewandelt wird, macht den Editor so speziell, dass man um eine Eigenentwicklung nicht herum kommt.

Trotzdem wurde eine weitergehende Recherche vorgenommen, um auszuschließen, dass es vielleicht doch einen solchen Editor gibt. Die Vermutung wurde jedoch bestätigt: Es gibt keinen Editor auf dem Markt, der diesen speziellen Anforderungen genügt.

4.7 Aus der Analyse gewonnene Anforderungen

Bei der Analyse haben sich eine Reihe von Anforderungen ergeben, die über die in der Aufgabenstellung gegebenen Anforderungen hinausgehen. Diese Anforderungen sollen nun noch einmal konzentriert zusammengefasst werden.

Zwingende Anforderungen:

- *Anpassbares Aussehen:* Das Aussehen muss leicht an Styleguides des Kunden anpassbar sein.

Wünschenswerte Anforderungen:

- *Nutzung von Webtechnologie:* Die Tools, die später zusammen mit dem Report-Editor verwendet werden sollen, sind in das jeweilige Intranet-Portal des Kunden bzw. der PTV AG integriert. Es wäre daher wünschenswert, wenn auch der Report-Editor in Webtechnologie erstellt wäre, so dass eine Integration in eine andere Webanwendung möglich ist.

- *Mächtiger Velocity-Editor:* Die Stärken des aktuellen Editors liegen bei der großen Unterstützung beim Entwickeln der Code-Teile einer Report-Vorlage. Um gerade bei den erfahrenen Benutzern die Akzeptanz des neuen Editors zu gewährleisten, wäre es gut, wenn auch der neue Editor wenigstens grundlegende Funktionen, wie z. B. Syntax-Highlighting, bieten würde.

- *Gute Selbstbeschreibungsfähigkeit und Lernförderlichkeit:* Da die Nutzungshäufigkeit mit einmal pro Monat eher niedrig ist, sollte der Editor ein klar verständliches Bedienkonzept bieten und weniger von Expertenfunktionen geprägt sein.

©2005 Tilman Schneider

Kapitel 5

Entwurf

5.1 Benutzerführung und Bedienkonzepte

5.1.1 Allgemeine Überlegungen

Neben der WYSIWYG-Ansicht – im Folgenden Vorschau genannt – sollte dem Benutzer auch eine Gliederungsansicht gezeigt werden, die ihm den logischen Aufbau der Report-Vorlage zeigt. Da ein Report letztendlich ein XML-Dokument ist, also eine Baumstruktur besitzt, eignet sich dafür eine Baumansicht (TreeView).

Außerdem muss der Benutzer die Möglichkeit haben, die Eigenschaften des selektierten Report-Knotens einzusehen und zu bearbeiten.

Daraus ergeben sich drei Hauptelemente: Eine Vorschau des Dokuments, eine Gliederungsansicht und eine Detail- oder Eigenschaftenansicht. Diese drei Elemente werden in der Praxis oft durch eine Master-Detail-Anordnung realisiert. Klassisches Beispiel hierfür ist ein E-Mail-Client: In einer Gliederungsansicht sieht man die einzelnen Ordner, daneben wird der Inhalt des selektierten Ordners in einer Tabelle gezeigt, rechts unten die selektierte E-Mail. Weitere Beispiele für diese Anordnung sind Dateiverwaltungsprogramme wie der Windows Explorer oder IDEs wie Eclipse (siehe Abbildung 5.1 auf der nächsten Seite).

Man kann also davon ausgehen, dass dieses Schema dem Benutzer bereits bekannt ist. Es zu übernehmen, unterstützt daher die Erwartungskonformität und die Lernförderlichkeit.

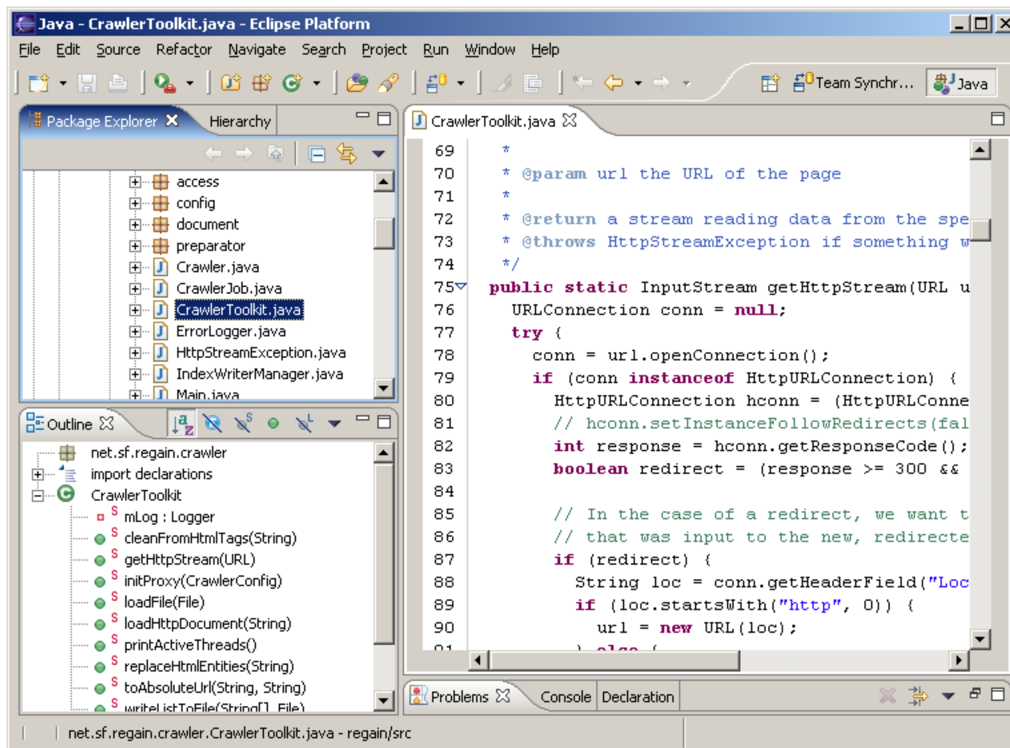

Abbildung 5.1: Master-Detail-Ansicht bei Eclipse

5.1.2 Hinzufügen und Verschieben von Report-Knoten

Für das Hinzufügen von Elementen in ein Dokument gibt es zwei gängige Methoden: Entweder die Anwendung zeigt einen Cursor und fügt an der damit markierten Stelle neue Objekte ein, sei es durch eine Tastatureingabe, durch Drücken von Knöpfen in der Symbolleiste oder durch die Auswahl von Menüpunkten. Dieses Schema ist vor allem in der Textverarbeitung zu finden.

Die zweite Methode ist das Einfügen per Drag-and-Drop. Man zieht ein Element aus einer Drag-Bar und lässt es an der Stelle fallen, an der man es im Dokument einfügen möchte. Dieses Schema wird beispielsweise vom Illustrationsprogramm Microsoft Visio genutzt.

Eine XSLT-Transformation ist zwar schnell, aber zu träge, um ein Tippen von Text in Echtzeit zu unterstützen. Text kann daher nicht direkt in der Vorschau eingegeben werden. Es wäre nicht erwartungskonform, wenn man in der Vorschau einen Cursor zeigen würde, während der Text an anderer Stelle eingegeben werden muss.

©2005 Tilman Schneider

Das Drag-and-Drop-Verfahren ließe sich auch für die Gliederungsansicht nutzen, so dass man neue Elemente entweder in die WYSIWYG-Ansicht oder in die Gliederungsansicht ziehen kann. Cursor in einer Gliederungsansicht sind absolut unüblich und würden wohl nicht verstanden werden.

Aus den oben genannten Gründen sollte das Einfügen über Drag-and-Drop erfolgen und nicht über einen Cursor. Das Verschieben von Elementen wird in praktisch allen Anwendungen, die Dokumente bearbeiten, durch Drag-and-Drop vorgenommen. Es würde also das Bedienkonzept vereinheitlichen, wenn auch das Einfügen von Elementen in gleicher Weise funktionieren würde.

5.1.3 Verändern von Report-Knoten

Das Verändern von Report-Knoten wird über die Eigenschaftenansicht ermöglicht. Wie bereits gesagt, werden dort die Eigenschaften des gerade selektierten Knotens aufgelistet. Es entspricht der Selbstbeschreibungsfähigkeit, wenn die Eigenschaften an dieser Stelle nicht nur angezeigt werden, sondern auch verändert werden können.

Wie die einzelnen Eigenschaften zu ändern sind, muss für jeden Knoten-Typ einzeln entschieden werden. Bei einem Textknoten würde sich beispielsweise anbieten, die gesamte Eigenschaftenansicht als ein Texteingabefeld zu nutzen. Bei anderen Knoten kann für jede Eigenschaft ein Textfeld oder eine Combo-Box angezeigt werden, je nachdem, ob es für diese Eigenschaft typische Werte gibt oder nicht.

Da bei einer Änderung eine Aktualisierung der Gliederungsansicht und eine XSLT-Transformation für die Vorschau durchgeführt werden muss, werden geänderte Einstellungen nicht sofort übernommen, sondern erst, wenn ein anderer Knoten selektiert wird bzw. durch explizites Drücken eines Aktualisieren-Knopfes. So wird ein flüssiges Arbeiten ermöglicht. Außerdem kommt es nicht zu unerwarteten Zwischenergebnissen, wenn der Benutzer mehrere Eigenschaften ändern will und die Vorschau nach der Änderung einer Eigenschaft seltsam aussieht, weil erst die Summe aller Änderungen das erwartete Ergebnis bringt.

5.1.4 Der Papierentwurf

Nach [DesEss97] bietet es sich an, beim Entwurf einer Oberfläche zunächst einen Papierentwurf zu zeichnen. Dieser kann schnell erstellt und abgeändert werden. Der Ent-

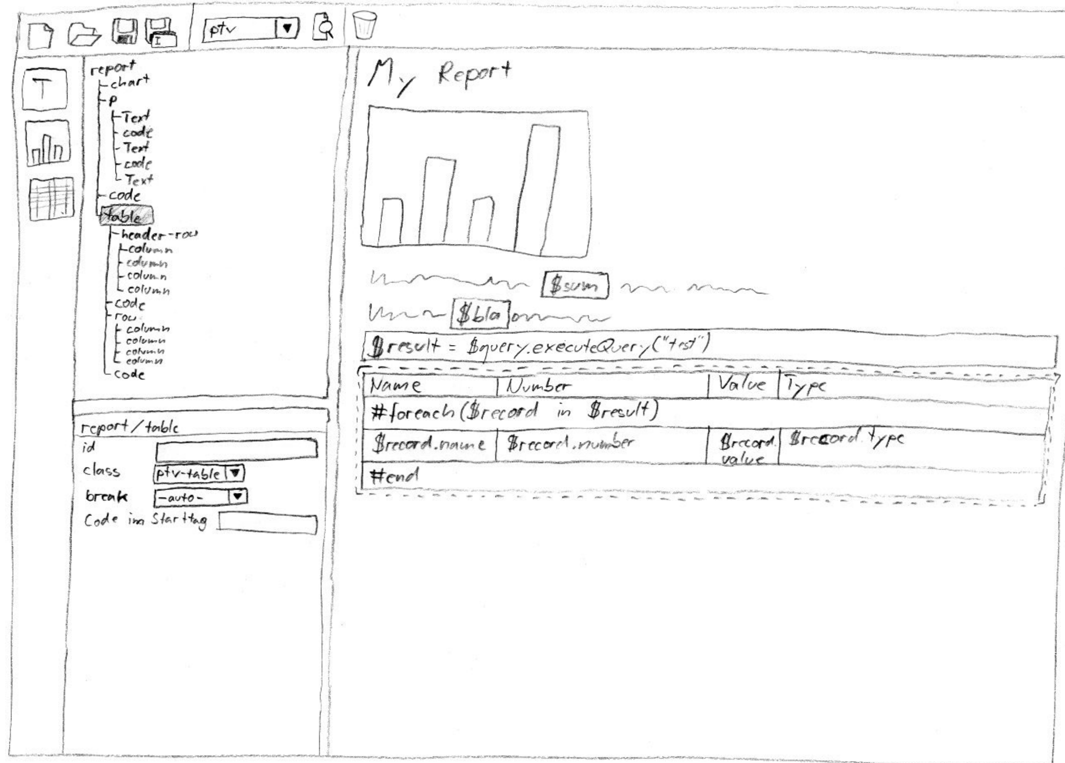

Abbildung 5.2: Entwurfsskizze der Oberfläche

wickler kann damit selbst noch einmal die Bedienung durchspielen oder er kann den Entwurf anderen vorlegen und Usability-Tests durchführen.

Im Gegensatz zu einem Prototyp kann der Papierentwurf schneller erstellt werden und er macht beim Kunden nicht den Eindruck, dass die Anwendung schon so gut wie fertig ist. Der Kunde soll wissen, dass sich die Anwendung noch in der Planung befindet und er soll verstehen, dass man noch vieles ändern kann. Bei einem Prototyp besteht die Gefahr, dass der Kunde von der perfekten Erscheinung geblendet wird und strukturelle Fehler nicht erkennt, oder sich nicht traut, große Änderungen vorzuschlagen. Diese Fehler müssen dann oft im Nachhinein teuer ausgeglichen werden.

Im Papierentwurf des Report-Editors (siehe Abbildung 5.2) sieht man die drei Bereiche: Die Gliederungsansicht links oben, die Eigenschaftenansicht links unten und die Vorschau zur Rechten, jeweils durch einen Schiebebalken voneinander getrennt, so dass der Benutzer einstellen kann, welcher Bereich wie viel Platz einnehmen soll. Am linken Rand ist die Drag-Bar zu sehen, mit deren Hilfe neue Elemente in den Baum oder in die Vorschau gezogen werden können. Im Entwurf sind nur drei dieser Elemente angedeu-

tet, im realen Editor gibt es natürlich eine größere Auswahl. Oben ist die Symbolleiste mit den Aktionen zu sehen, die zur Zeit möglich sind.

Im Entwurf ist auch zu sehen, wie die Selektion eines Knotens vonstatten geht: Der gerade selektierte Knoten wird in der Gliederungsansicht unterlegt und in der Vorschau durch einen gestrichelten Rahmen hervorgehoben. So kann der Benutzer sofort die Zuordnung von Aussehen und Struktur herstellen. Außerdem werden in der Eigenschaftenansicht die Eigenschaften des Knotens angezeigt. Dort ist auch sein Pfad zu sehen – hier: „report/table“.

5.1.5 Prüfung des Papierentwurfs

Zur Prüfung des Papierentwurfs wurden mehrere Usability-Tests durchgeführt. Den Probanden wurde der Entwurf vorgelegt und sie wurden gebeten, einige Aufgaben durchzuführen, wobei sie sich vorstellen sollten, der Entwurf sei eine fertige Anwendung. Dabei sollten sie alles laut aussprechen, was sie tun und was sie denken.

Die Tests haben ergeben, dass das Bedienkonzept sehr gut verstanden wird. Lediglich ein kleines Detail ist negativ aufgefallen. Für das Löschen von Report-Knoten war zunächst ein Knopf mit einer Mülleimer-Ikone in der Symbolleiste vorgesehen. Man selektiert das zu löschende Element und drückt dann auf diesen Knopf.

Bei den Tests kam jedoch heraus, dass manche Benutzer den zu löschenden Knoten per Drag-and-Drop auf den Mülleimer-Knopf ziehen wollten, wobei nichts passierte. Dieses Verhalten ist aus folgenden Gründen verständlich: Zum einen funktioniert auch das Hinzufügen und das Verschieben von Knoten per Drag-and-Drop, zum anderen ist das Ziehen von Objekten in den Papierkorb bereits eine gelernte Metapher. Der Entwurf wurde deshalb dahingehend geändert, dass der Knopf sowohl gedrückt werden, als auch Ziel eines Drags sein kann.

5.2 Wahl der Basistechnologie

Obwohl der PTV-Reporter dafür ausgelegt ist, verschiedene Ausgabeformate zu unterstützen, kommt momentan nur HTML zum Einsatz. Selbst wenn in Zukunft noch weitere Formate dazukommen werden, beispielsweise das PDF-Format, wird HTML das Hauptformat bleiben.

Um eine WYSIWYG-Anzeige zu erhalten, ist also ein Browser nötig. Manche Widget-Frameworks bieten zwar auch Widgets, die HTML darstellen können, diese sind jedoch auf die Anzeige einfacher HTML-Dokumente beschränkt. Damit die WYSIWYG-Anzeige auch wirklich so aussieht wie der fertige Report, muss sichergestellt werden, dass alle HTML-Konstrukte, die in einem Report vorkommen können, unterstützt werden und auch gleich aussehen. Dies kann nur ein Browser leisten.

Daraus ergeben sich folgende Möglichkeiten: Entweder man integriert den Browser als Widget in eine Stand-Alone-Applikation oder man realisiert die Anwendung im Browser.

Wenn die Anwendung im Browser ausgeführt werden soll, dann muss der größte Teil der Anwendungslogik clientseitig laufen – Eine reine HTML-Lösung wäre für eine derart interaktive Anwendung zu träge, da dann bei jeder Benutzeraktion Kommunikation mit dem Server stattfinden müsste. Außerdem wären komplexere Benutzer-Interaktionen wie Drag-and-Drop nicht möglich.

Die Realisierung im Browser kann auf zwei Arten erfolgen: Entweder bettet man über Browser-Plugins Teile der Anwendung in den Browser ein oder man lässt mit Hilfe von dynamischem HTML die gesamte Anwendung vom Browser darstellen.

Daraus ergeben sich drei Lösungsmöglichkeiten: Zum einen die Fat-Client-Lösung mit einem Browser als Widget, zum zweiten eine Rich-Client-Lösung mit Plugins und schließlich eine Rich-Client-Lösung mit dynamischem HTML. Diese Möglichkeiten werden im folgenden analysiert und gegeneinander abgewogen.

5.2.1 Fat Client: Browser als Widget

Einen Browser als Widget in eine Applikation einzubinden ist über verschiedene Technologien möglich:

- In einer Windows-Anwendung könnte der Internet Explorer als Active-X-Komponente eingebunden werden.
- In einer Desktop-Anwendung könnte die Gecko-Engine als Widget eingebunden werden.
- Die Java-Bibliothek SWT bietet ein Browser-Widget an. Unter Windows wird dabei wahlweise der Internet Explorer oder die Gecko-Engine eingebunden, unter Linux und MacOS wird die Gecko-Engine unterstützt.

Vorteile:

- Für die Anwendung können die vollen Möglichkeiten der entsprechenden Basistechnologie genutzt werden, d. h. eine umfangreiche Auswahl an Widgets, eine große Auswahl an Bibliotheken und voller Zugriff auf den Client-Rechner.

Nachteile:

- Zwischen Anwendung und WYSIWYG-Ansicht besteht ein Technologiebruch. Dadurch sind bei starker Interaktion größere Probleme zu erwarten, wie z. B. bei Drag-and-Drop von der Anwendung in die WYSIWYG-Ansicht und umgekehrt. Außerdem lassen sich auch optische Inkonsistenzen der Oberfläche nur schwer vermeiden.
- Die üblichen Fat-Client-Probleme: Die Installation muss aktuell gehalten werden. Dazu muss entweder ein Update-Check oder eine automatische Aktualisierung eingebaut werden.
- Bei einer Windows-Anwendung legt man sich auf Windows als Plattform fest.

5.2.2 Rich Client: Browser plus Plugin

Mit Hilfe des Java-Plugins oder des Flash-Plugins können Java-Applets bzw. Flash-Objekte in eine Webseite eingebunden werden. Diese können die sie umgebende Seite über das DOM verändern.

Die komplexen Bedienelemente könnten in Plugin-Objekte ausgelagert werden, welche über das DOM in der Lage wären, die WYSIWYG-Ansicht zu aktualisieren.

Vorteile:

- Es können die komplexen Bedienelemente genutzt werden, die von der Plugin-Technologie bereitgestellt werden. Im Falle des Java-Plugins kann z. B. Swing verwendet werden.
- Kann in ein Intranet-Portal eingebunden werden. Das ist vor allem deshalb nützlich, weil auch die fertigen Reports in ein Intranet-Portal eingebunden werden können. Damit sind dann Funktionen wie „diesen Report bearbeiten“ möglich, wie sie z. B. in Wikis zum Einsatz kommen.

Nachteile:

- Wie auch bei der Einbindung eines Browsers als Widget besteht hier ein Technologiebruch zwischen der Anwendung und der WYSIWYG-Ansicht, der starke Interaktion und eine konsistente GUI problematisch werden läßt.
- Das Plugin muss installiert und aktuell gehalten werden.

5.2.3 Rich Client: Dynamisches HTML

Durch Verwendung von AJAX[1] könnte sowohl die WYSIWYG-Ansicht als auch die restliche Anwendung ein HTML-Dokument sein, das mit Hilfe von JavaScript auf Eingabeereignisse reagiert und sich entsprechend verändert.

Vorteile:

- Kein Technologiebruch zwischen Anwendung und WYSIWYG-Ansicht. Damit keine größeren Probleme mit Interaktion und GUI-Konsistenz.
- Keine Probleme mit Installation und Aktualisierung: Der Browser holt sich immer die aktuellste Version.
- Kann, wie auch die Plugin-Lösung, in ein Intranet-Portal eingebunden werden.

Nachteile:

- HTML stellt nur eine sehr kleine Widget-Auswahl zur Verfügung. Diese enthält nur jene Widgets, die für Formulare notwendig sind. Komplexere Widgets, wie beispielsweise ein Baum, müssten neu implementiert werden, bzw. von Drittanbietern eingebunden werden.
- Trotz der Standardisierung von JavaScript und DOM muss durch Unterschiede in den Implementierungen der einzelnen Browser mit Inkompatibilitäten gerechnet werden.

5.2.4 Entscheidung

Gegen die ersten beiden Lösungsmöglichkeiten spricht vor allem der jeweilige Technologiebruch. Durch diesen Bruch ist mit einigen Problemen zu rechnen: Zum einen

[1] Siehe Abschnitt 2.2 auf Seite 12.

ist davon auszugehen, dass Drag-and-Drop von einer Technologie zur anderen entweder gar nicht oder nur mit erheblichem Aufwand möglich ist. Zum anderen werden sich die Unterschiede im Look-and-Feel der verschiedenen Technologien wohl kaum vollständig ausräumen lassen, so dass dem Benutzer der Bruch ständig bewusst wäre. Das würde die Akzeptanz der Software senken, weil immer der Eindruck einer „gebastelten“ Oberfläche zurückbleiben würde. Des Weiteren würde einiges an Entwicklung doppelt geleistet werden müssen.

Insgesamt kann man also davon ausgehen, dass eine solche Lösung den Entwicklungsaufwand erheblich steigern würde und dass das Ergebnis trotzdem unbefriedigend wäre.

Im Gegensatz dazu sind die Nachteile der DHTML-Lösung kompensierbar: Die benötigten komplexeren Widgets kann man mit überschaubarem Aufwand selbst erstellen, mit einem Browser als Plattform hat man die Möglichkeit, schnell und einfach komplexe Layouts zu realisieren. Wenn sich auch Inkompatibilitäten zwischen den verschiedenen Browsern in Grenzen halten, kann man die DHTML-Lösung komplett in den positiven Bereich hieven.

Um den Punkt der Inkompatibilitäten näher zu untersuchen, habe ich für die kritischen Punkte Proof-of-concept-Implementierungen durchgeführt und mit verschiedenen Browsern getestet.

Folgende Punkte wurden untersucht:

- Dynamisches HTML als solches, also das Verändern einer Webseite durch JavaScript und DOM➦.
- Die clientseitige XSLT-Transformation im Browser.
- Eine beispielhafte Implementierung der komplexen Widgets➦ SplitPane und TreeView.
- Das Erstellen und Manipulieren von XML-Dokumenten über das DOM➦.
- Das Parsen von Velocity-Scripten.
- Die XSLT-Transformation einer Velocity-Report-Vorlage, was der späteren WYSIWYG-Anzeige entspricht.
- Drag-and-Drop.

Diese Beispiel-Implementierungen haben gezeigt, dass es erstaunlich wenig Unterschiede im Verhalten der verschiedenen Implementierungen gibt - nicht zu vergleichen mit

den Querelen um den HTML-Standard in den späten 90er-Jahren zwischen dem Internet Explorer und Netscape Navigator.

5.3 Architektur

5.3.1 Verallgemeinerung des Editors

Hintergrund und Vorteile

Alle bisher beschriebenen Oberflächenkonzepte lassen sich auf jede Template-Sprache anwenden, solange dabei HTML-Dokumente erzeugt werden. Das in Abschnitt 3.2 auf Seite 17 beschriebene Prinzip der WYSIWYG-Vorschau für Vorlagen kann an Stelle des Velocity-Codes genausogut JSP-Anweisungen oder PHP-Befehlsblöcke im ansonsten statischen HTML-Dokument anzeigen. Auch die Gliederungsansicht kann verallgemeinert werden: Jedes Template-Dokument kann als ein Baum gesehen werden, der die statischen HTML-Elemente mit ihrer Schachtelung darstellt, wobei die Code-Teile zu Kindknoten derjenigen Element-Knoten werden, die den Code enthalten. D. h. Code in einer Tabellenzelle kann als Kindknoten des Tabellenzellenknotens dargestellt werden. Dieser Baum lässt sich schließlich in der Gliederungsansicht anzeigen und auch das Hinzufügen und Verschieben von Elementen per Drag-and-Drop funktioniert somit nach dem gleichen Prinzip.

Der Editor lässt sich also verallgemeinern. Was ist jedoch der Nutzen einer Verallgemeinerung? Die dabei nötige Abstraktionsschicht macht das gesamte System zwar komplexer, aber sie hilft auch dabei, das Design sauber zu halten, da zwischen allgemeinen Problemen und Problemen, die nur PTV-Reports betreffen, unterschieden wird. Der Code, der spezielle Eigenarten der PTV-Reports behandelt, wird automatisch in einem eigenen Bereich konzentriert, während sich der allgemeine Teil auf einer abstrakten Ebene bewegt, wodurch unbeabsichtigte „Hacks“ unwahrscheinlicher werden und das Gesamtkonstrukt damit solider wird.

Die Code-Basis des Editors würde damit grob aus drei Teilen bestehen:

- *Widget-Bibliothek:* Enthält alle Widgets und Hilfsklassen, die sich auch in anderen Anwendungen verwenden lassen.
- *Template-Editor:* Definiert einen allgemeinen, abstrakt gehaltenen Editor.

- *PTV-Reporter-Anpassung:* Erweitert den allgemeinen Editor um die Besonderheiten von PTV-Reports.

Ein weiterer, sehr wichtiger Vorteil ist, dass die PTV-AG durch diese Aufteilung in der Lage ist, den allgemeinen Teil unter einer Open-Source-Lizenz freizugeben, ohne die Anpassung für PTV-Reports auch freigeben zu müssen.

Um das Interesse noch zu erhöhen, könnte man eine Anpassung für JSPs oder PHP entwickeln und veröffentlichen. Damit würde man sicherlich viele interessierte Nutzer gewinnen, welche die Widget-Bibliothek oder den Editor nutzen und weiterentwickeln und damit kostenlose Bugfixes und neue Features an die PTV-AG zurückgeben.

Technische Umsetzung

Für die technische Umsetzung eignet sich am besten eine Kombination der Entwurfsmuster „Abstrakte Fabrik“ (auch „Kit“ genannt) und „Bridge“ nach [Gamma96] (siehe Abbildung 5.3 auf der nächsten Seite).

5.3.2 Adapter-Modell

Der abstrakte Template-Editor definiert die Klasse `AbstractEditorKit`. Diese erzeugt alle Objekte, die ein spezieller Editor durch eigene Implementierungen ersetzen können soll. Dies geschieht, indem er ein eigenes Kit anbietet, das von `AbstractEditorKit` erbt und die entsprechenden Fabrikmethoden überschreibt. Die abgeleitete Fabrik-Klasse muss dabei nicht alle Objekte durch eigene ersetzen: Wenn die allgemeine Version schon alles kann, was der spezielle Editor will, dann wird die entsprechende Fabrikmethode einfach nicht überschrieben. Die Spezialisierung enthält damit wirklich nur jene Teile, die auch wirklich speziell sind.

Die PTV-Reporter-Anpassung überschreibt beispielsweise den `TemplateLoader` und das `TemplateTreeModel`, während die SelectionModels übernommen werden. Durch den `ReportLoader` ist sie in der Lage, PTV-Reports zu laden und zu parsen und in einem speziellen `ReportModel` zu verwalten. Das `ReportTreeModel` enthält dabei jenen Code, der nötig ist, um die Struktur des speziellen `ReportModels` zu erfassen.

Durch diese Konstruktion können sowohl auf der allgemeinen Seite, wie auf der speziellen Seite voneinander unabhängige Vererbungshierarchien entstehen. So kann der

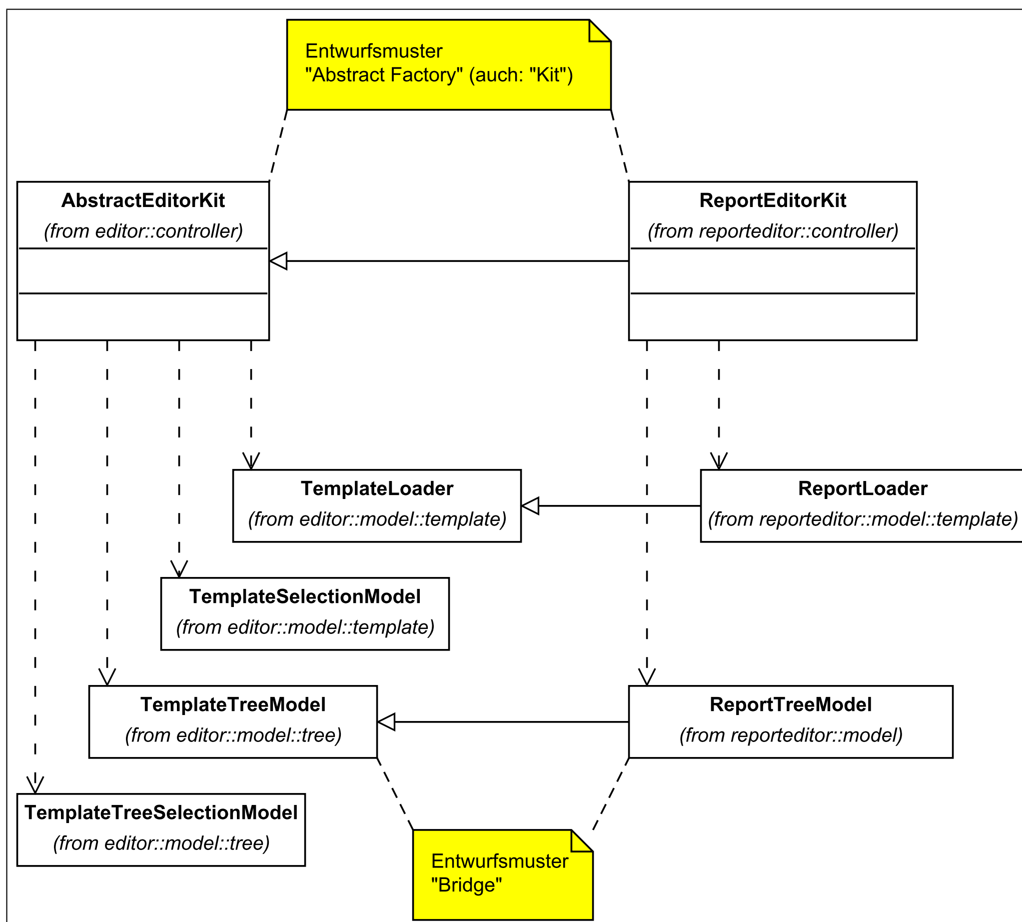

Abbildung 5.3: Klassendiagramm der Editor-Abstraktion

allgemeine Editor z.B. neben der abstrakten Template-Preview-Klasse eine abgeleitete Klasse anbieten, die die WYSIWYG-Vorschau durch eine XSLT-Transformation erzeugt. Spezielle Editoren können nun wahlweise von der abstrakten Preview oder von der XSLT-Preview ableiten.

Bei der Verwaltung der Template-Daten zeigt sich eine Reihe von Problemen: Die Modelle der Gliederungsansicht und der Vorschau sollen einerseits auf dieselben Daten zugreifen, andererseits jedoch voneinander entkoppelt sein. Außerdem soll das Template von einem Modell verwaltet werden, das es in seinem natürlichen Zustand als XML-Dokument belässt. Die TreeView soll eine Sicht auf die Daten in Form eines Baums von TreeNodes haben, die Preview soll eine Sicht in Form eines HTML-Dokuments haben. Außerdem sollen TreeView und Preview wiederverwendbar sein, d.h. ihre Modelle sollen allgemeingültige Bäume bzw. HTML-Dokumente repräsentieren.

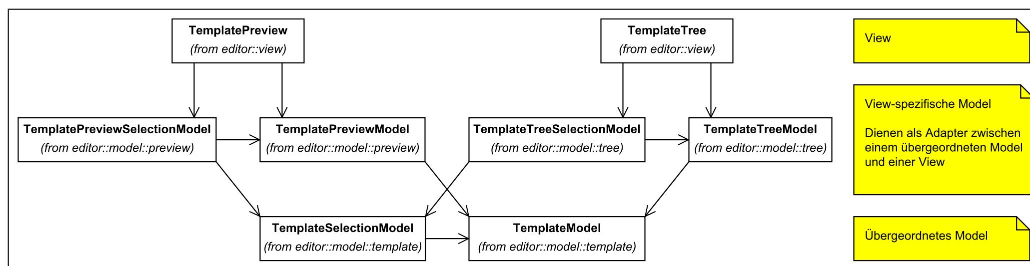

Abbildung 5.4: Klassendiagramm des Adapter-Modells

Trotzdem soll der Editor das mit dem Model-View-Controller-Muster gewonnene Verhalten zeigen: Eine Änderung des Templates in der Baum-Ansicht soll auch sofort in der Vorschau-Ansicht sichtbar sein und umgekehrt.

Dies alles kann man erreichen, indem man vor das eigentliche Modell ein Adapter-Modell vorlagert (siehe Abbildung 5.4). D.h. die Daten werden von einem Modell verwaltet, auf das mehrere Adapter-Modelle zugreifen. Die Adapter-Modelle arbeiten nach dem Entwurfsmuster „Adapter“ nach [Gamma96]: Sie passen das eigentliche Modell an die Schnittstelle eines anderen Modell-Typs an. Sie verändern quasi die Sicht auf die Daten. Die Modell-Typen sind somit entkoppelt und die zugehörigen Views wiederverwendbar.

Im Falle des Editors heißt das, dass man die eigentlichen Daten in einem übergeordneten `TemplateModel` verwaltet. Das Template wird in diesem Modell in Form eines XML-Dokuments verwaltet, was nach außen hin auch so dargestellt wird. Auf dieses übergeordnete Modell greifen nun sowohl das TreeModel als auch das PreviewModel zu. Sie greifen damit auf dieselbe Datenbasis zu und sie propagieren Änderungen an den Daten an ihre jeweiligen Views. Da sie selbst keine oder nur zwischengespeicherte Daten enthalten, dienen sie nur als Adapter. Sie bieten der jeweiligen View eine bestimmte Sicht auf die Daten. Für die TreeView werden die Daten in Form eines Baums aus TreeNodes dargestellt, für die Preview in Form eines HTML-Dokuments.

Wird nun in der Gliederungsansicht eine Änderung vorgenommen, so meldet die TreeView dies an ihr Modell. Das Modell reicht das Ereignis nun an sein übergeordnetes Modell weiter. Dieses ändert die Daten und erzeugt ein Ereignis. Dieses Ereignis wird nun – und das ist der Trick – sowohl vom TreeModel, als auch vom PreviewModel an ihre jeweiligen Views weitergegeben. Das Resultat erscheint also in der Gliederungsansicht und in der Vorschau gleichzeitig, ohne dass diese voneinander wissen.

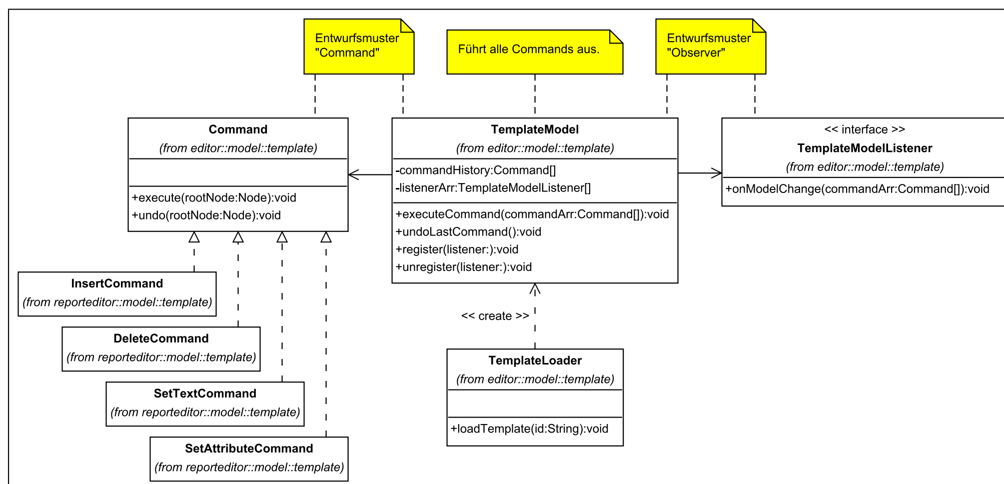

Abbildung 5.5: Klassendiagramm des Multilevel-Undo

5.3.3 Multilevel-Undo

Eine sehr hilfreiche und mächtige Funktion in Editoren ist das Multilevel-Undo. Dabei merkt sich der Editor jede Änderung, die der Benutzer im Dokument vornimmt, so dass er sie ggf. schrittweise wieder rückgängig machen kann. Bei einer versehentlichen großen Änderung kann das dem Benutzer sehr viel Zeit und Aufregung ersparen.

Die technische Umsetzung dieser Funktion erfolgt üblicherweise über das Entwurfsmuster „Command" nach [Gamma96]. Dabei wird jeder Befehl in einem Objekt gekapselt. Diese Befehls-Objekt können nun mit Parametern versorgt, ausgeführt und in einer Liste verwaltet werden. Will der Benutzer eine Aktion rückgängig machen, kann das Befehls-Objekt seine Änderung wieder zurücknehmen. Das Wichtige dabei ist, dass sowohl die Ausführung wie auch das Zurücknehmen einer Änderung in einem Objekt gekapselt stattfinden (siehe Abbildung 5.5).

In der Praxis kann eine Undo-Funktion eine große Fehlerquelle darstellen, und zwar immer dann, wenn ein Befehls-Objekt doch nicht den gleichen Ursprungszustand wiederherstellt und damit Folgefehler auslöst. Trotz dieser Probleme sollte bei der Entwicklung eines Editors von Anfang an mit dem Command-Muster gearbeitet werden, so dass ein Multilevel-Undo zumindest leicht nachgerüstet werden kann. So wird auch der Template-Editor von Anfang an Commands nutzen, um eine Änderung am Template auszuführen. Wenn gegen Ende der Entwicklungsphase noch genügend Zeit übrig ist, kann dann die ebenso sensible wie nützliche Funktion des Multilevel-Undo noch implementiert werden, ohne dass große strukturelle Änderungen nötig wären.

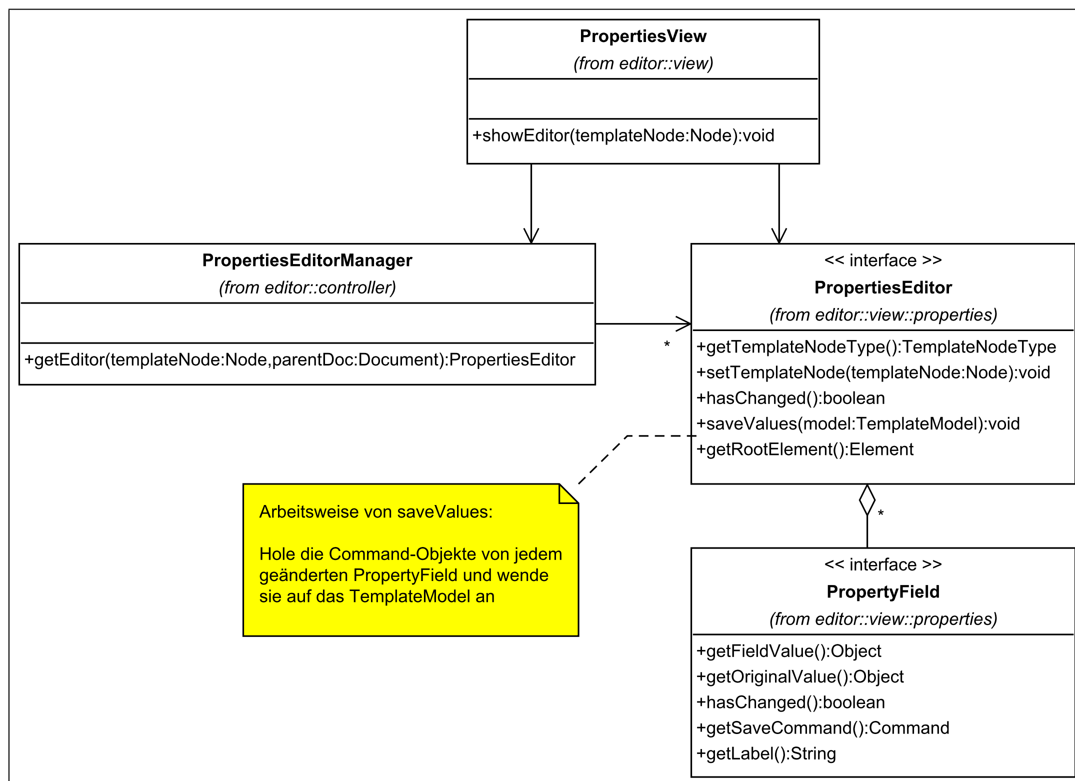

Abbildung 5.6: Klassendiagramm der Eigenschaftenansicht

5.3.4 Die Eigenschaftenansicht

Wie in Abschnitt 5.1.3 auf Seite 30 erläutert, dient die Eigenschaftenansicht dazu, die Eigenschaften des gerade selektierten Knotens anzuzeigen und zu verändern.

Sie muss dabei einer ganzen Reihe von Anforderungen genügen. Für die meisten Knoten wird sie aus einer Reihe von Textfeldern oder Comboboxen bestehen, mit deren Hilfe der Benutzer die einzelnen Eigenschaften ansehen und verändern kann. Damit dies nicht jeder spezielle Editor neu implementieren muss, sollte der abstrakte Editor ein Grundgerüst bieten, mit dessen Hilfe schnell eine eigene Eigenschaftenansicht zusammengesetzt werden kann. Es sollte damit möglich sein, sehr einfach weitere Felder hinzuzufügen, mit deren Hilfe z. B. ein Attribut oder der Text des Knotens verändert werden kann. Dieses Grundgerüst muss zudem noch erweiterbar sein, so dass ein spezieller Editor für manche Knoten auch eine komplett eigene Eigenschaftenansicht zeigen kann.

Es muss also eine klare Aufgabenteilung herrschen: Der abstrakte Editor sollte so viel Hilfe bieten wie möglich. Er sollte also besagtes Grundgerüst stellen und möglichst

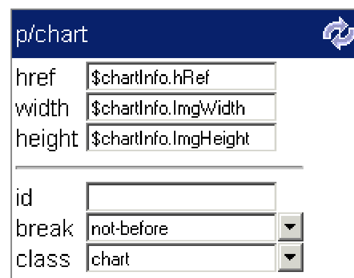

Abbildung 5.7: Die Eigenschaftenansicht

viel von der Darstellung übernehmen. Der spezielle Editor muss das Wissen liefern, für welchen Knoten welche Eigenschaftenansicht gezeigt werden soll und er muss Widgets liefern, falls er spezielle Eigenschaftenansichten anbieten will.

Dies wurde folgendermaßen umgesetzt (siehe Abbildung 5.6 auf der vorherigen Seite): Jede Eigenschaft wird von einem `PropertyField` verwaltet. Ein `PropertyField` bietet ein Eingabefeld, über das der Wert der Eigenschaft angesehen und geändert werden kann, es übernimmt die Speicherung des Wertes und es kennt den Namen der Eigenschaft, der als Beschriftung vor dem Eingabefeld angezeigt werden soll. Die Klasse `PropertyField` ist abstrakt. Der abstrakte Editor bietet jedoch einige Implementierungen für die gängigsten Fälle, also für die Änderung von Attributen oder den Text eines Knotens über ein Textfeld oder eine ComboBox.

Alle Eigenschaften eines Knotens werden in einem `PropertiesEditor` zusammengefasst. Dieser übernimmt für einen Knoten die Darstellung der gesamten Eigenschaftenansicht und das Speichern der Änderungen. Auch diese Klasse ist abstrakt, so dass ein spezieller Editor auch eine völlig eigene Ansicht anbieten kann. Der abstrakte Editor liefert jedoch auch hier eine Implementierung, welche die eben vorgestellten `PropertyFields` in einer typischen Beschriftung/Eingabefeld-Anordnung anzeigt (siehe Abbildung 5.7). Ein `PropertiesEditor` muss nicht mit `PropertyFields` arbeiten, er kann die Verwaltung der einzelnen Eigenschaften auch komplett selbst übernehmen.

Das Wissen, welcher `PropertiesEditor` für welchen Knoten gezeigt werden soll, wird vom `PropertiesEditorManager` verwaltet. Auch diese Klasse ist abstrakt, sie muss von einem speziellen Editor implementiert werden.

Die `PropertiesView` kümmert sich darum, dass immer der Editor des gerade selektierten Elements zu sehen ist. Ihr Modell ist das `TemplateSelectionModel`, welches auch von der Gliederungsansicht und der Vorschau verwendet wird (siehe Abschnitt 5.3.2 auf Seite 38).

©2005 Tilman Schneider

Wie spielen diese Komponenten nun zusammen? Sobald der Benutzer einen anderen Knoten selektiert, feuert das `TemplateSelectionModel` ein Event an seine Listener, zu denen auch die `PropertiesView` gehört. Diese veranlasst nun die Speicherung aller geänderten Eigenschaften beim gerade angezeigten `PropertiesEditor`. Danach holt sie sich vom `PropertiesEditorManager` den passenden `PropertiesEditor` für den neu selektierten Knoten und zeigt diesen an.

Der Benutzer kann nun nach Belieben Eigenschaften des selektierten Knotens ändern. Selektiert er einen anderen Knoten oder drückt er auf den Aktualisieren-Knopf (siehe Abbildung 5.7 auf der vorherigen Seite rechts oben), dann veranlasst die `PropertiesView`, wie eingangs erwähnt, die Speicherung der Änderungen. Dazu ruft sie beim `PropertiesEditor` die Methode `saveValues` auf. Diese geht durch die Liste der `PropertyFields` und holt sich bei jedem geänderten Feld ein Command-Objekt, das die Änderung durchführen kann. Die gesammelten Command-Objekte werden dann auf das `TemplateModel` angewendet (mehr Informationen zu Command-Objekten siehe Abschnitt 5.3.3 auf Seite 41).

Im einfachsten Fall muss ein spezieller Editor nur einen `PropertiesEditorManager` implementieren, der für jeden Knoten-Typ mit Hilfe der `PropertyFields` einen Standard-`PropertiesEditor` zusammensteckt. Falls er eine spezielle Anzeige wünscht, kann er für einen Knoten entweder einen komplett eigenen `PropertiesEditor` anbieten oder er kann eigene `PropertyFields` mitbringen, falls er nur bestimmte Eigenschaften auf eine andere Art darstellen möchte.

Mit diesem System kommt der Entwickler eines speziellen Editors im Standardfall also sehr schnell und einfach zu einer Lösung, gleichzeitig hat er die Möglichkeit, auf allen Ebenen auch spezielle Anpassungen einzubringen.

5.3.5 Drag-and-Drop

Kaum eine Oberflächentechnik ist so interaktiv wie das Drag-and-Drop[2]. Die Bewegung des Mauszeigers muss überwacht werden und es muss dem Benutzer fortlaufend gezeigt werden, was für ein Objekt er gerade bewegt und welche Auswirkungen ein Fallenlassen an der aktuellen Position hat. Da zu erwarten ist, dass diese hohe Interaktivität nur erreicht werden kann, indem genau auf die Eigenheiten der Plattform eingegangen wird, sollte die systemnahe Ereignisbehandlung möglichst gut gekapselt sein. So kann

[2] Engl. „drag and drop“: Ziehen und fallen lassen

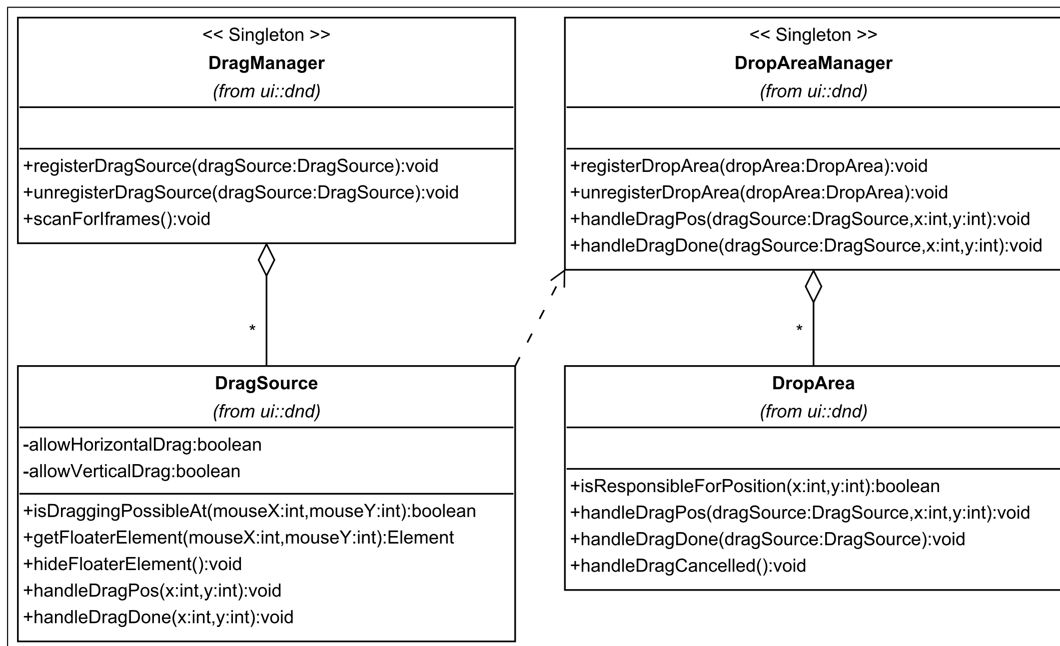

Abbildung 5.8: Klassendiagramm des Drag-And-Drop

sie leicht ausgetauscht werden und alle Anpassungen an spezielle Plattformen sind an einer Stelle konzentriert.

Diese Kapselung wird von einem zentralen Singleton, dem `DragManager` erreicht (Siehe Abbildung 5.8). Dieser hängt sich möglichst weit unten in die Ereignisverarbeitung ein, so dass er alle Mausbewegungen mitbekommt. Er ist damit in der Lage, Drags quer durch das gesamte HTML-Dokument zu überwachen. Der `DragManager` bildet die Schnittstelle zum Browser und kapselt alle Browser-spezifischen Details. Er bildet damit eine stabile Basis, auf der die eigentliche Behandlung der Drags aufgesetzt werden.

Die Definition eines dragbaren Elements erfolgt über eine `DragSource`. Diese meldet sich beim `DragManager` an. Sobald der `DragManager` feststellt, dass der Benutzer über einer `DragSource` mit gedrückter Maustaste eine Dragbewegung beginnt, fordert er von der `DragSource` einen sogenannten Floater an. Ein Floater ist ein HTML-Element, welches das Objekt repräsentiert, das der Benutzer bewegen will. Der Floater wird nun mit der Maus mitbewegt, also gedragt. Jede Neupositionierung wird der `DragSource` migeteilt. Da sich der `DragManager` nur um die systemnahe Ereignisbehandlung kümmern soll, ist es Aufgabe der `DragSource`, sich um die weiteren Maßnahmen zu kümmern, etwa das Verschieben des Floaters oder die Reaktion auf die aktuelle Drag-Position. Ei-

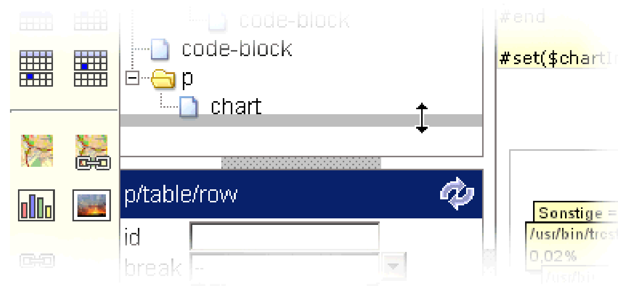

Abbildung 5.9: Ein Schiebebalken wird per Drag-and-Drop bewegt

ne `DragSource` ist also eine Quelle von Drags. Sie löst Drags aus, zeigt mit Hilfe des Floaters eine Repräsentation des Objektes und sie verwaltet den Drag an sich.

Manche Objekte werden einfach nur verschoben, ohne ein bestimmtes Ziel. Zu dieser Kategorie gehören beispielsweise die Schiebebalken, welche die Gliederungsansicht, Eigenschaftenansicht und Vorschau voneinander trennen. Diese Balken können vom Benutzer verschoben werden, um den verfügbaren Platz aufzuteilen (siehe Abbildung 5.9). Die zuständige `DragSource` zeigt als Floater einen grauen, halbtransparenten Balken, der den Schiebebalken repräsentiert. Lässt der Benutzer den Floater fallen, dann informiert der `DragManager` die `DragSource`, so dass diese den eigentlichen Schiebebalken auf die neue Position setzen kann.

Eine weitere Kategorie sind Objekte, die von einer Position zu einem bestimmten Ziel getragen werden. Hierzu zählen beispielsweise Elemente, die zur Vorlage hinzugefügt werden sollen. Der Benutzer zieht dazu ein Element aus der Drag-Bar an die gewünschte Stelle in der Vorlage. Zu dem Element, das Drags auslöst, kommt also noch ein Element, das Drags empfangen kann. Solche Elemente werden über `DropAreas` definiert und von einem zentralen Singleton, dem `DropAreaManager`, verwaltet. Eine `DropArea` ist also ein Ziel für Drags. Sie weiß, welche Objekte bei diesem Ziel an welcher Position welche Aktion auslösen können. Sie visualisiert diese Aktion während des Drags und führt sie im Falle eines Drops auch aus.

Die Standard-Implementierung der `DragSource` behandelt alle Ereignisse, die sie vom `DragManager` übergeben bekommt, nicht selbst, sondern sie delegiert sie an den `DropAreaManager`. Dieser kennt alle `DropAreas` und ermittelt unter ihnen diejenige, die für die aktuelle Position zuständig ist. Die zuständige `DropArea` verarbeitet daraufhin das Ereignis und zeigt beispielsweise einen Cursor.

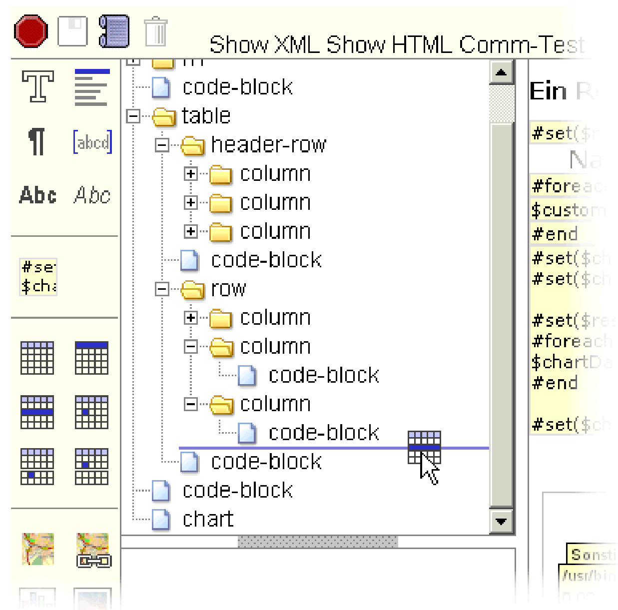

Abbildung 5.10: Ein row-Element wird in eine Tabelle gezogen

In Abbildung 5.10 wird dieses Zusammenspiel deutlich: Am Mauscursor ist das Symbol für eine Tabellenzeile zu sehen. Dieses Symbol ist der von der `DragSource` gezeigte Floater, der dem Benutzer zeigt, was für ein Objekt er gerade bewegt. Zusätzlich ist ein Cursor in Form eines blauen, halbtransparenten Striches zu sehen. Dieser wird von der `DropArea` – in diesem Fall die Gliederungsansicht – dargestellt und zeigt dem Benutzer, dass an dieser Stelle ein Drop möglich ist und wo genau er sich auswirkt. Der Benutzer sieht also auf einen Blick, dass er gerade eine Tabellenzeile in eine Tabelle zieht.

Durch diese Konstruktion entsteht eine Entkopplung von Quelle und Ziel. Die Quelle kümmert sich nur darum, anzuzeigen, welche Art von Objekt gerade bewegt wird. Was mit diesem Objekt passiert, liegt nicht in ihrem Aufgabenbereich und wird daher weiterdelegiert. Das Ziel wiederum kümmert sich nicht darum, wo ein Objekt herkommt. Es muss nur wissen, mit welcher Art von Objekt wo was gemacht werden kann.

Dadurch, dass die Events immer über die Quelle laufen, sind zum einen auch `DragManager` und `DropAreaManager` voneinander entkoppelt und zum andern sind auch Drags ohne bestimmtes Ziel möglich, wie am Beispiel des Schiebebalkens gezeigt. Durch die jeweils der Quelle und dem Ziel vorgelagerten Manager müssen jede Quelle und jedes Ziel nur jeweils ihre eigenen Ereignisse verarbeiten, was deren Entwicklung sehr einfach macht.

5.3.6 Entwurf eines komplexen Widgets

Der Entwurf von komplexen Widgets in einem JavaScript-Client unterscheidet sich nicht von dem bei anderen Architekturen. Die bei graphischen Oberflächen üblichen Architektur- bzw. Entwurfsmuster, wie Model-View-Controller oder Beobachter, lassen sich auch hier ohne weiteres einsetzen.

Ein sinnvoller Entwurf einer TreeView wäre beispielsweise folgender (siehe Abbildung 5.11 auf der nächsten Seite): Die Trennung zwischen Daten, Darstellung und Verhalten wird über das Architekturmuster Model-View-Controller hergestellt. Der eigentliche Baum wird von einem TreeModel verwaltet. Eine TreeView fragt dort alle Daten ab, die sie zur Darstellung braucht. Sobald sich die Daten ändern, informiert das Modell seine Views. Dieser Teil ist mit Hilfe des Entwurfsmusters „Beobachter“ entkoppelt, also über eine Listener-Schnittstelle. So besteht keine zirkuläre Abhängigkeit und es können auch andere Beobachter über Änderungen am Modell informiert werden. Der Controller, der das Verhalten des Dialogs implementiert, muss vom Klienten des Trees gestellt werden und ist daher nicht Teil dieses Entwurfs.

Der gerade selektierte Knoten wird auf analogem Weg von einem anderen Modell verwaltet. So kann dasselbe TreeModel von mehreren TreeViews genutzt werden, ohne dass auch der jeweils selektierte Knoten derselbe sein muss. Außerdem kann die Verwaltung der Selektion weiterdelegiert werden, siehe Abschnitt 5.3.2 auf Seite 38.

Schließlich wird noch die Darstellung der einzelnen Knoten in eine eigene Klasse ausgelagert, so dass die TreeView-Klasse wiederverwendet werden kann, auch wenn eine andere Darstellung gewünscht wird.

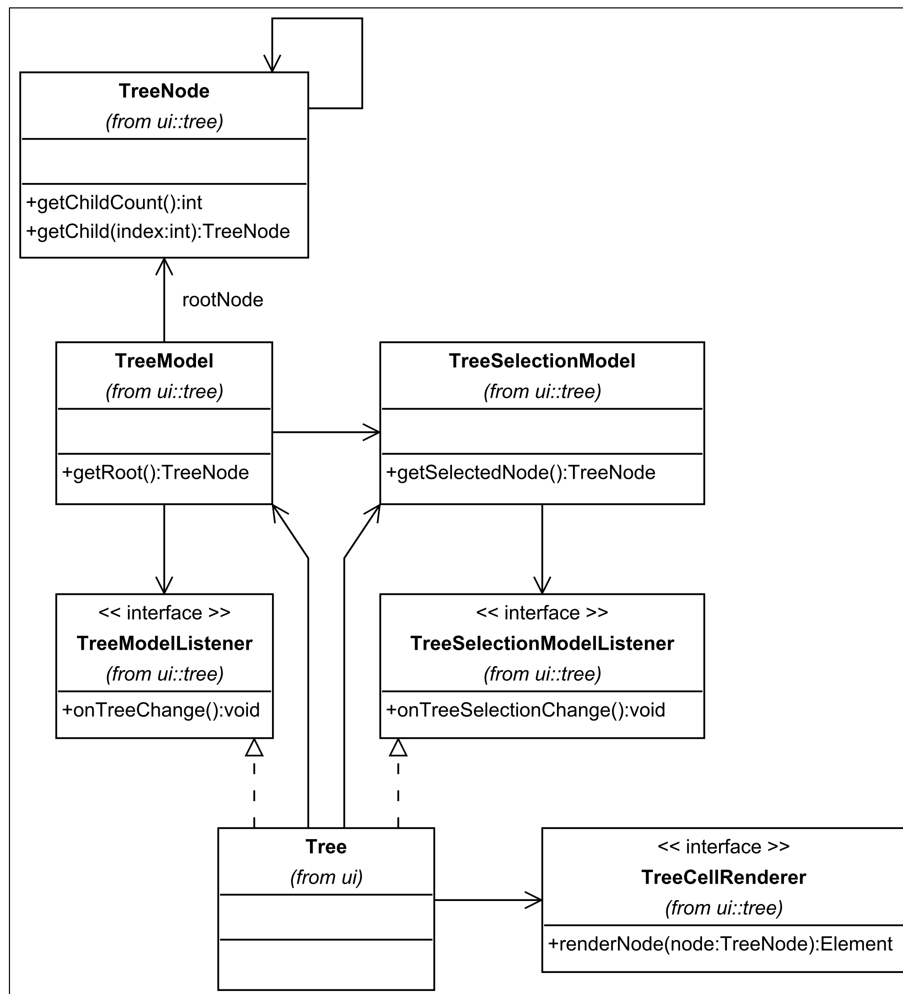

Abbildung 5.11: Klassendiagramm der TreeView

Kapitel 6

Implementierung

6.1 Implementierung eines komplexen Widgets

In Abschnitt 5.3.6 auf Seite 48 wurde gezeigt, dass der Entwurf eines komplexen Widgets genau so erstellt wird wie bei anderen Technologien auch. Bei der Implementierung ist das jedoch nicht der Fall. Hier kommt es naturgemäß sehr wohl auf die jeweilige Technologie an. Durch sie wird schließlich bestimmt, wie gezeichnet und wie auf Ereignisse reagiert wird.

Bei einem JavaScript-Client hat man das Glück, dass die Darstellung durch den Browser vorgenommen wird. Man arbeitet also auf einer Plattform, die für Layout und Darstellung sehr viel Unterstützung liefert. Im Gegensatz zu den meisten anderen graphischen Oberflächen zeichnet sich eine Komponente nicht selbst, sondern sie nutzt das DOM➦, um HTML-Elemente zu erzeugen, die an entsprechender Stelle in das HTML-Dokument eingefügt werden. Für die Ereignisverarbeitung werden an passender Stelle Event-Handler-Methoden[1] an den HTML-Elementen platziert.

Durch eine geschickte Anordnung von HTML-Elementen lassen sich manche Problemstellungen wesentlich eleganter lösen, als das bei herkömmlichen GUI-Frameworks der Fall ist. Auf der anderen Seite hat man jedoch in manchen Fällen auch mit den Browser-Eigenarten zu kämpfen.

Im Falle der TreeView wurde die Darstellung folgendermaßen umgesetzt (siehe Abbildung 6.1 auf der nächsten Seite): Jeder Knoten im Baum wird durch ein `div`-Element

[1] Details siehe Abschnitt 6.3 auf Seite 56.

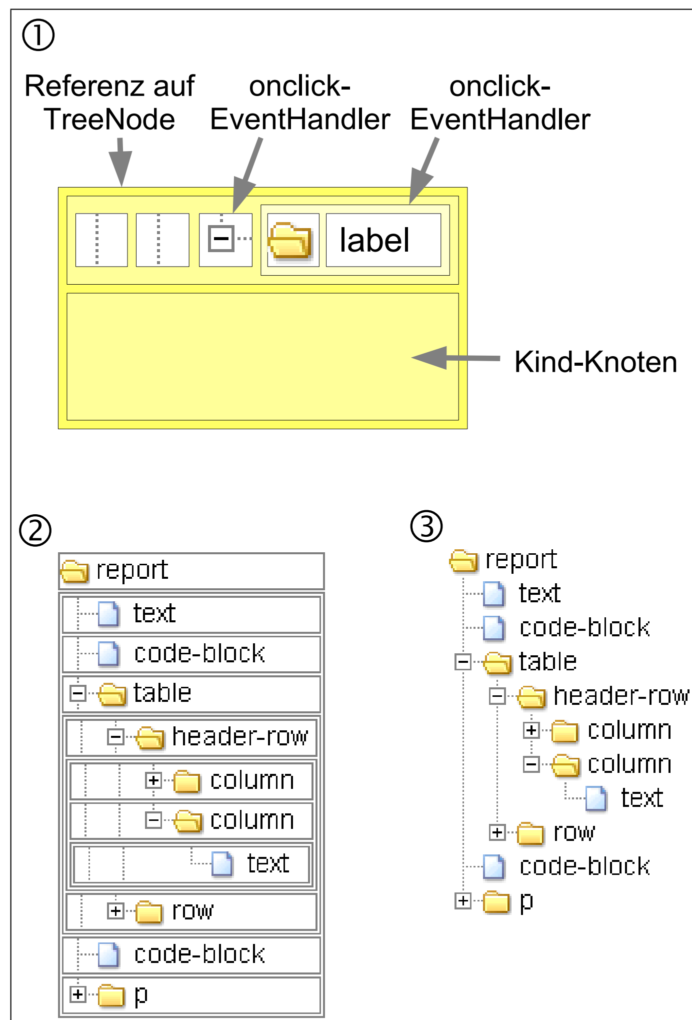

Abbildung 6.1: Aufbau der TreeView aus HTML-Elementen

repräsentiert (siehe ①). Dieses Haupt-Element enthält ein `div`-Element für die Darstellung des Knotens selbst und eines für dessen Kinder. Im `div`-Element, das den Knoten darstellt, sind zunächst einige `img`-Elemente, die die Linien für die höheren Ebenen darstellen. Danach folgt ein `img`-Element, welches das Plus/Minus-Zeichen zeigt, über das der Knoten auf- bzw. zugeklappt wird. Schließlich folgt ein `div`-Element mit dem Icon und der Beschriftung des Knotens.

Das äußerste `div`-Element erhält eine Referenz auf das TreeNode-Objekt aus dem Datenmodell, das damit repräsentiert wird. So kann man sehr einfach die Rückverbindung herstellen, ohne dass der ganze Baum traversiert werden muss. Dies wird gebraucht, wenn man beispielsweise beim Drag-and-Drop feststellen will, welcher Knoten an einer bestimmten Mausposition liegt.

Beim `img`-Element des Plus/Minus-Zeichens und beim `div`-Element der Beschriftung werden onclick-Handler registriert, die das Auf- und Zuklappen des Knotens, bzw. dessen Selektion auslösen. Bei herkömmlichen GUI-Frameworks hat man es üblicherweise mit größeren Komponenten zu tun, d. h. man müsste in diesem Fall bei der Ereignisbehandlung berechnen, ob der Benutzer nun auf das Plus/Minus-Zeichen oder auf die Beschriftung oder auf keines von beiden geklickt hat. Dadurch, dass man im Browser an jedes noch so kleine Element einen Handler hinzufügen kann, kann man sich diesen Overhead sparen.

Das `div`-Element mit den Kind-Knoten wird erst bei Bedarf erzeugt, um Ressourcen zu sparen. Dadurch, dass alle Kind-Knoten in einem `div`-Element vereint sind, kann dieses beim Zuklappen einfach versteckt werden, um beim erneuten Aufklappen wieder gezeigt zu werden. Dadurch müssen nicht ständig Kind-Knoten-Repräsentationen aufgeräumt und wieder erstellt werden, das Zu- und wieder Aufklappen ist daher sehr effizient. Außerdem wird automatisch der korrekte Zustand von aufgeklappten Unterknoten wieder hergestellt, d. h. Unterknoten, die zuvor aufgeklappt waren, sind dann wieder aufgeklappt. So muss sich der Benutzer nicht den ganzen Weg zu einem Unterknoten wieder aufklappen, wenn er versehentlich einen Knoten nahe der Wurzel zugeklappt hat. Oder er kann kurzzeitig Teile des Baumes ausblenden, ohne beim erneuten Einblenden erst mühsam den alten Zustand wiederherstellen zu müssen.

Die Kindknoten sind wieder nach demselben Muster aufgebaut, so dass sich die Repräsentation beliebig schachteln lässt (siehe ②). Diese Schachtelung ist für den Benutzer nicht sichtbar, sie dient lediglich der effizienten Verwaltung der Darstellung (siehe ③).

Das wirklich Schöne an der Entwicklung von Widgets im Browser liegt darin, dass man nur die nötigen HTML-Elemente in das Dokument einzufügen braucht. Alles andere, das Berechnen der gewünschten Größe, das Neuzeichnen, die Reaktion auf Änderungen im Dokument, usw. wird vom Browser übernommen.

6.2 Drag-and-Drop

Drag-and-Drop wird im Report-Editor an verschiedenen Stellen verwendet: Zum einen werden per Drag-and-Drop die Schiebebalken bewegt, welche die Gliederungsansicht, Eigenschaftenansicht und Vorschau voneinander abtrennen. Zum anderen ist Drag-and-Drop das zentrale Bedienkonzept des Report-Editors. Neue Elemente werden in die Vorlage eingefügt, indem sie von der Drag-Bar an die gewünschte Stelle gezogen werden. Bereits vorhandene Elemente werden durch einfaches Ziehen an den gewünschten

Zielort an eine andere Stelle verschoben. Auch das Löschen von Elementen kann mit Drag-and-Drop erfolgen, indem ein nicht mehr benötigtes Element auf den Mülleimer gezogen wird.

All diese Aktionen können wahlweise in der Gliederungsansicht oder in der Vorschau erfolgen. So kann man ein Element, das man in der Gliederungsansicht ausgewählt hat, entweder in der Vorschau oder wieder in der Gliederungsansicht fallen lassen und umgekehrt. Die gesamte Bedienung wird damit konsistent und sehr intuitiv. Durch die Tatsache, dass das gerade selektierte Element sowohl in der Vorschau als auch in der Gliederungsansicht hervorgehoben wird, wird dem Benutzer verdeutlicht, welches Element wie aussieht und welcher Teil der Vorschau zu welchem Element gehört. Durch das beliebige Bewegen der Elemente zwischen Gliederungsansicht und Vorschau verhält sich ein Element auch entsprechend.

Die technische Umsetzung von Drag-and-Drop im Browser ist recht kompliziert. Man muss dafür sorgen, dass man alle Ereignisse mitbekommt, damit man den Drag verfolgen kann. Dazu kommt, dass ein Browser bereits selbst auf Drag-Bewegungen reagiert. Der Benutzer kann so Text markieren oder Bilder in andere Fenster ziehen. Dieses normale Drag-and-Drop-Verhalten des Browsers muss unterdrückt werden. Schließlich muss dem Benutzer noch eine Rückmeldung gezeigt werden, also beispielsweise ein kleines Symbol, das sich mit dem Mauscursor mitbewegt oder ein Cursor, der anzeigt, wo genau eine Aktion stattfinden würde, wenn der Benutzer das gerade bewegte Objekt fallen läßt. Diese Rückmeldung soll sich flüssig mit der Maus mitbewegen, ohne allzu sehr zu ruckeln. Dem Benutzer soll damit intuitiv klar werden, was für ein Objekt er gerade bewegt und welche Auswirkungen ein Drop an einer bestimmten Position hat.

Da Drag-and-Drop hochinteraktiv ist und da seine Umsetzung eng mit der Ereignisverarbeitung des Browsers zusammenhängt, sind manche Teile der Umsetzung Browserspezifisch. Zu den gerade beschriebenen Problemen kommt also hinzu, dass sie teilweise für mehrere Browser auf verschiedene Weise gelöst werden müssen.

6.2.1 Die Rolle von Iframes im Report-Editor

Im Report-Editor werden für die Anzeige der Gliederungsansicht, der Eigenschaftenansicht und der Vorschau Inlineframes genutzt. Durch einen Inlineframe (kurz: Iframe) kann in ein HTML-Dokument ein weiteres HTML-Dokument eingebunden werden. Ist das Iframe-Dokument größer als der Iframe, so zeigt der Browser Scrollbalken.

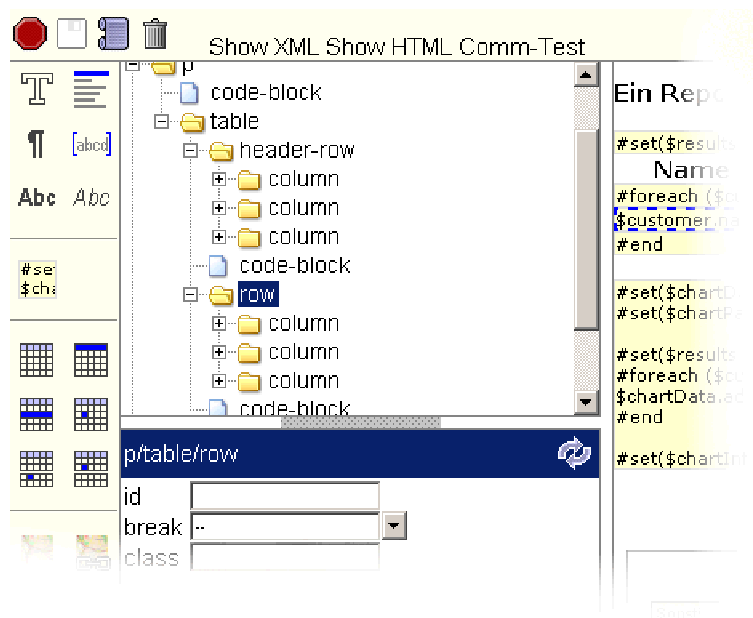

Abbildung 6.2: Durch den Iframe bekommt die Gliederungsansicht Scrollbalken

Diese Eigenschaft macht sich der Report-Editor zunutze, um Gliederungsansicht, Eigenschaftenansicht und Vorschau so anzuzeigen, dass sie bei Bedarf scrollbar sind (siehe Abbildung 6.2). Dieses Verhalten kann zwar auch erreicht werden, indem man bei einem `div`-Element die CSS-Eigenschaft `overflow` auf den Wert `scroll` setzt, dabei ergeben sich jedoch Probleme mit der dynamischen Größenzuweisung in Gecko-basierten Browsern, da solche `div`-Elemente sich weigern, einmal eingenommenen Platz wieder herzugeben.

Außerdem hat ein Iframe den Vorteil, dass sein Inhalt ein eigenständiges HTML-Dokument darstellt. Das Iframe-Dokument und das umgebende Dokument sind durch den Iframe sauber voneinander getrennt. Dadurch wird verhindert, dass beispielsweise CSS-Stylesheets, die im Iframe-Dokument geladen werden, das Aussehen des umgebenden Dokuments beeinflussen und umgekehrt. Da es bei der Vorschau möglich sein soll, beliebige CSS-Stylesheets einzubinden, kommt die Nutzung eines scrollbaren `div`-Elements dazu nicht in Frage.

Allerdings führen Iframes zu Problemen beim Drag-and-Drop, wie in den folgenden Abschnitten noch gezeigt wird. Aus den gerade genannten Gründen ist ihre Nutzung jedoch unumgänglich.

6.2.2 Die Drag-and-Drop-Bibliothek von Walter Zorn

Walter Zorn[2] bietet eine JavaScript-Bibliothek, die Drag-and-Drop ermöglicht, was auf seiner Webseite auch recht eindrucksvoll demonstriert wird. Diese Bibliothek unterstützt viele Browser und kann frei genutzt und weiterentwickelt werden.

Zunächst wurde mit Hilfe dieser Bibliothek Drag-and-Drop im Report-Editor realisiert. Allerdings zeigte sich, dass kein Draggen über Iframes möglich war. Da der Inhalt eines Iframes ein eigenständiges HTML-Dokument ist, wurden die Ereignisse, die die Mausbewegung propagieren, dem Iframe-Dokument und nicht dem umgebenden Dokument mitgeteilt. Da die Bibliothek von Walter Zorn nur auf Ereignisse des umgebenden Dokuments hört, blieb der Drag damit am Rand eines Iframes stehen.

Dieses Problem konnte recht einfach durch einen Patch gelöst werden, der das umgebende HTML-Dokument nach Iframes durchsucht und bei jedem Iframe Event-Handler registriert, welche eingehende Ereignisse in die Koordinaten des umgebenden Dokuments umrechnen und dann an die Bibliothek von Walter Zorn weitergeben.

Bei der Implementierung dieses Patches zeigte sich jedoch, dass der Code der Walter-Zorn-Bibliothek fast nicht wartbar ist. Es fehlen jegliche Art von Kommentaren. Die Bezeichner haben Namen wie „`d_i`“ oder „`d_o.defw`“ und der Code ist voll von Ausdrücken wie dem folgenden:
„`dd.n4RectPos(dd.rectI[d_i], dd.obj.x + (!(d_i-1)? (dd.obj.w-1) : 0), dd.obj.y + (!(d_i-2)? (dd.obj.h-1) : 0), d_i&1 || dd.obj.w, !(d_i&1) || dd.obj.h);`“.
Diese Zeile ist keine Ausnahme, wie man vielleicht meinen könnte, und man muss kein Experte sein, um zu verstehen, dass dieser Code praktisch nicht wartbar ist.

6.2.3 Die selbst entwickelte Drag-and-Drop-Implementierung

Dank des Patches war es nun möglich, Elemente vom umgebenden Dokument über oder in einen Iframe zu ziehen. Bei der Implementierung des Verschiebens von Code-Elementen stellte es sich als erforderlich heraus, dass auch Elemente aus einem Iframe heraus gezogen werden können. Dieses Problem ließ sich leider nicht so einfach lösen, wie es beim ersten Patch der Fall war, der quasi von außen der Walter-Zorn-Bibliothek Ereignisse unterjubelte. Um dieses Problem zu lösen, hätte tief in die Walter-Zorn-Bibliothek eingegriffen werden müssen.

[2] Siehe http://walterzorn.com/dragdrop/dragdrop_e.htm

```
// HTML-Element erzeugen
var htmlElem = document.createElement("img");

// Event-Handler-Funktion hinzufügen
htmlElem.onclick = function() {
  alert("Sie haben auf das Bild geklickt");
}
```

Quelltext 6.1: Ereignisverarbeitung in JavaScript

Aufgrund ähnlich schlechter Erfahrungen und mangels Ersatz durch eine andere Bibliothek hatte mein Betreuer Andreas Junghans in der Zwischenzeit eine eigene Drag-and-Drop-Implementierung erstellt. Diese unterstützt Gecko-basierte Browser, Internet Explorer und Apples Safari. Allerdings fehlt auch hier eine Unterstützung von Iframes.

Angesichts der miserablen Codequalität der Walter-Zorn-Bibliothek schätzte ich den Aufwand sehr hoch, sie durch einen weiteren Patch wieder nutzbar zu machen.

Die Alternative war, die Drag-and-Drop-Implementierung meines Betreuers um die Unterstützung von Iframes zu erweitern und in das in Abschnitt 5.3.5 auf Seite 44 beschriebene System zu integrieren, das ich bereits auf die Walter-Zorn-Bibliothek aufgesetzt hatte. Die Implementierung war zwar auch noch nicht dokumentiert, aber der Code war gut lesbar und klar strukturiert. Kein Vergleich zum Code von Walter Zorn, bei dem man nicht abschätzen konnte, welche Seiteneffekte eine Änderung haben würde.

Dank der guten Kapselung durch den `DragManager` (siehe Abschnitt 5.3.5 auf Seite 44) war der Austausch der systemnahen Bestandteile der Drag-and-Drop-Behandlung ohne allzu großen Aufwand möglich.

6.3 EventDispatcherFactory

Die Ereignisverarbeitung wird in JavaScript über so genannte „Event-Handler-Funktionen“ abgewickelt. D. h. man kann bei Objekten, die Ereignisse auslösen, Funktionen hinzufügen, die aufgerufen werden, sobald das Ereignis eintritt. Bei einem HTML-Element kann man beispielsweise eine `onclick`-Funktion hinzufügen. Sobald der Benutzer auf dieses Element klickt, wird geprüft, ob es eine `onclick`-Funktion besitzt, welche dann aufgerufen wird (siehe Quelltext 6.1).

Im Entwurfsmuster „Beobachter[3]“ nach [Gamma96] sind zwei Objekte beteiligt: Das Subjekt, also das Objekt, welches das Ereignis auslöst, und der Beobachter, also das Objekt, welches das Ereignis konsumiert.

Der Event-Handler bei JavaScript ist Teil des Subjekts und nicht Teils des Beobachters, was im Beobachter-Muster vorgesehen wäre. Dies ist möglich, weil man bei jedem JavaScript-Objekt beliebig Unterobjekte hinzufügen kann, wie bereits in Abschnitt 2.1.1 auf Seite 3 erwähnt. Es gibt keine Klassen-Definition, wie beispielsweise in Java, die einen festen Rahmen darstellt, in dem der Umfang eines Objekts beschrieben wird. Dadurch sind Objekte in der Lage, zu fremden Objekten Funktionen hinzuzufügen.

Dieses Vorgehen hat den Vorteil, dass es sehr einfach und anfängerfreundlich ist. Allerdings erlaubt es immer nur die Definition eines Event-Handlers. Es können also nicht zwei Beobachter auf dasselbe Ereignis horchen, es sei denn, ein Beobachter weiß vom anderen und ruft ihn auf, sobald die von ihm definierte Event-Handler-Methode ausgeführt wird. Eine solche Abhängigkeit ist in einem einigermaßen komplexen System nicht mehr akzeptabel, da sie zu fehleranfällig und schlecht wartbar ist.

Aus diesem Grund habe ich die `EventDispatcherFactory` entwickelt, mit deren Hilfe `EventDispatcher` erzeugt werden können. Diese speziellen Event-Handler-Methoden handeln nach dem Beobachter-Muster, d. h. es werden auch 1-zu-n-Abhängigkeiten unterstützt und Objekte können sich sowohl an- als auch wieder abmelden. Damit besteht eine Entkopplung zwischen Subjekt und Beobachter sowie zwischen den Beobachtern untereinander.

Außerdem werden nicht einfache Handler-Methoden registriert, sondern Handler-Objekte, die eine bestimmte Methode implementieren müssen. Die Methoden, die das Ereignis verarbeiten, gehören damit nicht zum Subjekt, sondern zum Beobachter, so wie es das Beobachter-Muster vorsieht.

Die Implementierung nutzt dazu eine Eigenart von JavaScript aus. Wie in Abschnitt 2.1.1 auf Seite 3 erläutert, ist in JavaScript alles ein Objekt, auch Methoden. Und an jedes Objekt können weitere Unterobjekte angehängt werden. Das bedeutet, dass eine Methode selbst wieder Methoden haben kann. Genau das ist beim `EventDispatcher` der Fall: Ein `EventDispatcher` definiert eine `register`- und eine `unregister`-Methode, mit deren Hilfe sich Beobachter an- und abmelden können. Der `EventDispatcher` ist gleichzeitig die Event-Handler-Methode, die das Ereignis entgegennimmt und an alle registrierten Beobachter verteilt.

[3] Engl. „observer“: Beobachter

```
// HTML-Element erzeugen
var htmlElem = document.createElement("img");

// EventDispatcher hinzufügen
htmlElem.onclick =
  util.EventDispatcherFactory.createEventDispatcher("onElemClicked");

// Definition einer Beobachter-Klasse
MyObserver = function(htmlElem) {
  // Handler-Methode
  this.onElemClicked = function() {
    alert("Sie haben auf das Bild geklickt");
  }

  // Registrierung
  htmlElem.onclick.register(this);
}

// Erzeugung einer Beobachter-Instanz
var myObserver = new MyObserver(htmlElem);
```

Quelltext 6.2: Ereignisverarbeitung mit Hilfe der `EventDispatcherFactory`

Diese Konstruktion macht die Benutzung der `EventDispatcher` sehr elegant (siehe Quelltext 6.2): Man erzeugt mit Hilfe der `EventDispatcherFactory` einen `EventDispatcher` und nutzt diesen als Event-Handler-Methode. Nun kann man Beobachter definieren, die Handler-Methoden bereitstellen und sich beim `EventDispatcher` registrieren. Im Gegensatz zu herkömmlichen Event-Handler-Methoden sind diese Beobachter nun in der Lage auf ihren eigenen Kontext, also beispielsweise auf ihre privaten Variablen, zuzugreifen, da die Handler-Methode Teil des Beobachters und nicht des Subjekts ist.

Dadurch, dass bei der Erzeugung des `EventDispatchers` der Name der Handler-Methode angegeben wird, kann ein Beobachter auch auf mehrere verschiedene Ereignisse horchen. Er muss nur für jeden Typ die entsprechende Handler-Methode definieren.

6.4 Die WYSIWYG-Anzeige

Wie bereits in Abschnitt 3.2 auf Seite 17 erklärt, sollen in der Vorschau die statischen Teile der Report-Vorlage so gezeigt werden, wie sie auch im fertigen Report aussehen werden, während die dynamischen Teile, also die Velocity-Anweisungen, von Platzhaltern repräsentiert werden sollen.

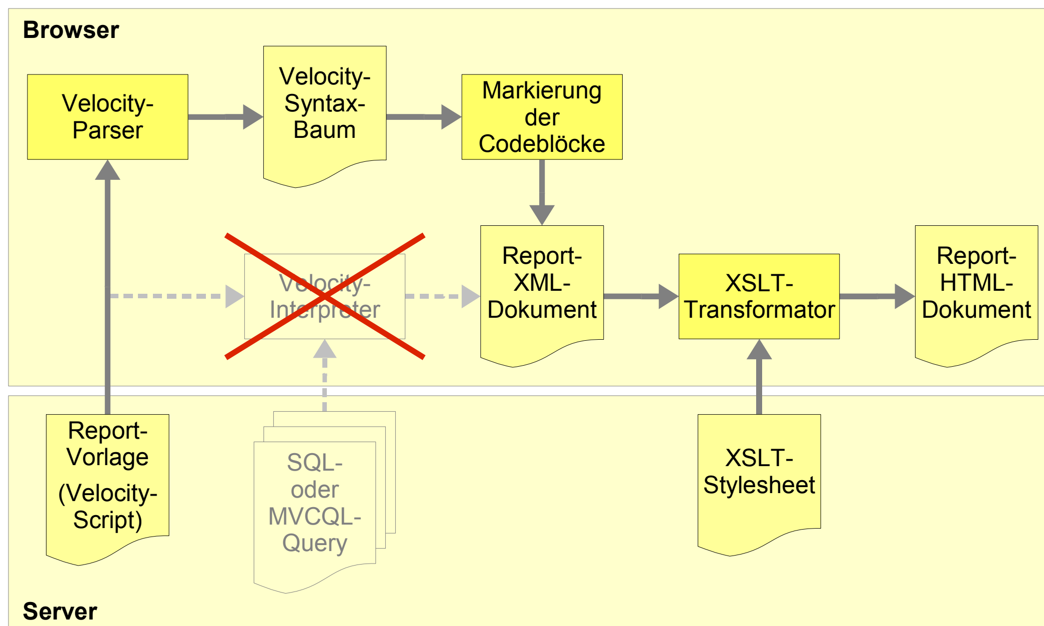

Abbildung 6.3: Flussdiagramm – Erstellung der Vorschau im Report-Editor

Der Abschnitt 4.1 auf Seite 18 erklärt, dass eine Report-Vorlage im Grunde ein Velocity-Script ist und wie der PTV-Reporter daraus einen konkreten Report generiert. Damit der Editor in der Vorschau die statischen Teile der Report-Vorlage korrekt anzeigen kann, wird sie auf dem gleichen Wege erzeugt wie im PTV-Reporter: über eine XSLT-Transformation. Der einzige Unterschied besteht darin, dass die komplette Verarbeitung auf dem Client stattfindet und dass der Velocity-Code nicht ausgeführt, sondern innerhalb von Platzhaltern angezeigt wird.

Wie funktioniert das? Die Report-Vorlage wird nicht wie im PTV-Reporter durch einen Velocity-Interpreter verarbeitet. Dadurch werden auch die Queries nicht ausgeführt, d. h. es werden keine Daten in die Ausgabe eingearbeitet (siehe Abbildung 6.3).

Statt dessen wird die Report-Vorlage von einem Velocity-Parser verarbeitet (Details siehe Abschnitt 6.5 auf der nächsten Seite). Das Ergebnis dieses Schrittes ist ein Velocity-Syntax-Baum. Dieser wird nun dazu verwendet, den Velocity-Code in der Report-Vorlage durch Code-Tags zu markieren. Dabei kommt ein Report-XML-Dokument heraus. Es handelt sich dabei um die gleiche Art XML-Dokument wie im PTV-Reporter, mit dem Unterschied, dass überall dort, wo in der Report-Vorlage Daten eingebunden werden, keine Daten sondern Code-Tags stehen.

Bei der anschließenden XSLT-Transformation kann daher das gleiche XSLT-Stylesheet eingesetzt werden, es muss nur um die Verarbeitung der Code-Tags erweitert werden.

Tatsächlich mussten aus dem Original-XSLT-Stylesheet noch XSLT-Extensions entfernt werden, über die auf Java-Extensions zugegriffen wurde. Da die XSLT-Transformatoren der Browser keine Java-Extensions unterstützen, war diese Änderung nötig.

Alle Erweiterungen, die nötig waren, um die Code-Tags zu verarbeiten, wurden in ein weiteres XSLT-Stylesheet ausgelagert. Dieses importiert das von den Extensions befreite Original-XSLT-Stylesheet und überschreibt dort nur wenige Details des Originals.

Beide Systeme nutzen also das Original-XSLT-Stylesheet, mit dem Unterschied, dass beim Stylesheet des Report-Editors die Java-Erweiterungen entfernt wurden. Diese Änderung lässt sich durch die Ausführung von wenigen Ersetzungsregeln erreichen, so dass sich dieser Schritt auch leicht automatisieren lässt. Alle weiteren Anpassungen werden durch das zweite Stylesheet definiert. Damit werden Redundanzen vermieden und die exakt gleiche Darstellung der statischen Teile ist garantiert.

Während der Bearbeitung im Report-Editor liegt die Report-Vorlage nur als Report-XML-Dokument vor. Das Parsen und Markieren der Code-Blöcke muss also nur ein einziges Mal, beim Laden eines Reports, durchgeführt werden. Alle Änderungen, die der Benutzer über den Editor in der Report-Vorlage vornimmt, werden direkt in diesem XML-Dokument umgesetzt. Damit ist zur Aktualisierung der Vorschau lediglich eine XSLT-Transformation notwendig.

6.5 Der Velocity-Parser

Wie bereits in Abschnitt 6.4 auf Seite 58 erklärt, wird auf Client-Seite ein Velocity-Parser benötigt, um den Velocity-Code innerhalb einer Report-Vorlage zu identifizieren. So kann dieser in einem weiteren Schritt durch Code-Tags markiert werden, was wiederum die Grundlage der Vorschau ist.

Parser für kontextfreie Sprachen (Chomsky-Typ 2) werden in den meisten Fällen implementiert, indem ein endlicher Automat für die Erkennung der Grundsymbole eingesetzt wird, die dann mit Hilfe des rekursiven Abstiegs in einem Syntax-Baum angeordnet werden. Diese Systeme haben den Vorteil, dass sie sehr schnell sind. Im Gegenzug jedoch stellen sie ein monolithisches Programm dar, in dem die Eigenheiten der jeweiligen Sprache hartcodiert enthalten sind. Sie sind daher schlecht erweiterbar und anpassbar und sie sind aufgrund ihrer monolithischen Struktur schlecht wartbar.

Aus diesen Gründen gibt es für andere Sprachen wie C oder Java Parser-Generatoren, die den Quelltext für einen Parser aus einer formalen Grammatik-Beschreibung gene-

©2005 Tilman Schneider

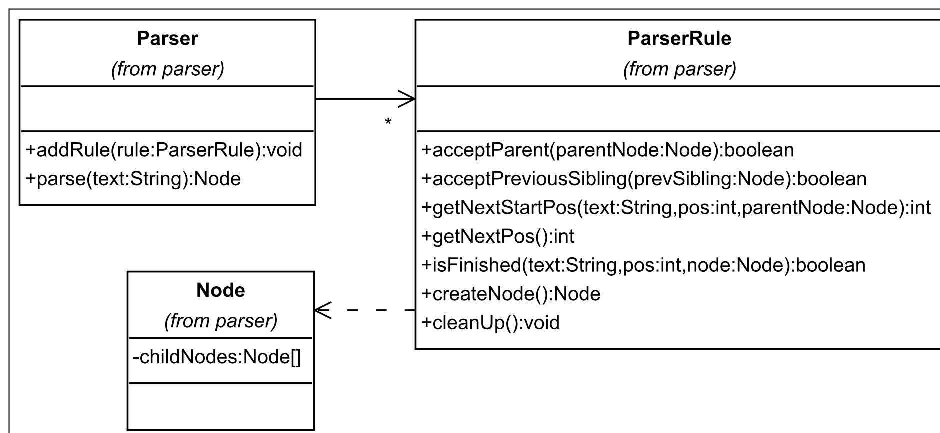

Abbildung 6.4: Klassendiagramm des Parsers

rieren. Beispiele sind die freie YACC-Implementierung Bison[4] für C oder der ebenfalls freie Generator JavaCC[5] für Java. Für JavaScript gibt es solche Generatoren jedoch nicht.

Um dennoch eine schlanke, gut wartbare Lösung zu erhalten, funktioniert der von mir entwickelte Parser mit einem Regelsystem, das Codeteile mit Hilfe von Regulären Ausdrücken erkennt. Dadurch besteht der eigentliche Parser aus einem schlanken Kern, der mit einer Reihe von Regeln gespeist wird.

Der Parseprozess verläuft dann wie folgt: Die Regeln werden nacheinander gefragt, wo sie im Code als nächstes zutreffen. Diejenige Regel, die als nächstes zutrifft, wird als die passende Regel angesehen und dazu verwendet, eine bestimmte Code-Struktur zu identifizieren. Die Regel kann nun entscheiden, ob sie weitere Code-Strukturen enthält und wann sie zu Ende ist. Außerdem kann jede Regel entscheiden, ob sie innerhalb einer bestimmten Code-Struktur anwendbar ist (siehe Abbildung 6.4). Auf diese Weise werden geschachtelte Strukturen erkannt.

Ein Beispiel: Im Quelltext „`#set($var = "value")`“ erkennt die Anweisungsregel zunächst den Anfang einer Anweisung „`#set(`“. Die öffnende Klammer am Ende zeigt ihr, dass sie weitere Strukturen enthält. Alle Strukturen, die nun gefunden werden, werden also der Anweisung als Kinder zugeordnet. Als nächstes erkennen die Variablenregel das „`$var`“, die Operatorregel das „=“ und die Konstantenregel das „`"value"`“, wobei die äußere Anweisungsregel nach jeder Regel gefragt wird, ob die Schachtelung

[4] Siehe `http://www.gnu.org/software/bison`

[5] Siehe `http://javacc.dev.java.net`

nun fertig ist. Dies bejaht sie erst nach der Konstante, wenn sie die schließende Klammer erkennt.

Die gesamte Syntax kann also durch eine Reihe von recht einfachen Regeln definiert werden. Dadurch ist sowohl der Parser als auch die einzelne Regel sehr einfach und damit gut wartbar. Außerdem kann dieses System leicht für andere Sprachen genutzt werden. Es muss lediglich ein anderer Satz von Regeln verwendet werden.

Jeder Text, der keiner Regel zugeordnet werden kann, wird der aktuellen Struktur als Textstruktur zugeordnet. Ein anschließender Syntax-Check kann dann alle Textstrukturen innerhalb von Anweisungen, die nicht nur Whitespace enthalten, als Syntaxfehler markieren. Text außerhalb von Anweisungen ist in Velocity erlaubt.

Damit nicht in jeder Runde für jede Regel ein Regulärer Ausdruck auf den verbleibenden Code angewendet werden muss, merkt sich jede Regel ihren zuletzt ermittelten Treffer. Sie wendet erst dann erneut ihren Regulären Ausdruck an, wenn der Parser bereits über diese Trefferstelle hinaus fortgeschritten ist. Auf diese Weise werden nur dann unnötig Reguläre Ausdrücke angewendet, wenn bestimmte Strukturen in anderen Strukturen eingeschlossen sind, wie es beispielsweise bei auskommentiertem Code der Fall ist.

Ein Beispiel: Im Quelltext „`#* $a *# $b = $c`“ wird in der ersten Runde die Kommentarregel an Position 0 einen Treffer liefern, die Variablenregel an Position 3 und die Operatorregel an Position 12. Der Parser wird daraufhin den Kommentar als gülig ansehen, weil sein Treffer der früheste ist. Die Kommentarregel ist an Position 8 zu Ende.

In der nächsten Runde wird die Kommentarregel keinen Treffer mehr liefern. Die Variablenregel wird feststellen, dass ihr letzter Treffer vor der Parserposition liegt ($3 < 8$), so dass sie ihren Regulären Ausdruck erneut anwenden wird, welcher an Position 9 auch schon anschlägt. Die Operatorregel hingegen stellt fest, dass ihr Treffer bei Position 12 immer noch aktuell ist. Der Parser erkennt damit die Variable „`$b`“. In den folgenden Runden werden noch der Operator „=“ und die Variable „`$c`“ erkannt. Es wurde also nur einmal, im Falle der auskommentierten Variablen „`$a`“, unnötigerweise ein Regulärer Ausdruck angewendet.

Im Grunde arbeitet dieser Parser ähnlich wie ein klassischer Parser. Die Grundsymbole werden durch Reguläre Ausdrücke erkannt, welche nichts anderes als endliche Automaten sind. Und der rekursive Abstieg wird ebenso vorgenommen, er wird lediglich über Regeln definiert, nicht über ein monolithisches Programm. Über die Regeln wird jedoch eine Kapselung der Spracheigenheiten erreicht. Zusätzlich vereint eine Regel die Syn-

tax eines Grundsymbols mit seiner Bedeutung, was beim herkömmlichen Ansatz nicht der Fall ist. Dort sind der endliche Automat und der rekursive Abstieg zwei getrennte Teile.

Der hier vorgestellte Parse-Algorithmus hat zwar im schlechtesten Fall eine Laufzeit von $O(n^2)$, doch das Beispiel zeigt, dass er im Regelfall eine annähernd lineare Laufzeit zeigen wird. Trotz allem ist er natürlich langsamer als die Kombination von endlichem Automat und rekursivem Abstieg, welche eine garantierte Laufzeit von O(n) hat[6]. Da der Parser jedoch nur dazu gebraucht wird, eine Report-Vorlage ein einziges Mal beim Laden zu parsen, kann das ohne Probleme in Kauf genommen werden. Im Gegenzug bekommt man einen schlanken, wiederverwendbaren und leicht wartbaren Parser.

6.6 Server-Seite

6.6.1 Client-Server-Kommunikation

Die Client-Server-Kommunikation erfolgt über einen Mechanismus, der von meinem Betreuer, Andreas Junghans, entwickelt wurde. Dieser erlaubt den generischen Zugriff auf entfernte Java-Objekte und den Austausch komplexer Objekt-Strukturen. Die dazu genutzten Techniken, JSON und BeanUtils, sollen nun vorgestellt werden.

JSON[7] ist ein Datenaustauschformat, das Objekte in JavaScript-Notation serialisiert. Damit ist es natürlich prädestiniert für den Datenaustausch mit JavaScript-Clients, da die serialisierten Daten einfach nur mit Hilfe der `eval`-Funktion ausgeführt werden müssen, um wieder JavaScript-Objekte zu erhalten.

JSON unterstützt eine sehr große Auswahl an Programmiersprachen, unter anderen auch Java. Mit Hilfe dieses Java-Adapters können primitive Datentypen, Arrays und Maps in eine JSON-Repräsentation und wieder zurück gewandelt werden. Die Daten können dabei beliebig geschachtelt werden. Es können also beispielsweise Arrays von Maps, die wieder Maps enthalten, serialisiert werden. Die Daten müssen jedoch zyklenfrei sein.

[6] Endliche Automaten haben eine Laufzeit von O(n). Ein LL-Parser mit einem Lookahead von einem Grundsymbol (LL(1)-Parser) ist für alle geläufigen Programmiersprachen ausreichend. Der rekursive Abstieg ist eine Möglichkeit, einen LL(1)-Parser zu implementieren und hat ebenfalls eine Laufzeit von O(n).

[7] Siehe `http://json.org`

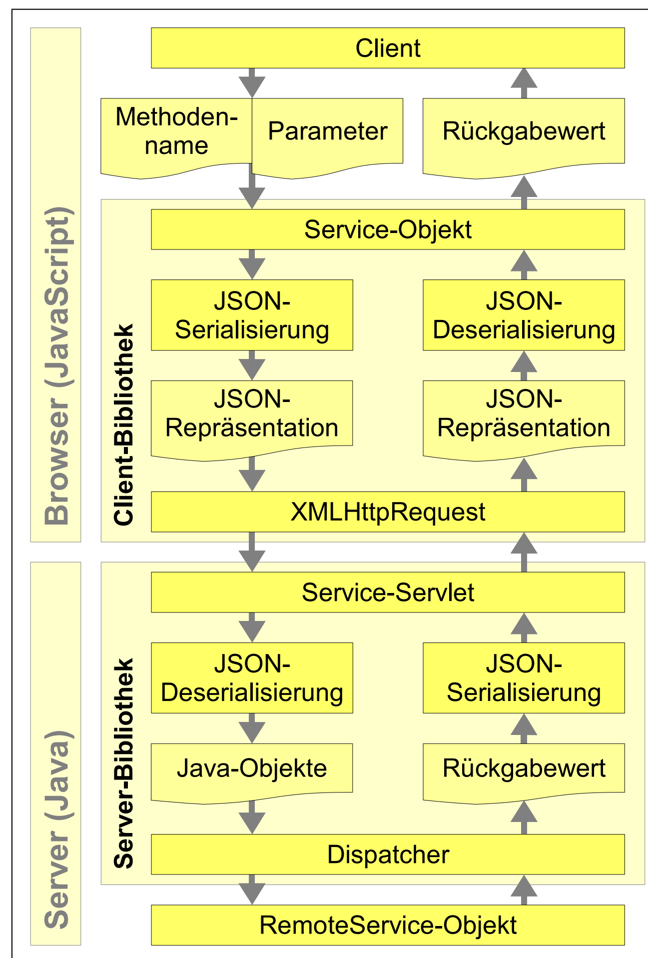

Abbildung 6.5: Aufrufstack der Client-Server-Kommunikation

BeanUtils[8] ist eine vom Jakarta Apache Projekt entwickelte API, die den Zugriff auf Java-Beans stark vereinfacht. Für die Introspektion bietet sie eine Wrapper-API um die Java Reflections API, deren Benutzung wesentlich eleganter ist. Beim Setzen von Properties oder beim Aufruf von Methoden sucht BeanUtils die passende Methode einer Java-Bean heraus und wandelt notfalls Parameter um. So muss man sich nicht darum kümmern, ob ein Getter nun mit „`is`“ oder „`get`“ anfängt und man kann auch ein `Integer`-Objekt übergeben, obwohl es nur einen Setter für den primitiven Typ `int` gibt.

Der von meinem Betreuer entwickelte Mechanismus kombiniert nun JSON, BeanUtils und den XMLHttpRequest[9], um einen generischen Zugriff auf serverseitige Java-

[8] Siehe `http://jakarta.apache.org/commons/beanutils`

[9] Details zum XMLHttpRequest siehe Abschnitt 2.2 auf Seite 12

Objekte von JavaScript aus zu ermöglichen. Auf JavaScript-Seite hat er dazu eine Client-Bibliothek entwickelt, die den gesamten Zugriff kapselt. Auf Java-Seite kommt ein Servlet zum Einsatz, das Anfragen entgegennimmt, den gewünschten Methodenaufruf ausführt und das Ergebnis zum Client zurückschickt (siehe Abbildung 6.5 auf der vorherigen Seite).

Ein entfernter Methodenaufruf funktioniert wie folgt: Auf JavaScript-Seite erzeugt man einmalig ein Service-Objekt für jede Java-Klasse, auf die man zugreifen möchte. Man muss dazu nur den Namen der gewünschten Klasse übergeben. Dieses Service-Objekt bietet zwei Methoden, eine für den synchronen und eine für den asynchronen Aufruf einer entfernten Methode. Jede dieser Methoden nimmt den Methodennamen und eine Liste von Parametern entgegen. Die Methode für den asynchronen Aufruf erwartet zusätzlich noch einen Handler, der den Rückgabewert bzw. die Exception entgegennimmt. Der Methodenname und die Parameter werden nun mit Hilfe von JSON serialisiert und per XMLHttpRequest zum Server gesendet.

Auf dem Server werden die serialisierten Objekte in Java-Objekte gewandelt. Ein Dispatcher sucht nun das passende Java-Objekt heraus, bzw. erzeugt eine neue Instanz, falls es noch kein solches geben sollte, und führt mit Hilfe der BeanUtils den gewünschten Methodenaufruf durch. Das Resultat, also der Rückgabewert oder die geworfene Exception, wird mit Hilfe von JSON serialisiert und an den Client zurückgegeben. Der Dispatcher wandelt dabei auch automatisch Java-Bean-Objekte um, so dass neben primitiven Typen, Arrays und Maps, die JSON schon von Haus aus unterstützt, auch Java-Beans zurückgegeben werden können. Auch hier kann beliebig geschachtelt werden, so lange alles zyklenfrei ist.

Auf der Client-Seite wird das Ergebnis schließlich zu JavaScript-Objekten deserialisiert und zurückgegeben.

Damit ein Client nicht jede beliebige Methode einer beliebigen Klasse aufrufen kann, muss eine Klasse, die von außen zugänglich sein soll, die Schnittstelle `RemoteService` implementieren. Diese Schnittstelle enthält keine Methoden. Sie dient nur dazu, die Klasse zu markieren. Sie wird daher „Tagging-Interface“ genannt[10]. Außerdem muss jede Methode, die von außen zugänglich sein soll, die Exception `RemoteServiceException` werfen. Auch diese Exception dient in erster Linie der Markierung von Methoden, die von außen angesprochen werden dürfen, sie darf aber auch wirklich geworfen werden. Damit ist sichergestellt, dass nur markierte Methoden von markierten Klassen nach außen hin zugänglich sind.

[10] Java nutzt bei der Objekt-Serialisierung die Schnittstelle `java.io.Serializable` für einen ähnlichen Zweck.

Die Client-Server-Kommunikation bietet auch elegante Lösungen zur Initialisierung von RemoteService-Objekten und zum Zugriff auf die Kontextobjekte des Servlets, aber das würde an dieser Stelle zu weit führen.

Um die Client-Server-Kommunikation zu nutzen, muss man auf der Server-Seite lediglich eine Klasse schreiben, die einen Standardkonstruktor hat, die Schnittstelle `RemoteService` implementiert und die gewünschten Service-Methoden anbietet. Nun kann man sich auf Client-Seite ein Service-Objekt erzeugen und die Methoden der Klasse aufrufen. Die gesamte Kommunikation erfolgt vollkommen generisch, es müssen keine Stubs oder Skeletons generiert werden und keine Beschreibungsdateien erzeugt werden, wie es bei anderen Techniken für entfernte Methodenaufrufe nötig ist.

6.6.2 Erweiterung des PTV-Reporters

Die Report-Vorlagen werden serverseitig vom PTV-Reporter verwaltet. Da dieser bisher nur lesend auf Vorlagen zugegriffen hat, musste er erweitert werden, um auch die vom Report-Editor benötigten Grundfunktionen zu beherrschen.

Die PTV AG nutzt zur Entwicklung ihrer Server-Komponenten einen Model-Driven-Architecture-Ansatz. Dabei werden alle Klassen, die von außen zugänglich sein sollen, in einem UML-Editor entworfen. Aus dem so entstandenen Modell wird mit Hilfe eines Maven-Skripts eine Reihe von Zugriffsmöglichkeiten generiert. Unter anderem beinhaltet das den Zugriff über RMI und Webservices. Dieses Skript generiert die besagten Zugriffsklassen, es kompiliert das gesamte Projekt und packt schließlich alle benötigten Komponenten in ein knapp 30 MB großes `ear`-Archiv, das schließlich direkt beim J2EE-Server installiert werden kann.

Leider benötigt dieser ganze Build-Prozess stolze 30 Minuten. Bei der Entwicklung kommt es vor, dass man häufig kleine Änderungen vornimmt, die man dann testen muss. Gerade bei der Analyse und Korrektur von Bugs ist ein solches Vorgehen oft unumgänglich. Wenn man nach zwei Minuten Entwicklung 30 Minuten warten muss, bis man die Änderung testen kann, dann ist das für eine effiziente Entwicklung inakzeptabel.

Ich habe daher zunächst einmal ein Ant-Skript geschrieben, das nur die jeweils geänderten Quellen übersetzt und ein zuvor erstelltes `ear`-Archiv damit aktualisieren und auch gleich beim J2EE-Server installieren kann. Dieses Skript hatte dadurch eine Laufzeit von unter einer Minute, so dass wieder ein effizientes Arbeiten möglich war.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<report id="mytemplate">
  <contents template-language="vtl">
    <![CDATA[
      <h1>Ein Report</h1>

      #set($results = $query.executeQuery("customers"))
      <table class="ptv-table" id="test-table">
        <header-row>
          <column>Name</column>
        </header-row>
        #foreach ($customer in $results)
        <row>
          <column>$customer.name</column>
          <column format="number">$customer.turnover</column>
        </row>
        #end
      </table>
    ]]>
  </contents>
</report>
```

Quelltext 6.3: Eine Report-Vorlage eingebettet in einem XML-Containern

Der PTV-Reporter legt Report-Vorlagen im Dateisystem in XML-Containern ab, d. h. das Velocity-Script, also die eigentliche Vorlage, ist in ein XML-Gerüst eigebettet. Dieses Gerüst definiert im Wesentlichen die ID der Vorlage und ein `contents`-Tag, welches die eigentliche Vorlage als Text in einer `CDATA`-Sektion enthält (siehe Quelltext 6.3). Durch dieses Konstrukt soll ermöglicht werden, später einmal neben Velocity-Skripten auch andere Formate als Vorlagen nutzen zu können. Der Dateiname muss identisch zur ID sein, plus die Erweiterung „`.xml`“, so dass die ID im Grunde redundant vorliegt. Auch dies ist gewollt, weil man in der Lage sein will, auch Reports direkt aus Datenströmen zu generieren.

Der PTV-Reporter wurde um folgende Grundfunktionen erweitert:

- *Anlegen:* Es kann nun eine neue, leere Report-Vorlage erstellt werden. Dabei wird im Dateisystem eine neue Vorlage erstellt, die aus einem leeren XML-Container mit leerem `contents`-Tag besteht.
- *Laden:* Ermöglicht das Laden einer Report-Vorlage. Bisher war es nur möglich, einen Report zu generieren. Das Betrachten der Vorlage, also der Rohversion, wurde nicht benötigt.
- *Speichern:* Speichert eine Report-Vorlage. Hierbei wird unterschieden zwischen einer Speicherung, die der Report-Editor in regelmäßigen Abständen automatisch durchführt (Autosave) und einer vom Benutzer veranlassten Speicherung. Ein Au-

tosave erfolgt in einem separaten Autosave-Verzeichnis, wobei immer die letzten fünf Autosave-Versionen behalten werden. Bei einer normalen Speicherung werden die letzten fünf Versionen in einem separaten Backup-Verzeichnis aufgehoben, außerdem werden die Autosaves gelöscht.

- *Kopieren:* Kopiert eine Vorlage. Die Kopie ist identisch zum Original, abgesehen von der ID, die auf den neuen Namen angepasst wird.
- *Umbennen:* Gibt einer bereits vorhandenen Vorlage einen neuen Namen. Auch hier wird neben dem Dateinamen auch die ID des XML-Containers angepasst.
- *Löschen:* Löscht eine Vorlage. Wie auch beim Speichern werden hier die letzten fünf Versionen im Backup-Verzeichnis gesichert.

Bei jeder nicht-reversiblen Aktion wird also immer ein Backup angelegt. Von jeder ID können so bis zu fünf Backup-Versionen existieren. Dadurch kann ein versehentlicher Fehler leicht wieder rückgängig gemacht werden. Gleichzeitig wird vermieden, dass im Laufe der Zeit die Platte mit Backups vollgeschrieben wird.

Die Autosaves sind eine zusätzliche Sicherheitsfunktion. Alle fünf Minuten prüft der Report-Editor, ob im Dokument etwas verändert wurde. Wenn dem so ist, dann wird das Dokument automatisch als Autosave gespeichert. Falls der Client-Rechner abstürzen sollte oder falls sich der Report-Editor durch eine Fehlfunktion nicht mehr bedienen lassen würde, geht dennoch nicht viel Arbeit verloren.

6.6.3 Service-Servlet

Der Report-Editor wurde in einer eigenen Webanwendung realisiert. Sie enthält alle Ressourcen, die vom Client geladen werden, wie CSS-Stylesheets, Bilder, HTML-Dateien und den JavaScript-Code. Außerdem enthält sie das in Abschnitt 6.6.1 auf Seite 63 beschriebene Service-Servlet, über das der Client mit dem Server kommuniziert.

Die `RemoteService`-Klasse, die schließlich die Client-Anfragen entgegen nimmt, greift via Webservice auf den PTV-Reporter zu, um die letztendlichen Grundfunktionen auszuführen. Dadurch ist es möglich, den Report-Editor und den PTV-Reporter auf verschiedenen Servern auszuführen. Gleichzeitig ist eine klare Abgrenzung dieser beiden Anwendungen garantiert.

Kapitel 7

Ergebnisse

Mit dem Report-Editor ist eine Webanwendung entstanden, die sich so leicht und flüssig bedienen lässt wie eine Desktop-Anwendung (siehe Abbildung 7.1 auf der nächsten Seite). Gleichzeitig benötigt sie auf Client-Seite keinerlei Installations- oder Wartungsaufwand und setzt lediglich einen gängigen Browser voraus, wie er auf praktisch jedem System bereits installiert ist.

7.1 Vergleich mit der Aufgabenstellung

Zu Beginn der Arbeit wurden einige Anforderungen aufgestellt. Für jede dieser Anforderungen soll im Folgenden gezeigt werden, in wie weit sie erfüllt wurde.

Zwingende Anforderungen:

- *Kompatibel zum PTV-Reporter:* Der Report-Editor ist nicht nur kompatibel zum PTV-Reporter, er bettet sich auch in die gleiche Umgebung ein. Beide Webanwendungen stellen separate Module dar, so dass sie klar getrennt sind und prinzipiell auch auf verschiedenen Servern betrieben werden können. Der Report-Editor greift jedoch auf den PTV-Reporter zu, so dass keine redundante Datenhaltung nötig ist und Reports im laufenden Betrieb geändert werden können.
- *WYSIWYG:* Dieser Punkt wurde durch zwei Elemente erreicht: Zum einen bietet die Vorschau eine Ansicht der Report-Vorlage, durch die gleichzeitig das statische Layout und die dynamischen Code-Teile sichtbar werden. Zum anderen kann noch während der Bearbeitung über einen Knopf in der Toolbar ein Report mit echten Daten aus dem aktuellen Stand der Vorlage generiert werden.

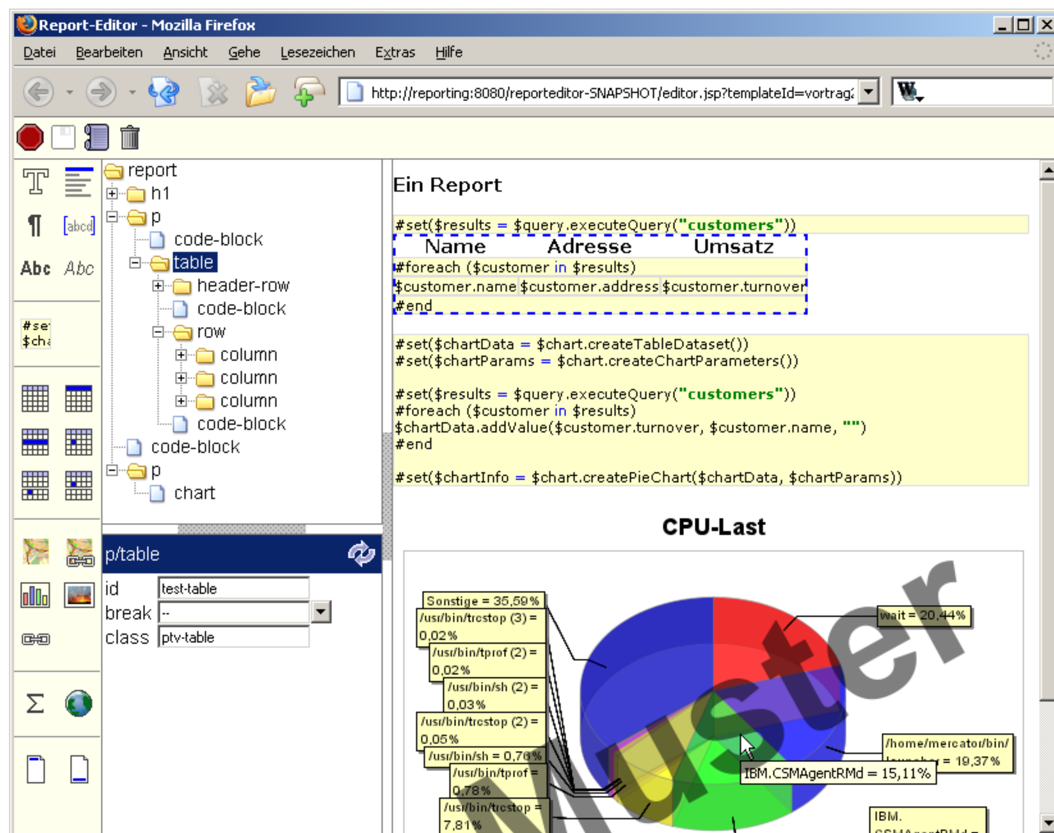

Abbildung 7.1: Der Report-Editor

Optionale Anforderungen:

- *Umschalten zwischen XSLT-Stylesheets:* Da es bisher nur ein XSLT-Stylesheet gibt, wurde das Umschalten zwar in Entwurf und Implementierung berücksichtigt, es fehlt jedoch eine GUI-Komponente, über die man tatsächlich umschalten könnte.

- *Umschalten zwischen Entwurf und fertiger Ausgabe:* Wie bereits erwähnt, kann über einen Knopf in der Toolbar aus der gerade bearbeiteten Vorlage ein Report generiert werden. Im Gegensatz zu der Entwurfsansicht, die durch die Vorschau gegeben ist, entspricht der generierte Report der fertigen Ausgabe.

- *Universeller Editor:* Die Code-Basis des Editors besteht aus drei Teilen: Einer Widget-Bibliothek, einem abstrakten Template-Editor und der PTV-Reporter-Anpassung. Der abstrakte Template-Editor ist so gestaltet, dass analog zur PTV-Reporter-Anpassung auch Anpassungen für andere Template-Sprachen entwickelt werden können. Diese Anpassungen müssen lediglich jenen Code beinhalten, der die Eigenheiten der jeweiligen Template-Sprache behandelt.

Aus der Analyse gewonnene Anforderungen:

- *Anpassbares Aussehen:* Das Aussehen der Oberfläche wird über die Verwendung von CSS-Stylesheets an zentraler Stelle bestimmt. Will man das Aussehen an Styleguides eines Kunden anpassen, so muss lediglich das CSS-Stylesheet verändert werden.

- *Nutzung von Webtechnologie:* Der Report-Editor ist eine Webanwendung. Diese Anforderung ist demnach erfüllt.

- *Mächtiger Velocity-Editor:* Der in der Vorschau gezeigte Velocity-Code wird mit Syntax-Highlighting dargestellt. Eventuelle Syntaxfehler werden dabei rot unterlegt. Insgesamt wurde der Schwerpunkt jedoch auf die Erstellung der XML-Elemente gelegt. So soll auch verhindert werden, dass Elemente über Velocity-Code generiert werden, obwohl sie auch in Form von XML-Elementen beschreibbar wären.

- *Gute Selbstbeschreibungsfähigkeit und Lernförderlichkeit:* Neben den üblichen Techniken der Selbstbeschreibung, wie z. B. Tooltips, wurden auch mögliche Fehlbedienungen ausgemacht und mit Hinweisen versehen, falls der Benutzer eine Fehlbedienung vornimmt. Beispielsweise zeigen die Elemente in der Dragbar einen Hinweisdialog, falls der Benutzer in der Annahme, es handle sich um Knöpfe, auf eines der Elemente klickt. Des Weiteren ist die gesamte Bedienung sehr einheitlich mit Drag-and-Drop realisiert, so dass eine hohe Lernrate zu erwarten ist.

Zusammenfassend kann man sagen, dass alle Anforderungen erreicht wurden, sogar die optionalen. Außerdem wurde durch den Report-Editor gezeigt, dass sich auch komplexe Oberflächen mit Hilfe der AJAX-Technologie entwickeln lassen.

7.2 Allgemeine Erfahrungen mit AJAX

Insgesamt gesehen ist die AJAX-Technologie sehr vielversprechend. Die heute üblichen Browser bieten eine Plattform, mit der weit mehr möglich ist, als von den meisten Webseiten genutzt wird. AJAX-Applikationen bringen sehr viele Vorteile mit sich: Sie sind Zero-Admin-Clients, unabhängig von bestimmten Plattformen oder Herstellern, sie bauen auf freien Standards auf, können zentral auf dem Server verwaltet werden, lassen sich flüssig bedienen und benötigen wenig Bandbreite.

©2005 Tilman Schneider

JavaScript zeigt unter den Browsern eine sehr hohe Kompatibilität. Die Sprache verhält sich bei jeder Implementierung praktisch gleich. Allerdings gibt es teilweise Unterschiede im Verhalten der einzelnen Browser-APIs. Dank der stabilen Sprache läßt sich jedoch Code entwickeln, der die Eigenheiten der einzelnen Browser kapselt, so dass Browser-Weichen nur auf unterster Ebene nötig sind.

Da JavaScript eine Skriptsprache ist, wird vom Client der Quellcode geladen und nicht etwa eine Binärversion. JavaScript-Anwendungen können daher sehr leicht von Dritten analysiert und in eigene Produkte übernommen werden. Man kann Reverse Engeneering zwar erschweren, indem man Kommentare entfernt oder einen Obfuscator verwendet, aber das ist nicht so sicher wie eine Binärversion.

Ein weiterer Nachteil ist, dass der JavaScript-Code im Browser in einer Sandbox-Umgebung ausgeführt wird, die aus Sicherheitsgründen nur eine festgelegte Menge von Operationen zulässt. Man hat dadurch zwar sehr viele, aber dennoch beschränkte Möglichkeiten. Beispielsweise hat man keinen Zugriff auf das lokale Dateisystem. Die Sicherheitsanforderungen, die an einen Browser gestellt werden, haben noch einen weiteren Effekt: Wenn Browser-Hersteller Sicherheitslücken in bestimmten API-Funktionen feststellen, dann kann es passieren, dass sie das Verhalten der API bei einer neuen Browser-Version ändern. So kommt es hin und wieder zu einer Verletzung der Rückwärtskompatibilität, so dass Code, der auf einer alten Browser-Version noch funktionierte, in einer neueren Version nicht mehr läuft. Das trifft zwar auch auf andere Plattformen zu, jedoch tritt dieser Effekt bei Browsern verstärkt auf. Dabei handelt es sich jedoch meist um Kleinigkeiten, die sich ohne großen Aufwand anpassen lassen.

Es gibt bereits sehr viele JavaScript-Bibliotheken. Allerdings gibt es kaum Bibliotheken, die sich leicht mit eigenen Entwicklungen kombinieren lassen. Viele bieten ein Framework, durch das vorgegeben wird, wie eine Webanwendung technisch umgesetzt werden muss. Sie verändern massiv den globalen Namensraum, so dass sie sich schlecht mit anderen Bibliotheken zusammen verwenden lassen. Dazu kommt, dass es kaum Bibliotheken gibt, die auf einem soliden objektorientierten Design basieren. Sie sind daher oft schlecht skalierbar und schlecht erweiterbar oder sie stellen sehr spezielle Anforderungen an den Server oder an die Webseite, in die sie integriert werden sollen. Viele Bibliotheken versprechen sehr viel auf ihrer Webseite, lassen dann jedoch jegliche Dokumentation vermissen. Somit fällt es sehr schwer, die Spreu vom Weizen zu trennen.

Die in Abschnitt 2.1 auf Seite 3 vorgestellten Techniken wurden von meinem Betreuer Andreas Junghans und mir aus verschiedenen Quellen zusammengetragen und teilweise noch abgeändert oder ergänzt. Es gibt keine Standardquelle, die beschreibt, wie man am besten JavaScript-Code entwickelt. Das ist wohl der Grund, warum viele JavaScript-Bibliotheken prozedural aufgebaut sind.

©2005 Tilman Schneider

Insgesamt gesehen kann man mit der AJAX-Technologie mächtige Clients entwickeln, die sehr viele Vorteile bieten. Da es jedoch kaum Bibliotheken gibt, die eine solide Widget-Sammlung bieten, ist die Entwicklung aufwändiger als bei herkömmlichen Benutzungsschnittstellen. Der Einsatz macht daher vor allem dann Sinn, wenn die Anwendung viel mit HTML-Dokumenten arbeitet, wie im Falle des Report-Editors, oder wenn man viele Clients hat, deren Wartung entweder kostenintensiv wäre oder einfach nicht möglich ist, weil der Dienstanbieter keinen Zugriff darauf hat, wie es beispielsweise im Internet der Fall ist.

©2005 Tilman Schneider

Kapitel 8

Ausblick

Anschließend an diese Arbeit wäre eine Weiterentwicklung auf allen drei Architektur-Ebenen denkbar: Die Widget-Bibliothek könnte um weitere Widgets erweitert und auch in anderen Anwendungen eingesetzt werden. Für den Template-Editor könnte eine Anpassung für PHP oder JSPs entwickelt werden und die PTV-Report-Anpassung könnte um Expertenfunktionen ergänzt werden.

Die Widget-Bibliothek und der abstrakte Template-Editor könnten als Open-Source-Projekt veröffentlicht werden. Wenn es gelingt, ein Projekt auf die Beine zu stellen, das ein solides Design hat, minimalinvasiv mit dem JavaScript-Namensraum umgeht und zusätzlich gut dokumentiert ist, dann könnte man mit Sicherheit viel Interesse wecken.

Es sind bereits viele große Namen dabei, Anwendungen auf der Basis von AJAX zu entwickeln: Google erprobt mit Projekten wie GMail, Google-Maps oder Google-Suggest[1] bereits den Einsatz in der breiten Öffentlichkeit. Auch Microsoft bietet mit MSN Virtual Earth[2] bereits einen Klon der Google-Maps. SAP setzt im Web Application Server seit der Version 6.40 im Rahmen seiner neuen „SAP WebDynpro“-Technologie auf dynamische HTML-Technik. Yahoo befindet sich mit einem neuen Mailsystem gerade in den Beta-Tests.

Auch die PTV AG plant, in Zukunft verstärkt auf diese Technik zu setzen, so dass diese Arbeit neben der Entwicklung des Editors auch der Evaluierung dieser Technik wegen interessant ist.

---

[1] GMail siehe `http://gmail.google.com`, Google-Maps siehe `http://maps.google.com`, Google-Suggest siehe `http://labs.google.com/suggest`

[2] MSN Virtual Earth siehe `http://virtualearth.msn.com`

Literaturverzeichnis

[AJAX05] Wikipedia-Artikel „AJAX“, Abruf am 07.09.05, http://de.wikipedia.org/wiki/AJAX.

[BusInt05] Wikipedia-Artikel „Business Intelligence“, Abruf am 07.09.05, http://de.wikipedia.org/wiki/Business_Intelligence.

[Cooper95] Alan Cooper: *About Face. The Essentials of User Interface Design*, 1995, IDG Books (ISBN 1-56884-322-4)

[CrysRep05] Homepage des Reporting-Systems „Crystal Reports“ von Business Objects, Abruf am 11.04.05 http://businessobjects.com.

[DesEss97] Susan Weinschenk, Pamela Jamar, Sarah C. Yeo: *GUI Design Essentials*, 1997, John Wiley & Sons (ISBN 0-4711-7549-8)

[Gamma96] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides: *Entwurfsmuster*, Deutsche Übersetzung: 1996, Addison-Wesley (ISBN 3-8273-1862-9)

[Garrett05] Jesse James Garrett: *Ajax: A New Approach to Web Applications*, 18.02.05, http://www.adaptivepath.com/publications/essays/archives/000385.php.

[JScript05] Wikipedia-Artikel „JavaScript“, Abruf am 02.05.05, http://de.wikipedia.org/wiki/Javascript.

[Jung05] Andreas Junghans: *Web-Browser als Rich Client*, Vortrag bei F&E-Runde des STZ-IDA am 17.02.05, http://stz-ida.de/download/forschung/fue/fue_20050217_ajunghans.pdf.

[RepNet05] Homepage des Reporting-Systems „Report Net“ von Cognos, Abruf am 11.04.05,
http://www.cognos1.de/app/1021/index.jsp.

[RepServ05] Homepage des Reporting-Systems „Microsoft SQL Server 2000 Reporting Services“ von Microsoft, Abruf am 11.04.05,
http://www.microsoft.com/germany/sql/technologien/reportingservices.

[SAS05] Homepage des Reporting-Systems „SAS 9“ von SAS, Abruf am 11.04.05,
http://www.sas.com/offices/europe/germany/solutions/te_s9_.html.

[Velocity05] Homepage von Velocity, einer auf Java basierenden Template-Engine, Abruf am 28.04.05,
http://jakarta.apache.org/velocity.

[WebEdit05] Homepage des Eclipse-Plugins Velocity Web Edit, Abruf am 20.05.05,
http://velocitywebedit.sourceforge.net.

Glossar

Ant[3] ist ein im Rahmen des Jakarta Apache Projekts entwickeltes Java-basiertes Build-Tool, das über XML-Dateien konfiguriert wird. Ant ist mittlerweile zum Quasi-Standard bei der Erstellung automatisierter Build-Umgebungen bei Java-Projekten geworden und wird auch außerhalb des Java-Umfeldes gerne eingesetzt. Siehe `http://ant.apache.org`

API ist die Abkürzung für Application Programming Interface. Eine API ist die Schnittstelle, die eine Applikation nutzen soll, um auf eine Bibliothek zuzugreifen.

Business Intelligence umfasst nach [BusInt05] „die analytischen Konzepte, Prozesse und Werkzeuge, um Unternehmens- und Wettbewerbsdaten in entscheidungsrelevantes Wissen zu transformieren. Es werden unternehmensinterne und -externe Daten als Quellen herangezogen."

DOM steht für Document Objekt Model und ist eine vom W3C standardisierte API, mit deren Hilfe HTML- und XML-Dokumente traversiert und verändert werden können. Die großen Browser bieten eine DOM-Unterstützung, so dass es möglich ist, Webseiten clientseitig zu verändern.

EJB ist die Abkürzung für Enterprise Java Bean. Eine EJB ist eine standardisierte Komponente in einer J2EE-Umgebung. Sie kann entweder lokal (innerhalb der VM) oder entfernt (also über Prozess- und Rechnergrenzen hinweg) aufgerufen werden. Der Zugriff und der Lebenszyklus einer EJB werden vom J2EE-Server kontrolliert, in dem sie läuft.
Siehe `http://java.sun.com/products/ejb`

EJBQL steht für EJB Query Language und ist eine an SQL angelehnte Abfragesprache für EJBs. Sie wurde von Sun entwickelt und ist Teil der EJB-Spezifikation.

[3] Engl. „ant": Ameise

Jakarta Apache Projekt ist eine Vereinigung von Entwicklern, die Open-Source-Software in Java entwickeln. Es ist Teil der Apache Software Foundation. Vom Jakarta Projekt stammen viele bekannte Systeme wie Jakarta Tomcat, Jakarta Regexp und Velocity.
Siehe `http://jakarta.apache.org`

J2EE steht für Java 2 Platform, Enterprise Edition und ist eine von Sun entwickelte Spezifikation einer Architektur, mit deren Hilfe verteilte Systeme entwickelt werden können. Die J2EE-Spezifikation beschreibt damit eine Java-Umgebung, die für den Einsatz in einer Unternehmensinfrastruktur optimiert ist. Verglichen zur Standard Edition (J2SE) und Micro Editon (J2ME) definiert sie die umfangreichste Java-Umgebung. Siehe auch EJB➦.
Siehe `http://java.sun.com/j2ee`

Gecko-Engine ist eine HTML-Rendering-Engine, die von der Mozilla Foundation entwickelt wird. Sie ist die Basis vieler freier Browser wie Mozilla Suite, Mozilla Firefox, Netscape Navigator, Camino und Epiphany.

Getter nennt man eine Methode, die den Wert eines Objektattributs nach außen zugänglich macht. In der Java-Bean-Konvention beginnen solche Methoden mit „get", bzw. „is". Beispiel: `public int getSize()` (siehe auch Setter➦).

IDE steht für Integrated Development Environment und bezeichnet Anwendungen zur Softwareentwicklung, in die alle wichtigen Entwicklungswerkzeuge (Editor, Compiler, Debugger) integriert sind.

Maven ist ein Build- und Projektverwaltungssystem. Zum Bauen von Programmpaketen können verschiedene Build-Sprachen eingebunden werden, beispielsweise Ant➦. Fertig gebaute Pakete werden in einem Repository veröffentlicht, so dass sie von anderen Projekten genutzt werden können, ohne dass diese ihre gesamten Abhängigkeiten bauen müssen. Maven wird daher oft im Firmenumfeld eingesetzt, um Build-Prozesse zu vereinheitlichen und zu beschleunigen.
Siehe `http://maven.apache.org`

MVCQL ist ein von MVCSoft entwickelter Dialekt der EJBQL➦.
Siehe `http://www.mvcsoft.com/overview.htm`

Regulärer Ausdruck (engl: „regular expression", kurz: „RegExp" oder „Regex") ist eine Zeichenkette, die mit Hilfe einer speziellen Syntax eine Menge von Zeichenketten beschreibt. Sie sind vergleichbar mit Zeichenketten, die Wildcards enthalten, nur dass man mit Regulären Ausdrücken noch komplexere Muster beschreiben kann. Reguläre Ausdrücke wurden vor allem durch das UNIX-Tool `grep`

bekannt, heute sind sie in sehr vielen Programmiersprachen und Anwendungen etabliert.

Setter nennt man eine Methode, die den Wert eines Objektattributs von außen veränderbar macht. In der Java-Bean-Konvention beginnen solche Methoden mit „set“. Beispiel: `public void setSize(int newSize)` (siehe auch Getter).

VTL steht für Velocity Template Language und ist – wie der Name schon sagt – die Template-Sprache von Velocity. Details zu Velocity siehe Abschnitt 2.3 auf Seite 14.

W3C steht für World Wide Web Consortium und ist eine Vereinigung zur Entwicklung freier und interoperabler Internettechnologien. Vom W3C stammen das HTTP-Protokoll und viele Internetformate, wie z. B. HTML, CSS, XML, XSL, PNG und SVG, um nur die wichtigsten zu nennen.
Siehe `http://www.w3.org`

Widget ist eine Zusammensetzung aus Window[4] und Gadget[5] und bezeichnet ein Bedienelement in einer graphischen Benutzungsoberfläche. Beispiele für Widgets sind Schaltflächen, Menüs, Texteingabefelder oder Scroll-Leisten. Ein häufig genutztes Synonym ist Control[6].

Wiki ist eine Webseite mit Beiträgen, die von ihren Benutzern nicht nur gelesen, sondern auch verändert werden können. Dadurch können falsche oder fehlende Informationen schnell korrigiert bzw. erweitert werden. Das wohl bekannteste Wiki ist die offene Enzyklopädie Wikipedia.
Siehe `http://wikipedia.org`

WYSIWYG ist die Abkürzung für „what you see is what you get“ und bezeichnet Editoren, die das Dokument während der Bearbeitung so zeigen, wie es in der letztendlichen Ausgabe aussieht.

[4] Engl. „window“: Fenster

[5] Engl. „gadget“: Ding, Gerät

[6] Engl. „control“: Regler, Bedienelement

©2005 Tilman Schneider

Index

AbstractEditorKit-Klasse, 38
Active-X-Komponente, 33
AJAX, 14
Autosave, 67

Backup-Verzeichnis, 68
BeanUtils, 64
Beispiel-Implementierungen, 36

Client-Server-Kommunikation, 63
Closure, 5

Dispatcher, 65
Drag-and-Drop, 52
Drag-Bar, 52
DragManager-Klasse, 45–47, 56
DragSource-Klasse, 45–47
Drill-Down, 15, 16
DropArea-Klasse, 46, 47
DropAreaManager-Klasse, 46, 47
Duck Typing, 10

Eclipse-Plugin, 19
ECMAScript, 3
Eigenschaft, 3
Eigenschaftenansicht, 28, 31, 42
Einfügen per Cursor, 29
Einfügen per Drag-and-Drop, 29
entfernter Methodenaufruf, 65
Event-Handler-Funktionen, 56
EventDispatcherFactory-Klasse, 57, 58

Flash-Plugin, 34

Gliederungsansicht, 28, 31
Google Maps, 14

Iframe, 53
Import, 7
Inlineframe, 53
instanceof, 9
Interview, 21

J2EE-Umgebung, 18
Java Reflections API, 64
Java-Bean, 64
Java-Plugin, 34
JavaScript, 3
JSON, 63

lokale Variable, 5

Mülleimer, 53
Master-Detail-Anordnung, 28
Maven-Skript, 66
Model-Driven-Architecture, 66

objektbasierte Sprache, 4
overflow=scroll, 54

Paket-Objekte, 7
Patch, 55
Personas, 23
private, 5
Proof-of-concept-Implementierungen, 36
PropertiesEditor-Klasse, 43, 44
PropertiesEditorManager-Klasse, 43, 44

PropertiesView-Klasse, 43, 44
PropertyField-Klasse, 43, 44
Prototyp-Objekt, 6
PTV-Reporter, 16, 18, 66

Query, 18

RemoteService-Klasse, 68
RemoteService-Schnittstelle, 65, 66
RemoteServiceException, 65
Report, 15
Report-Vorlage, 18
Report-XML-Dokument, 18
Reportgenerator, 15
ReportLoader-Klasse, 38
ReportModel-Klasse, 38
ReportTreeModel-Klasse, 38
RMI, 66

saveValues-Methode, 44
Schiebebalken, 52
Service-Objekt, 65
Swing, 34
SWT, 33

Tagging-Interface, 65
Technologiebruch, 34, 35
Template-Engine, 14
TemplateLoader-Klasse, 38
TemplateModel-Klasse, 40, 44
TemplateSelectionModel-Klasse, 43, 44
TemplateTreeModel-Klasse, 38
Typprüfung, 9

Velocity, 14
Velocity Web Edit, 19
Velocity-Skript, 15, 18
Vorlagen-Autor, 22
Vorschau, 28, 31

Walter Zorn, 55
Webservice, 66
Windows-Anwendung, 33
WYSIWYG-Ansicht, 28

XML-Container, 67
XMLHttpRequest, 12, 64
XSLT-Transformation, 18

©2005 Tilman Schneider