Programmierung, Anpassung & Integration

Magento

Das Handbuch für Entwickler

Roman Zenner, Vinai Kopp
sowie Claus Nortmann, Sebastian Heuer,
Dimitri Gatowski & Daniel Brylla
von Visions new media

O'REILLY®

Inhalt

KAPITEL 6

Angebote und Bestellungen

6.0 Einführung

Die Rezepte dieses Kapitels widmen sich der Frage, wie Sie Modifikationen durchführen können, die Angebote und Bestellungen in Ihrem Magento-Shop betreffen. So lernen Sie beispielsweise, wie Sie Bestelldaten individuell anreichern können, neue Bestellstatus einführen oder einen produktspezifischen Versandaufpreis integrieren.

6.1 Bestelldaten anreichern

Problem

Sie möchten der Bestellung ein Feld *campaign* hinzufügen, um die Kampagne festzuhalten, durch die der Kunde auf Ihren Shop gekommen ist, und ein weiteres Feld *check_status* bei der Adresse, um festzuhalten, ob die Adresse erfolgreich geprüft werden konnte.

Lösung

Diese Attribute fügen wir über ein Installationsskript innerhalb einer eigenen Magento-Extension ein. Damit Magento beim Konvertieren des Angebots in einen Auftrag Ihre zusätzlichen Attribute mitkopiert, müssen Sie die entsprechenden *Fieldsets* erweitern.

Diskussion

Beim Platzieren einer Bestellung in Magento wird die *Quote* in eine *Order* konvertiert. Eine Quote ist eine Entität, die den Warenkorb und alle Daten, die während des Checkout-Prozesses angegeben wurden (Rechnungs- und Versandadresse usw.), enthält. Sie ist sozusagen ein Angebot des Onlinehändlers an den Kunden, der dieses Angebot beim Bestellen annimmt, wodurch aus dem Angebot ein Auftrag wird. Entsprechend werden die Artikel im Warenkorb als Quote-Items gespeichert und beim Bestellen in Order-

Items umgewandelt. So verhält es sich mit Quote-Address (Rechnungs- und Lieferadresse) und Order-Address.

Die Daten, die beim Konvertieren des Angebots in den Auftrag übernommen werden, sind in sogenannten *Fieldsets* in der Konfigurationsdatenstruktur angelegt. Sie finden die Core-Fieldsets in der *config.xml*-Datei des Moduls *Mage_Sales*. Indem Sie die entsprechenden XML-Pfade in der Datei *config.xml* eines Ihrer Module erweitern (siehe nachfolgendes Codebeispiel), können Sie die bei einer Konvertierung berücksichtigten Attribute ergänzen. Tragen Sie Folgendes in die Datei *config.xml* ein:

```
<config>
    <!-- … -->
    <global>
        <fieldsets>
            <sales_convert_quote>
                <campaign><to_order>*</to_order></campaign>
            </sales_convert_quote>
            <sales_convert_quote_address>
               <check_status><to_order_address>*</to_order_address></check_status>
            </sales_convert_quote_address>
        </fieldsets>
    </global>
    <!-- … -->
</config>
```

Damit Magento beim Konvertieren des Angebots (also eines Quote-Models) in einen Auftrag (d.h. in ein Order-Model) Ihre zusätzlichen Attribute mitkopiert, müssen Sie die entsprechenden Fieldsets wie in obigem Beispiel erweitern. Der Kopiervorgang funktioniert nur, wenn diese Attribute für die Entitäten zuvor angelegt worden sind. Dies geschieht in dem Installationsskript des Moduls (*mysql4-install-0.1.0.php*):

```
$installer = $this;
$installer->addAttribute('quote', 'campaign', array());
$installer->addAttribute('quote_address', 'is_checked', array());

$installer->addAttribute('order', 'campaign', array());
$installer->addAttribute('order_address', 'is_checked', array());
```

Das Fieldset `sales_convert_quote` ist für das Konvertieren eines Angebots in einen Auftrag verantwortlich und enthält nun zusätzlich unser Attribut `campaign`. Das Fieldset `sales_convert_quote_address` legt fest, welcher Teil einer Adresse aus einem Angebot in eine Adresse eines Auftrags kopiert wird. Hier fügen Sie das Attribut `check_status` hinzu. Das entsprechende Fieldset für Bestellpositionen ist `sales_convert_quote_item`.

Es gibt jedoch kein Fieldset für die Konvertierung eines Produkts in ein Quote-Item. Wie in Rezept 6.3, »Einen produktspezifischen Versandaufpreis festlegen«, auf Seite 144, gezeigt wird, muss in diesem Schritt ein Attributwert von einem Produkt mittels eines Event-Observers auf das Quote-Item übertragen werden. Dazu bietet sich der Event `sales_quote_product_add_after` an.

Wie Sie sehen, ist Magento sehr flexibel, was das Anreichern von Objekten wie z.B. Aufträgen und Adressen um zusätzliche Daten angeht. Erweiterungen dieser Art sind sehr updatesicher, da der Sourcecode des Cores von Magento weder angefasst noch modifiziert werden muss. Jedoch ist es empfehlenswert, spezielle Präfixe für die Attributcodes zu verwenden, um Namenskonflikte mit neuen Attributen aus zusätzlichen Modulen von MagentoConnect oder dem Magento-Core zu vermeiden.

6.2 Einen zusätzlichen Status für Bestellungen hinzufügen

Problem

Ein Onlineversandhändler für Backwaren lässt jede Bestellung auf Mängel prüfen, bevor die Ware zum Käufer verschickt wird. Dieser Qualitätssicherungsprozess kann bis zu einem Tag dauern, sodass Kunden per E-Mail informiert werden sollen, wenn ihre Bestellung in der Qualitätssicherung landet.

Dieser Prozess soll in Magento abgebildet werden. Mitglieder des Teams für Qualitätssicherung sollen bei Bestellungen den Status »in Qualitätssicherung« setzen können.

Lösung

Zunächst erweitern Sie das Objekt *Order* um den Status »in Qualitätssicherung«. Anschließend führen Sie eine Methode ein, die eine Bestellung in den Status »in Qualitätssicherung« versetzt.

Diskussion

Um das Objekt *Order* zu erweitern, leiten Sie die Klasse `Mage_Sales_Model_Order` (*/app/code/local/Webkochshop/QualityAssurance/Model/Sales/Order.php*) ab und definieren einen neuen Status:

```
class Webkochshop_QualityAssurance_Model_Sales_Order
    extends Mage_Sales_Model_Order
{
    /**
     * Definition des neuen Order State Werts
     *
     * @const string
     */
    const STATE_QUALITY_ASSURANCE = 'qa';
}
```

Anschließend erstellen Sie eine Methode, die eine Bestellung in den Status »in Qualitätssicherung« versetzt. Die Methode soll vorher aber prüfen, ob die Bestellung diesen Zustand überhaupt annehmen kann – z.B. können noch nicht verarbeitete Bestellungen nicht geprüft werden:

```
/**
 * Prüfen, ob die Bestellung in den Quality Assurance State gesetzt werden kann
 *
 * @return bool
 */
public function canQa()
{
    if ($this->canUnhold())
    {
        return false;
    }

    if ($this->getState() === self::STATE_CANCELED ||
        $this->getState() === self::STATE_COMPLETE ||
        $this->getState() === self::STATE_CLOSED)
    {
        return false;
    }

    return true;
}

/**
 * Setzen der Order in den Quality Assurance State
 *
 * @return Mage_Sales_Model_Order
 */
public function qa()
{
    if (!$this->canQa())
    {
        Mage::throwException(
            Mage::helper('qa')->__('Quality assurance action is not available')
        );
    }

    $this->setState(self::STATE_QUALITY_ASSURANCE, true);
    return $this;
}
```

Damit Magento den Rewrite für das Order-Model auch berücksichtigt, muss jetzt der entsprechende Eintrag in die Modulkonfiguration, also in die Datei *app/etc/local/Webkochshop/QualityAssurance/etc/config.xml*, gemacht werden:

```
<?xml version="1.0"?>
<config>
    <modules>
        <Webkochshop_QualityAssurance>
            <version>0.1.0</version>
        </Webkochshop_QualityAssurance>
    </modules>

    <global>
        <models>
```

```
            <!-- Überschreiben des Magento Sales Order Model -->
            <sales>
                <rewrite>
                    <order>Webkochshop_QualityAssurance_Model_Sales_Order</order>
                </rewrite>
            </sales>
        </models>
        <helpers>
            <!-- Eintrag eines Helpers für die Übersetzungen -->
            <qa>
                <class>Webkochshop_QualityAssurance_Helper</class>
            </qa>
        </helpers>
    </global>
</config>
```

Nun brauchen Sie einen Button für die Bestellansicht im Admin-Panel, damit Mitarbeiter den neuen Status setzen können. Dazu überschreiben Sie den Konstruktor des Blocks, der für die Ansicht von Bestellungen im Admin-Panel dient, und fügen einen neuen Button ein, den Mitarbeiter anklicken können, um die Bestellung in die Qualitätssicherung zu schicken. Die Änderungen nehmen Sie in der Datei */app/code/local/Webkochshop/QualityAssurance/Block/Adminhtml/Sales/Order/View.php* vor.

```
<?php

class Webkochshop_QualityAssurance_Block_Adminhtml_Sales_Order_View
    extends Mage_Adminhtml_Block_Sales_Order_View
{
    /**
     * Hinzufügen des Quality Assurance-(qa-)Buttons
     */
    public function __construct()
    {
        parent::__construct();

        if ($this->getOrder()->canQa())
        {
            $this->_addButton('order_qa', array(
                'label'   => Mage::helper('qa')->__('Qa'),
                'onclick' => "setLocation('" . $this->getQaUrl() . "')",
            ));
        }
    }

    /**
     * Rückgabe der Quality Assurance-Action-URL
     *
     * @return string
     */
    public function getQaUrl()
    {
        return $this->getUrl('*/*/qa');
    }
}
```

Auch diese Klassenersetzung muss in der *config.xml* registriert werden:

```
    <global>
        <!--...-->
        <blocks>
            <!-- Überschreiben des Adminhtml Sales Order View-Blocks -->
            <adminhtml>
                <rewrite>
                    <sales_order_view>Webkochshop_QualityAssurance_Block_Adminhtml_Sales_
Order_View</sales_order_view>
                </rewrite>
            </adminhtml>
        </blocks>
        <!--...-->
    </global>
    <!--...-->
```

Damit nach Anklicken des Buttons auch etwas geschieht, müssen Sie eine Controller-Action anlegen, die dafür sorgt, dass der neue Bestellstatus gesetzt wird. Legen Sie dazu einen entsprechenden Controller in */app/code/local/Webkochshop/QualityAssurance/controllers/Adminhtml/Sales/OrderController.php an:*

```
<?php

/*
 * Einbinden der Elternklasse, da der Autoloader diese nicht automatisch
 * finden kann
 */
require_once 'Mage/Adminhtml/controllers/Sales/OrderController.php';

class Webkochshop_QualityAssurance_Adminhtml_Sales_OrderController
    extends Mage_Adminhtml_Sales_OrderController
{
    /**
     * Setzen der Bestellung in den Quality Assurance State
     */
    public function qaAction()
    {
        if ($order = $this->_initOrder())
        {
            try
            {
                $order->qa()->save();
                $this->_getSession()->addSuccess(
                    $this->__('Order was successfully put in quality assurance.')
                );
            }
            catch (Mage_Core_Exception $e)
            {
                $this->_getSession()->addError($e->getMessage());
            }
            catch (Exception $e)
            {
                $this->_getSession()->addError(
                    $this->__('Order was not put in quality assurance.'));
```

```
                Mage::logException($e);
            }

            $this->_redirect('*/sales_order/view',
                array('order_id' => $order->getId())
            );
        }
    }
}
```

Und auch das Überlagern des Controllers muss Magento bekannt gemacht werden. Fügen Sie folgenden Eintrag in die *config.xml* ein:

```
<config>
    <!--...-->
    <admin>
        <routers>
            <adminhtml>
                <args>
                    <modules>
                        <qa before="Mage_Adminhtml">Webkochshop_QualityAssurance_
                            Adminhtml</qa>
                    </modules>
                </args>
            </adminhtml>
        </routers>
    </admin>
</config>
```

Die vorbereitenden Maßnahmen sind damit abgeschlossen, lediglich der neue Status wird noch nicht angezeigt – Magento kennt bisher kein Etikett dafür. Fügen Sie also noch diesen letzten fehlenden XML-Abschnitt in die Konfiguration ein:

```
<global>
    <!--...-->
    <sales>
        <order>
            <states>
                <qa translate="label">
                    <label>Quality Assurance</label>
                    <statuses>
                        <qa default="1"/>
                    </statuses>
                    <visible_on_front/>
                </qa>
            </states>
            <statuses>
                <qa translate="label">
                    <label>Quality Assurance</label>
                </qa>
            </statuses>
        </order>
    </sales>
    <!--...-->
</global>
```

Durch die An- oder Abwesenheit des Tags `<visible_on_front/>` wird bestimmt, ob Bestellungen in diesem State dem Kunden in seiner Historie angezeigt werden.

Das Hinzufügen weiterer Bestellstatus ist mit Vorsicht zu genießen: Durch jeden weiteren Status steigt die Komplexität des Bestellprozesses und damit auch die Fehleranfälligkeit. Außerdem sollten Sie nicht in Versuchung kommen, aus Magento ein Warenwirtschaftssystem zu bauen, indem Sie anfangen, komplexe Geschäftsprozesse in den Onlineshop zu integrieren.

Vielmehr ist es sinnvoll, den Status der Bestellung von der Warenwirtschaft kontrollieren zu lassen und Magento als Informationsschnittstelle zwischen der Warenwirtschaft und dem Kunden zu nutzen. So bleiben komplexe und sensible Geschäftsprozesse in der Warenwirtschaft und die Kommunikation mit dem Kunden im Onlineshop.

Siehe auch

Die Extension *QualityAssurance* finden Sie im Download-Code zum Buch im Archiv *Webkochshop_QualityAssurance-0.1.0.zip*.

6.3 Einen produktspezifischen Versandaufpreis festlegen

Problem

Sie verkaufen einige Produkte, zu denen Sie gern individuelle Versandkosten angeben möchten, weil sie beispielsweise besonders sperrig sind und die generellen Versandkosten dafür nicht ausreichen.

Lösung

Über verschiedene Setup-Klassen fügen Sie den Entitäten `catalog_product`, `order` und `quote_item` jeweils ein neues Attribut hinzu. Außerdem legen Sie ein neues Total-Model an, das sowohl im Frontend als auch im Backend angezeigt wird.

Diskussion

Beginnen Sie zunächst wie immer damit, das Grundgerüst einer neuen Extension anzulegen, und speichern Sie es im Code-Pool *local* im Namespace *Webkochshop* unter dem Namen *Versandaufpreis*. Den Aufbau des Moduls beginnen wir, indem wir uns die beiden verschiedenen Setup-Skripte ansehen, mit deren Hilfe wir die nötigen neuen Attribute in die Datenbank speichern. Das erste Skript liegt in */sql/versandaufpreis_catalog_setup/mysql4-install-0.1.0.php* und liest sich wie folgt:

```
<?php

/**
 * @var Mage_Catalog_Model_Resource_Eav_Mysql4_Setup $installer
 */
$installer = $this;

$installer->startSetup();

$installer->addAttribute('catalog_product', 'shipping_surcharge', array(
    'label'           => 'Versandkosten Aufpreis',
    'type'            => 'decimal',
    'input'           => 'price',
    'required'        => '0',
    'is_configurable' => '0'
));

$installer->endSetup();
```

In bekannter Manier fügen Sie hier der Entität `catalog_product` ein neues Attribut mit dem internen Code `shipping_surcharge` hinzu. Dieses Attribut ist ein Dezimalwert und wird im Admin-Bereich mit einem Preis gefüllt, der den Versandkostenaufschlag für das jeweilige Produkt widerspiegelt. Da dieser Aufschlag jedoch an den verschiedensten Stellen im Frontend und im Backend angezeigt werden muss – denken Sie beispielsweise an die Bestellverwaltung und die Rechnungserstellung –, müssen wir dieses Attribut über die Sales-Setup-Klasse ebenfalls hinzufügen. (Da wir es also mit zwei verschiedenen Setup-Klassen zu tun haben, benötigen wir auch zwei verschiedene Setup-Skripte.) Dies geschieht über ein zweites Setup-Skript in */sql/versandaufpreis_sales_setup* mit dem folgenden Inhalt:

```
<?php

/**
 * @var Mage_Sales_Model_Mysql4_Setup $installer
 */
$installer = $this;

$installer->startSetup();

$installer->addAttribute('quote_item', 'shipping_surcharge', array(
    'label' => 'Versandkostenaufpreis',
    'type'  => 'decimal',
));

$installer->addAttribute('order_item', 'shipping_surcharge', array(
    'label' => 'Versandkostenaufpreis',
    'type'  => 'decimal',
));

$installer->addAttribute('order', 'base_shipping_surcharge', array(
    'label' => 'Basiswährung Versandkostenaufpreis',
    'type'  => 'decimal',
));
```

```
$installer->addAttribute('order', 'shipping_surcharge', array(
    'label' => 'Versandkostenaufpreis',
    'type'  => 'decimal',
));

$installer->addAttribute('invoice', 'base_shipping_surcharge', array(
    'label' => 'Basiswährung Versandkostenaufpreis',
    'type'  => 'decimal',
));

$installer->addAttribute('invoice', 'shipping_surcharge', array(
    'label' => 'Versandkostenaufpreis',
    'type'  => 'decimal',
));

$installer->addAttribute('creditmemo', 'base_shipping_surcharge', array(
    'label' => 'Basiswährung Versandkostenaufpreis',
    'type'  => 'decimal',
));

$installer->addAttribute('creditmemo', 'shipping_surcharge', array(
    'label' => 'Versandkostenaufpreis',
    'type'  => 'decimal',
));

$installer->endSetup();
```

Sie sehen hier, dass die Attribute `shipping_surcharge` und `base_shipping_surcharge` an verschiedenen Stellen hinzugefügt werden, um den Versandkostenaufschlag auch dort anzuzeigen, wo es erforderlich ist. Hierbei ist `shipping_surcharge` der Betrag in der Shopwährung, und `base_shipping_surcharge` stellt den Betrag in der Basiswährung des Shops dar. Letzerer wird gebraucht, da sich der Kurs in der Zukunft ja mal ändern könnte und sich der Betrag dann nie wieder ermitteln ließe. Wie diese einzelnen Attribute zusammenzuspielen, werden Sie in der weiteren Diskussion dieses Rezepts erfahren.

Verantwortlich für die Definition der Setup-Klassen, die den beiden gezeigten Setup-Skripten zugrunde liegen, ist wie so oft die Konfigurationsdatei *config.xml*. Der für uns relevante Bereich lautet:

```
<global>
    <!--...-->
    <resources>
        <versandaufpreis_catalog_setup>
            <setup>
                <module>Webkochshop_Versandaufpreis</module>
                <class>Mage_Catalog_Model_Resource_Eav_Mysql4_Setup</class>
            </setup>
        </versandaufpreis_catalog_setup>
        <versandaufpreis_sales_setup>
            <setup>
                <module>Webkochshop_Versandaufpreis</module>
                <class>Mage_Sales_Model_Mysql4_Setup</class>
```

```
            </setup>
        </versandaufpreis_sales_setup>
    </resources>
    <!--...-->
</global>
```

An dieser Stelle werden die oben beschriebenen Setup-Skripte eingebunden, und damit wird definiert, welche Setup-Klasse jeweils zur Ausführung verwendet werden soll.

Nachdem alle erforderlichen Attribute ihre neue Heimat in der Magento-Datenbank gefunden haben, widmen wir uns dem Anlegen eines neuen Total-Models, um den Aufschlag auch überall anzeigen zu können. Ein Total-Model wird dazu verwendet, eine Art von Summe in der Bestellauflistung auszudrücken. Wenn Sie sich beispielsweise den Warenkorb eines Shops vorstellen, stellen Einträge wie Mehrwertsteuer, Zwischensumme und Versandkosten, die sich aus den einzelnen Positionen der Bestellung errechnen, diese Summen dar (siehe Abbildung 6-1):

Zwischensumme	10,00 €
Versand & Bearbeitung (Flat Rate - Fixed)	5,00 €
Gesamtsumme	**15,00 €**

Abbildung 6-1: Anzeige der Total-Models im Warenkorb

Mit anderen Worten, es gibt ein Total-Model für die Steuer, eins für die Zwischensumme, eins für die Versandkosten usw. Diese Total-Models stehen in verschiedenen Bereichen des Shops zur Verfügung und können je nach Einsatzbereich ein- und ausgeblendet sowie umsortiert werden.

Für die standardmäßig in Magento vorhandenen Total-Models finden Sie im Admin-Bereich unter *Verkäufe* → *Reihenfolge der Gesamtbeträge des Bezahlvorgangs* eine einfache Sortiermöglichkeit.

Damit der von uns anstrebte Versandaufpreis nicht in der Gesamtsumme der Bestellung verschwindet, sondern explizit ausgewiesen wird, erstellen wir für ihn ein nagelneues Totals-Model. Sehen Sie sich den folgenden Ausschnitt der *config.xml* an:

```
<global>
    <!-- … -->
    <sales>
        <quote>
            <totals>
                <shipping_surcharge>
                    <class>versandaufpreis/quote_address_total_shipping_surcharge</class>
                    <after>shipping</after>
                    <before>grand_total</before>
                </shipping_surcharge>
            </totals>
        </quote>
```

```
            <order_invoice>
                <totals>
                    <shipping_surcharge>
                        <class>versandaufpreis/order_invoice_total_shipping_surcharge</class>
                        <after>shipping</after>
                        <before>grand_total</before>
                    </shipping_surcharge>
                </totals>
            </order_invoice>
            <order_creditmemo>
                <totals>
                    <shipping_surcharge>
                        <class>versandaufpreis/order_creditmemo_total_shipping_surcharge
                        </class>
                        <after>shipping</after>
                        <before>grand_total</before>
                    </shipping_surcharge>
                </totals>
            </order_creditmemo>
        </sales>
        <!-- … -->
    </global>
```

Insgesamt kommen also drei neue Total-Models aus unserem Modul zum Einsatz:

- *quote_address_total_shipping_surcharge* zur Anzeige im Checkout
- *order_invoice_total_shipping_surcharge z*ur Anzeige in der Rechnung
- *order_creditmemo_total_shipping_surcharge* zur Anzeige in Rückerstattungen

Über den oben gezeigten Code definieren Sie die zugehörigen Klassen der Total-Models und bestimmen außerdem über die `after`- und `before`-Tags, wie sie sich in die standardmäßig vorhandenen Total-Models einreihen. Wir haben uns in diesem Fall dafür entschieden, den Versandaufpreis nach den Versandkosten (`shipping`) und vor der Gesamtsumme (`grand total`) erscheinen zu lassen. Dies gilt sowohl für die Darstellung im Frontend (`quote`), also beispielsweise im Warenkorb und in der Bestellhistorie, als auch für die über das Backend generierten Rechnungen (`order_invoice`) und Gutschriften (`order_creditmemo`). Sehen wir uns das erste Total-Model nun etwas genauer an:

```
<?php

class Webkochshop_Versandaufpreis_Model_Quote_Address_Total_Shipping_Surcharge
    extends Mage_Sales_Model_Quote_Address_Total_Abstract
{
    /**
     * Berechnen des gesamten Versandkostenaufpreises
     *
     * @param Mage_Sales_Model_Quote_Address $address
     * @return
     *    Webkochshop_Versandaufpreis_Model_Quote_Address_Total_Shipping_Surcharge
     */
```

```
public function collect(Mage_Sales_Model_Quote_Address $address)
{
    parent::collect($address);

    $total = 0;
    foreach ($address->getAllItems() as $item)
    {
        $total += $item->getShippingSurcharge() * $item->getQty();
    }

    /*
     * zum Grand Total addieren
     */
    $baseTotal = $this->_getBaseAmount($total);

    $this->_addAmount($total);
    $this->_addBaseAmount($baseTotal);

    /*
     * im Address-Model ablegen zur späteren Referenz in fetch()
     */
    $address->setShippingSurcharge($total);
    $address->setBaseShippingSurcharge($baseTotal);

    return $this;
}
```

Mithilfe der Methode `collect()` wird zunächst über alle Einträge von `$address` iteriert, um etwaige Aufschläge für jedes Produkt einer Bestellung berechnen zu können. Hierbei ist `$address` eine Instanz von `Mage_Sales_Model_Quote_Address`. Magento speichert die Totals in den Address-Models, was insofern praktisch ist, als dass sich die Totals ja zum Beispiel beim Checkout mit mehreren Adressen von Lieferanschrift zu Lieferanschrift unterscheiden können.

Über die Getter-Methode `getShippingSurcharge()` wird der entsprechende Attributwert ausgelesen, mit der bestellten Menge des Produkts multipliziert, und daraus wird die Gesamtsumme `$total` gebildet. Nun könnte man denken, dass es bei dieser Berechnung bleiben könnte – tatsächlich ist es allerdings so, dass es in Ihrem Onlineshop verschiedene Währungen geben kann. Der Versandaufpreis muss also noch die Währung umgerechnet werden, die in der jeweiligen Shopansicht als Standard festgelegt wurde. Um diese Umrechnung elegant lösen zu können, führen wir eine zweite Funktion in unsere Surcharge-Klasse ein:

```
/**
 * den Versandkostenaufpreis in die Basiswährung des Shops umrechnen
 *
 * @param float $amount
 * @return float $amount konvertiert in die Basis Währung
 */
public function _getBaseAmount($amount)
```

```
    {
        $currentCurrency = Mage::app()->getStore()->getCurrentCurrency();
        $baseCurrency = Mage::app()->getStore()->getBaseCurrency();

        if ($baseCurrency->getCode() == $currentCurrency->getCode())
        {
            $baseAmount = $amount;
        }
        else
        {
            $baseAmount = Mage::helper('directory')
                ->currencyConvert($amount, $currentCurrency, $baseCurrency);
        }

        return $baseAmount;
    }
```

Zuerst werden über zwei Getter-Methoden die Codes für die aktuell eingestellte und die in der Konfiguration als Standard definierte Währung ausgelesen und verglichen. Sind beide identisch, findet keine Umrechnung statt, gibt es einen Unterschied, wird über die Methode `currencyConvert()` der entsprechenden Helper-Klasse der Betrag in die aktuell anliegende Währung überführt.

Mithilfe der dritten und letzten Methode der Surcharge-Klasse wird dem Adressobjekt der Versandaufpreis zugewiesen, damit er im Frontent auch angezeigt werden kann. Die Methode `fetch()` ist in jedem Total-Model vorhanden. Wenn Sie möchten, dass ein Total-Betrag in der Auflistung angezeigt wird, muss ein Array mit den drei Einträgen dem Address-Model hinzugefügt werden. Ohne diesen Schritt würde der Betrag zwar berechnet, aber dem Kunden nicht angezeigt werden.

```
    /**
     * Zuweisen des Versandkostenaufpreises an das Addressobjekt zur Anzeige
     *
     * @param   Mage_Sales_Model_Quote_Address $address
     * @return
     *    Webkochshop_Versandaufpreis_Model_Quote_Address_Total_Shipping_Surcharge
     */
    public function fetch(Mage_Sales_Model_Quote_Address $address)
    {
        if ($address->getShippingSurcharge() > 0)
        {
            $address->addTotal(array(
                'code'  => $this->getCode(),
                'title' => Mage::helper('versandaufpreis')->__('Versandaufpreis'),
                'value' => $address->getShippingSurcharge()
            ));
        }
        return $this;
    }
}
```

Ist der Versandkostenaufpreis auf diese Weise berechnet und in die aktuelle Währung überführt worden, kann er von den beiden anderen Total-Models, die wir für unsere Extension benötigen, übernommen werden. Für das Total-Model der Rechnungserstellung im Backend wird dies über folgende übersichtliche Klasse realisiert:

```
<?php

class Webkochshop_Versandaufpreis_Model_Order_Invoice_Total_Shipping_Surcharge
    extends Mage_Sales_Model_Order_Invoice_Total_Abstract
{
    /**
     * Übertrag des Versandkostenaufpreises auf das Rechnungs-Model
     *
     * @param Mage_Sales_Model_Order_Invoice $address
     * @return
     *    Webkochshop_Versandaufpreis_Model_Order_Invoice_Total_Shipping_Surcharge
     */

    public function collect(Mage_Sales_Model_Order_Invoice $invoice)
    {
        $order = $invoice->getOrder();
        $surcharge = $order->getShippingSurcharge();
        $baseSurcharge = $order->getBaseShippingSurcharge();

        $invoice->setGrandTotal($invoice->getGrandTotal() + $surcharge);
        $invoice->setBaseGrandTotal(
            $invoice->getBaseGrandTotal() + $baseSurcharge
        );

        return $this;
    }
}
```

Sie fragen sich vielleicht an dieser Stelle, wie die bisher berechneten Aufpreisdaten an dieses Model übergeben werden. Dies erledigen die folgenden Fieldsets aus der *config.xml*:

```
<fieldsets>
    <sales_convert_quote_item>
        <shipping_surcharge>
            <to_order_item>*</to_order_item>
        </shipping_surcharge>
    </sales_convert_quote_item>
    <sales_convert_order_item>
        <shipping_surcharge>
            <to_quote_item>*</to_quote_item>
            <to_shipment_item>*</to_shipment_item>
        </shipping_surcharge>
    </sales_convert_order_item>
    <sales_convert_quote_address>
        <shipping_surcharge>
            <to_order>*</to_order>
        </shipping_surcharge>
```

```
            <base_shipping_surcharge>
                <to_order>*</to_order>
            </base_shipping_surcharge>
        </sales_convert_quote_address>
        <sales_convert_order>
            <shipping_surcharge>
                <to_invoice>*</to_invoice>
                <to_cm>*</to_cm>
            </shipping_surcharge>
        </sales_convert_order>
    </fieldsets>
```

Alle hier aufgelisteten Attribute werden entsprechend der Konfiguration bei einer Konvertierung (*Quote-Item* → *Order-Item*, *Order* → *Invoice* usw.) automatisch für das neue Model übernommen.

Nachdem Sie die eigentliche Berechnung und Zuweisung des Versandkostenaufpreises in drei neuen Total-Models kennengelernt haben, besteht der nächste Schritt darin, mithilfe der bereits häufiger erwähnten Event-Observer-Methode diese neuen Models überhaupt anzuwenden. Neue Attribute von Produkten werden nicht automatisch auf die Quote-Items übernommen, deswegen müssen wir das manuell mit dem Observer erledigen. Ohne den gingen die Informationen einfach verloren.

Wie immer ist die Konfigurationsdatei *config.xml* unser Ausgangspunkt, und die folgenden Zeilen verknüpfen Observer-Methoden mit den beiden relevanten Magento-Events:

```
<events>
    <sales_quote_product_add_after>
        <observers>
            <versandaufpreis>
                <type>singleton</type>
                <class>versandaufpreis/observer</class>
                <method>salesQuoteProductAddAfter</method>
            </versandaufpreis>
        </observers>
    </sales_quote_product_add_after>
    <sales_convert_quote_to_order>
        <observers>
            <versandaufpreis>
                <type>singleton</type>
                <class>versandaufpreis/observer</class>
                <method>salesConvertQuoteToOrder</method>
            </versandaufpreis>
        </observers>
    </sales_convert_quote_to_order>
</events>
```

In Rezept 5.4 »Kundenpreise anlegen«, auf Seite 123, haben Sie bereits kennengelernt, wie mit dieser Schreibweise definiert wird, welche Methoden welcher Observer-Klasse aufgerufen werden, wenn die jeweiligen Events ausgelöst werden. Das Event `sales_quote_product_add_after`, das auftritt, wenn ein Produkt dem Warenkorb hinzugefügt

worden ist, ruft die Methode `salesQuoteProductAddAfter()` des Observers in */Model/Observer.php* auf. Analog verhält es sich mit dem Event `sales_convert_quote_to_order`, das immer dann auftritt, wenn aus einer Quote eine Order wird. Schauen wir uns nun die Observer-Klasse genauer an:

```
<?php

class Webkochshop_Versandaufpreis_Model_Observer
{
    /**
     * Übertrag des Aufpreises vom Product-Model auf das Quote-Item-Model
     *
     * @param Varien_Event_Observer $observer
     */
    public function salesQuoteProductAddAfter($observer)
    {
        foreach ($observer->getEvent()->getItems() as $quoteItem)
        {
            $surcharge = $quoteItem->getProduct()->getShippingSurcharge();
            if ($surcharge > 0)
            {
                $quoteItem->setShippingSurcharge($surcharge);
            }
        }
    }

    /**
     * Übertrag des Versandkostenaufpreises vom Quote-Model auf das Order-Model
     *
     * @param Varien_Event_Observer $observer
     */
    public function salesConvertQuoteToOrder($observer)
    {
        $order = $observer->getEvent()->getOrder();
        $quote = $observer->getEvent()->getQuote();

        /*
         * Es ergibt zwar keinen Sinn, einen Versandkostenaufpreis für virtuelle
         * Produkte zu berechnen, doch dies ist ja auch nur ein Beispielrezept.
         */
        if ($quote->getIsVirtual())
        {
            $address = $quote->getBillingAddress();
        }
        else
        {
            $address = $quote->getShippingAddress();
        }

        $order->setShippingSurcharge($address->getShippingSurcharge());
        $order->setBaseShippingSurcharge($address->getBaseShippingSurcharge());
    }
}
```

Je nachdem, welches Event eintritt, wird über Setter-Methoden der zuvor berechnete Versandkostenaufschlag dem Quote-Item-Model bzw. dem Order-Model zugewiesen. Nach dem Zuweisen werden die neuen Attribute automatisch gespeichert, da wir ja die entsprechenden Attribute mit dem Setup-Skript angelegt haben. Die Attribute werden im Observer gesetzt, da es für das Übertragen von Werten aus dem Address-Model auf ein Order-Model keine automatische Zuweisung via Fieldsets gibt, wenn dynamisch entschieden werden muss, welches Adress-Model verwendet werden soll.

Nachdem Sie sich bisher vornehmlich darauf konzentriert haben, die zugrunde liegenden Werte zu berechnen und diese auf die relevanten Models zu übertragen, widmen Sie sich nun der Anzeige der berechneten Werte. Dies geschieht zum einen über einen neuen Block, der im Anschluss über Layoutdateien eingebunden wird. Den Block für den Versandkostenaufpreis erzeugen Sie auf die folgende Weise:

```
<?php

class Webkochshop_Versandaufpreis_Block_Sales_Total_Shipping_Surcharge
    extends Mage_Core_Block_Abstract
{
    /**
     * Hinzufügen des Versandkostenaufpreises zum Totals-Array
     *
     * @return Webkochshop_Versandaufpreis_Block_Sales_Total_Shipping_Surcharge
     */
    public function initTotals()
    {
        $parent = $this->getParentBlock();

        $source = $parent->getSource();

        $value = $parent->getSource()->getShippingSurcharge();

        if ($value > 0)
        {
            $total = new Varien_Object(array(
                'code'       => 'shipping_surcharge',
                'value'      => $parent->getSource()->getShippingSurcharge(),
                'base_value' => $parent->getSource()->getBaseShippingSurcharge(),
                'label'      => $this->__('Versandaufpreis'),
                'field'      => 'shipping_surcharge'
            ));
            $parent->addTotal($total, 'shipping');
        }
        return $this;
    }
}
```

In `Mage_Sales_Block_Order_Totals::_beforeToHtml()` wird für jeden via Layout-XML zugewiesenen Kindblock die Methode `initTotals()` automatisch aufgerufen, falls sie vorhanden ist. Diese Kindblöcke erzeugen selbst keine direkte Ausgabe, sondern fügen das

Ergebnis dem Totals-Array des Elternblocks (`Mage_Sales_Block_Order_Totals`) hinzu. Bei der Beschreibung der Layoutdateien weiter unten werden Sie sehen, wie ein Totals-Block als Kindblock zugewiesen wird. Diese auf den ersten Blick recht ungewöhnliche Methode hat den Vorteil, dass sich Totals komplett in Modulen kapseln lassen.

In der Konfigurationsdatei braucht es zum Schluss nur die Verknüpfung zweiter Layout-XML-Dateien, die die Formatierung des Front- bzw. Backends unserer *Versandaufpreis*-Extension übernehmen. Dies entspricht der Layoutupdate-Logik, die Sie bereits in Kapitel 4 kennengelernt haben:

```
<frontend>
    <!-- … -->
    <layout>
        <updates>
            <versandaufpreis>
                <file>versandaufpreis.xml</file>
            </versandaufpreis>
        </updates>
    </layout>
</frontend>

<adminhtml>
    <layout>
        <updates>
            <versandaufpreis>
                <file>versandaufpreis.xml</file>
            </versandaufpreis>
        </updates>
    </layout>
</adminhtml>
```

Das Einbinden des neuen Eintrags für den Versandkostenaufpreis im Frontend geschieht also zunächst über die Datei *versandaufpreis.xml*. Mit deren Besprechung endet zugleich auch unsere Arbeit im */app/code*-Ordner der Magento-Installation. Wie Sie bereits wissen, werden in Magento programm- und designrelevante Codeteile voneinander getrennt – die restliche Arbeit an unserer Extension erledigen wir demnach in */app/design/*. Dort hinterlegen wir im *frontend*-Bereich im *default*-Theme des *base*-Packages im Verzeichnis *layout* eine XML-Datei mit folgendem Inhalt:

```
<?xml version="1.0"?>
<layout version="0.1.0">

    <sales_order_view>
        <reference name="order_totals">
            <block type="versandaufpreis/sales_total_shipping_surcharge"
                 name="total_shipping_surcharge" as="shipping_surcharge"/>
        </reference>
    </sales_order_view>

    <sales_order_invoice>
        <reference name="invoice_totals">
```

```
            <block type="versandaufpreis/sales_total_shipping_surcharge"
                   name="total_shipping_surcharge" as="shipping_surcharge"/>
        </reference>
    </sales_order_invoice>

    <sales_order_creditmemo>
        <reference name="creditmemo_totals">
            <block type="versandaufpreis/sales_total_shipping_surcharge"
                   name="total_shipping_surcharge" as="shipping_surcharge"/>
        </reference>
    </sales_order_creditmemo>

    <... alle anderen relevanten Update-Handles ...>

</layout>
```

Anhand dieses Ausschnitts erkennen Sie, dass das Layout der für unsere Zwecke wichtigen Bereiche so modifiziert wird, dass der neue Versandaufpreis-Block zum Einsatz kommen kann. Dies ist mit `sales_order_view` beispielsweise die Detailansicht einer Bestellung im Kundenkonto.

Auf ähnliche Weise ergänzen Sie die Totals-Anzeige für Bestellungen im Admin-Panel via Layoutdatei */app/design/adminhtml/default/default/layout/versandaufpreis.xml*.

Siehe auch

Das gesamte Modul finden Sie im Download-Code zum Buch im Archiv *Webkochshop_Versandaufpreis-0.1.0.zip*.

6.4 Das Admin-Panel um eigene Konfigurationsmöglichkeiten erweitern

Problem

Sie möchten eine Extension so dynamisch programmieren, dass bestimmte Konfigurationswerte im Admin-Panel gepflegt werden können und diese Änderungen nicht im eigentlichen Programmcode durchgeführt werden müssen.

Lösung

Magento stellt zu diesem Zweck die Datei *system.xml* zur Verfügung, die im *etc/*-Verzeichnis einer Extension abgelegt wird und dafür sorgt, dass nach der Installation der neuen Extension neue Eingabefelder im Admin-Panel zur Verfügung stehen. Diese Datei ist zur gleichen Zeit die Grundlage für das Rezept 6.6, »Gratisartikel in den Warenkorb legen«, auf Seite 163, mit dem wir uns weiter unten in diesem Kapitel beschäftigen. Für dieses Rezept benötigen wir die folgenden Konfigurationswerte:

Gratisartikel-SKU
: Eindeutige Bestellnummer des Gratisartikels, der unter bestimmten Umständen in den Warenkorb gelegt und dem anschließend der Preis 0 zugewiesen wird.

Artikel-Mindestwert
: Ein Artikel muss einen gewissen Wert haben, um das Hinzufügen des Gratisartikels auch auszulösen.

Artikel-Mindestanzahl:
: Zusätzlich zum Mindestwert muss auch noch eine Mindestanzahl an Artikeln im Warenkorb vorhanden sein, um den Gratisartikel hinzuzufügen.

Zusatztext für Gratisprodukt
: Hier wird ein beliebiger Text eingetragen, der beim Gratisartikel angezeigt wird.

Diskussion

Erstellen Sie die Grundstruktur einer Extension und legen Sie im */app/code/local/Webkochshop/GratisArtikel/etc/*-Verzeichnis die Datei *system.xml* mit dem folgenden Inhalt an:

```
<?xml version="1.0"?>
<config>
    <tabs>
        <webkochshop translate="label" module="gratis">
            <label>Webkochshop</label>
            <sort_order>508</sort_order>
        </webkochshop>
    </tabs>
    <!--...-->
```

Mit diesem ersten `<tabs>`-Knoten der XML-Datei legen Sie fest, dass Magento einen neuen Tab in der linken Navigationsleiste des Admin-Panels anlegt. Es ist also nicht nötig, dies manuell zu erledigen und eventuell ein Template anzupassen – stattdessen macht Magento die Arbeit für Sie und präsentiert einen neuen Eintrag, wie in Abbildung 6-2 dargestellt.

Abbildung 6-2: Ein neuer Menüeintrag erscheint im Admin-Panel

Mithilfe der Angabe `<sort_order>` haben Sie Einfluss auf die Position dieses neuen Tabs. Standardmäßig sortiert Magento die Einträge von 0 bis n und abhängig davon, wie andere (Core-)Module konfiguriert sind (schauen Sie sich dazu doch einfach mal die *system.xml*-Dateien dieser Module an, um das Geheimnis ihrer Sortierung ein wenig zu lüf-

ten). Meistens braucht es nur etwas Experimentierfreude, den passenden Platz zwischen den anderen Modul-Tabs zu finden.

Als Nächstes veranlassen wir die Extension, um unter dem Tab einen neuen, klickbaren Menüeintrag anzulegen. Dies geschieht im `<sections>`-Knoten:

```
<sections>
    <gratis translate="label" module="gratis">
        <tab>webkochshop</tab>
        <label>Gratisartikel</label>
        <sort_order>10</sort_order>
        <show_in_default>1</show_in_default>
        <show_in_website>1</show_in_website>
        <show_in_store>1</show_in_store>
        <!--...-->
```

Der Extension weisen wir hiermit den internen Konfigurationspfad *gratis* und den menschenlesbaren Titel *Gratisartikel* zu, der auch im Admin-Panel angezeigt wird. Über den `<tab>`-Knoten weisen wir diese Section dem vorher definierten Eintrag in der Navigation zu. Ebenso finden Sie hier erneut einen Sortierungsparameter `<sort_order>`, mit dessen Hilfe Sie Ordnung in Ihre Menüeinträge bringen können. Die letzten drei Einträge legen fest, in welchen Geltungsbereichen – `default`, `website` und `store` – diese Konfigurationsmöglichkeiten gelten sollen.

Menueinträge schön und gut – aber wo verbergen sich denn nun die Konfigurationseinstellungen? Diese Frage wird zu einem guten Teil von den nachfolgenden Zeilen der *system.xml* beantwortet:

```
    <groups>
        <general translate="label" module="gratis">
            <label>Gratisartikel</label>
            <sort_order>10</sort_order>
            <show_in_default>1</show_in_default>
            <show_in_website>1</show_in_website>
            <show_in_store>1</show_in_store>
            <!--...-->
```

Mithilfe des `<groups>`-Knotens erzeugen Sie die Zwischenüberschriften im Inhaltsbereich des Admin-Panels, die sich per Mausklick so herrlich auf- und zuklappen lassen. Die von uns erzeugte Gruppe erhält den Namen *Gratisartikel* und kann – analog zu den `<sections>` – sowohl sortiert als auch den einzelnen Geltungsbereichen zugewiesen werden.

Nachdem Sie in der linken Navigationsleiste einen neuen Tab und einen neuen Menüpunkt hinzugefügt und im Inhaltsbereich eine neue Zwischenüberschrift mithilfe der *system.xml* angelegt haben, ist es nun an der Zeit, die eigentlichen Konfigurationsmöglichkeiten zu definieren. Dies geschieht im `<fields>`-Knoten:

```
        <!--...-->
        <fields>
            <free_product_sku translate="label">
                <label>Gratisartikel-SKU</label>
```

```
                <frontend_type>text</frontend_type>
                <sort_order>10</sort_order>
                <show_in_default>1</show_in_default>
                <show_in_website>1</show_in_website>
                <show_in_store>0</show_in_store>
            </free_product_sku>
            <min_item_value translate="label,comment">
                <label>Artikel-Mindestwert</label>
                <comment>Nur Artikel mit diesem Mindestwert werden gezählt
                </comment>
                <frontend_type>text</frontend_type>
                <sort_order>20</sort_order>
                <show_in_default>1</show_in_default>
                <show_in_website>1</show_in_website>
                <show_in_store>0</show_in_store>
            </min_item_value>
            <min_qty translate="label,comment">
                <label>Artikel-Mindestanzahl</label>
                <comment>Ab dieser Artikelanzahl wird der Gratisartikel dem
                Warenkorb hinzugefügt</comment>
                <frontend_type>text</frontend_type>
                <sort_order>30</sort_order>
                <show_in_default>1</show_in_default>
                <show_in_website>1</show_in_website>
                <show_in_store>0</show_in_store>
            </min_qty>
            <free_label translate="label">
                <label>Zusatztext für Gratisprodukt</label>
                <frontend_type>text</frontend_type>
                <sort_order>40</sort_order>
                <show_in_default>1</show_in_default>
                <show_in_website>1</show_in_website>
                <show_in_store>1</show_in_store>
            </free_label>
        </fields>
    </general>
</groups>
</gratis>
</sections>
</config>
```

Auf diese Weise lässt sich also eine beliebige Anzahl von Eingabefeldern erstellen, die an der vorher festgelegten Stelle im Admin-Panel angezeigt werden. Jedes dieser Felder hat analog zu den `<sections>` und den `<groups>` einen internen Code, mit dem Sie in der Programmierung auf die eingetragenen Werte zugreifen können, sowie eine Benennung (`label`). Über den Knoten `<frontend_type>` haben Sie Einfluss darauf, welche Art von Eingabe möglich sein wird. Wählen Sie hier `text` aus wie in unserem Beispiel, steht Ihnen ein Textfeld zur Verfügung, das sich mit einer beliebigen Zeichenfolge füllen lässt. Weitere Varianten sind `select` und `multiselect`, mit deren Hilfe Sie Drop-down- und Multiselect-Menüs realisieren können.

Last, but not least erlaubt Ihnen der <sort_order>-Parameter auch auf dieser Ebene, alle Eingabefelder beliebig zu sortieren. Mittlerweile sollten Ihnen die Parameter für die Wahl des Geltungsbereichs ebenfalls bekannt vorkommen. In diesem Zusammenhang sehen Sie eine Besonderheit im aktuellen Beispiel: Im Feld `min_qty` haben wir den Wert `0` in den Knoten <show_in_store> eingetragen. Damit erreichen wir, dass dieser Konfigurationswert nicht auf StoreView-Ebene geändert werden kann, sondern nur auf globaler und auf Website-Ebene. Haben Sie demnach Magento so aufgebaut, dass über einzelne StoreViews verschiedene Sprachen abgebildet werden, gilt für jeden Sprachbereich die gleiche Artikelanzahl für den Gratisartikel.

Wenn Sie die gezeigte *system.xml* in eine eigene Extension integrieren, wird im Admin-Bereich Folgendes dargestellt (siehe Abbildung 6-3).

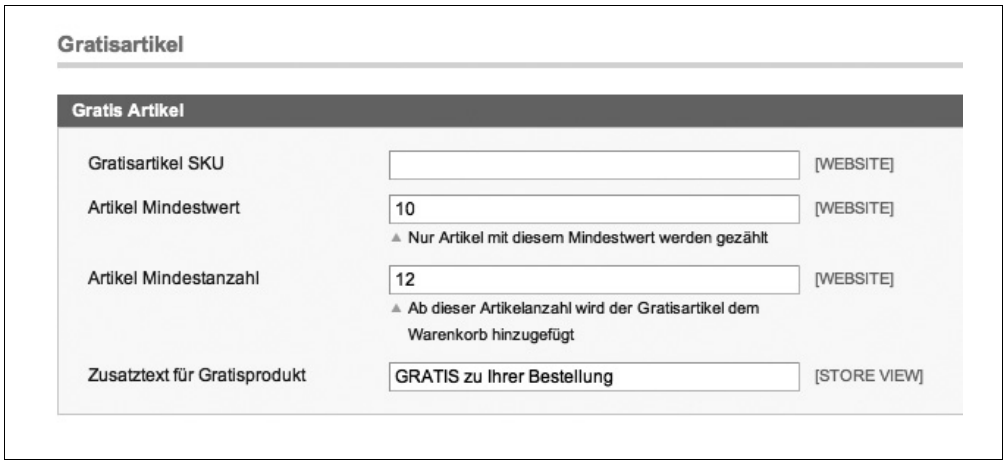

Abbildung 6-3: Das Admin-Panel wurde um einige Eingabefelder erweitert.

In diesem Rezept haben Sie erfahren, wie Sie Ihrer eigenen Extension auf einfache Weise verschiedene Konfigurationsmöglichkeiten spendieren können. Für diese Extension ist es dabei nicht von Bedeutung, an welcher Stelle und mit welcher Sortierung die neuen Eingabefelder angezeigt werden; die oben gezeigte Aufteilung in Tabs, Menüpunkte und Zwischenüberschriften dient lediglich der besseren Übersicht im Admin-Panel und soll denjenigen, die mit der Extension arbeiten, die Arbeit so leicht wie möglich machen. Bei der Verwendung der Konfigurationswerte in Ihrer Programmierung müssen Sie lediglich darauf achten, diese Hierarchiestufen beim Aufruf zu berücksichtigen. Auf die Mindestanzahl greifen Sie mit dem Aufruf `Mage::getStoreConfig('gratis/general/min_qty')` zu, in dem die <section>, die <group> und das <field> angegeben werden.

Um Standardwerte für die Eingabefelder zu vergeben, können Sie die XML-Struktur der *config.xml* wie folgt erweitern:

```
<config>
    <!-- ... -->
    <default>
        <gratis>
            <general>
                <free_product_sku></free_product_sku>
                <min_item_value>10</min_item_value>
                <min_qty>12</min_qty>
                <free_label>GRATIS zu Ihrer Bestellung</free_label>
            </general>
        </gratis>
    </default>
</config>
```

Wie Sie sehen können, spiegelt sich die Hierarchie aus der *system.xml*-Datei hier wider.

6.5 Nutzerrechte für neue Extensions anlegen

Problem

Werden für eine neue Extension erweiterte Konfigurationsmöglichkeiten mithilfe einer *system.xml* geschaffen, können Nutzer noch nicht darauf zugreifen, wenn sie innerhalb einer neuen Section angelegt werden. Fügen Sie neue Groups und Eingabefelder zu bestehenden Sections hinzu, hat der Admin sofort Zugriff darauf, und es müssen keine neuen Nutzerrechte vergeben werden.

Lösung

Wir ergänzen unser Modul um eine neue Datei *etc/adminhtml.xml* und fügen dort die sogenannten *Access Control Lists* (*ACL*) ein, sodass ein Admin-User auch die entsprechenden Rechte zur Konfiguration der Extension hat.

Diskussion

Das Hinzufügen von zusätzlichen Nutzerrechten für Ihre neue Extension vollzieht sich ausschließlich in der *adminhtml.xml*-Konfigurationsdatei, der zugehörige Code lautet wie folgt:

```
<?xml version="1.0" ?>
<config>
    <acl>
        <resources>
            <admin>
                <children>
                    <system>
                        <children>
                            <config>
                                <children>
```

```
                                <gratis translate="title" module="gratis">
                                    <title>Gratisartikel</title>
                                </gratis>
                            </children>
                        </config>
                    </children>
                </system>
            </children>
        </admin>
    </resources>
</acl>
</config>
```

Lassen Sie sich von dieser komplexeren Verschachtelung nicht aus der Ruhe bringen; diese resultiert letztlich aus der Magento-eigenen Konvention und muss mit übernommen werden. Sie sorgt dafür, dass bei der Einstellung der Admin-Gruppen im Admin-Panel ein neuer Eintrag erstellt wird, wie Sie in Abbildung 6-4 sehen können.

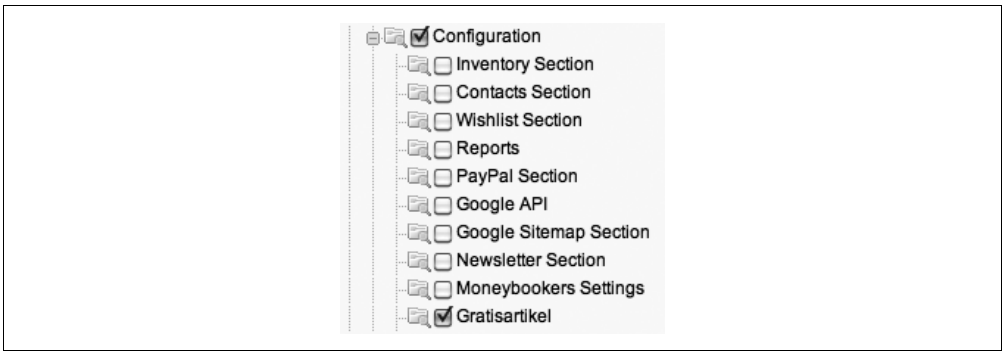

Abbildung 6-4: Im Admin-Panel kann nach Anpassung der ACL der Zugriff gesteuert werden.

Vor Magento 1.4 wurde diese Konfiguration übrigens in der *config.xml* unter dem Knoten `config/adminhtml` hinzugefügt. Das funktioniert immer noch, entspricht jedoch nicht mehr den aktuellen Konventionen.

Entscheidend ist der `<gratis>`-Knoten, mit dem genau die Section unserer Extension angesprochen wird, für die Sie in Rezept 6.6, »Gratisartikel in den Warenkorb legen«, auf Seite 163, die neuen Konfigurationswerte angelegt haben. Mithilfe des oben gezeigten XML-Konstrukts erweitern Sie Magentos ACL so, dass ein Admin-Benutzer Zugriff auf die erweiterten Konfigurationsmöglichkeiten erhält. Damit die neuen ACL auch erkannt werden, müssen Sie zuerst den Magento-Cache aktualisieren und sich dann aus dem Admin-Panel erst aus- und dann wieder dort einloggen.

6.6 Gratisartikel in den Warenkorb legen

Problem

Ein vorher festgelegter (Gratis-)Artikel soll automatisch in den Warenkorb gelegt werden, wenn ein Kunde einen anderen, regulären Artikel dem Warenkorb hinzufügt. Wird dieser Artikel wieder entfernt, verschwindet auch der Gratisartikel. Der Gratisartikel darf in dieser Beispiel-Extension nicht als reguläres Produkt mit der gleichen SKU in den Warenkorb gelegt werden können, da in diesem Fall beim Entfernen der benötigten Anzahl nicht nur der Gratisartikel aus dem Warenkorb entfernt wird, sondern auch das reguläre Produkt.

Lösung

Um die beschriebene Funktionalität zu realisieren, nutzen wir das Event `sales_quote_save_before`, das immer vor dem Speichern des Quote-Models ausgelöst wird. Wir verwenden die Event-Observer-Methode, um die notwendige Logik für die Gratisartikel zu implementieren.

Außerdem nutzen wir die erweiterten Konfigurationsmöglichkeiten, um aus dem Gratismodul auf notwendige Konfigurationswerte zugreifen zu können, und die neuen Nutzungsrechte, um im Admin-Panel unter *System* → *Konfiguration* darauf zugreifen zu können.

Diskussion

Die eigentliche Logik der *Gratikartikel*-Extension vollzieht sich ausschließlich in einer – überschaubaren – Observer-Klasse, die wir an dieser Stelle genauer beleuchten möchten:

```
<?php

class Webkochshop_GratisArtikel_Model_Observer
{
    /**
     * Prüfen, ob die Artikel im Warenkorb die Bedingungen für den Gratisartikel
     * erfüllen. Wenn ja, Aufruf der Methode zum Anlegen des Gratisartikels.
     *
     * @param Varien_Event_Observer $observer
     */
    public function salesQuoteSaveBefore($observer)
    {
        /**
         * @var $quote Mage_Sales_Model_Quote
         */
        $quote = $observer->getEvent()->getQuote();

        /*
         * Prüfen, ob die Bedingungen für den Gratisartikel erfüllt sind
         */
```

```
        $minItemValue = Mage::getStoreConfig('gratis/general/min_item_value');
        $count = 0;
        foreach ($quote->getAllItems() as $item)
        {
            if ($item->isDeleted()) continue;

            if ($item->getCalculationPrice() >= $minItemValue)
            {
                $count += $item->getQty();
            }
        }
        if ($count >= Mage::getStoreConfig('gratis/general/min_qty'))
        {
            $freeProductQty = 1;
        }
        else
        {
            $freeProductQty = 0;
        }

        $this->_setFreeProductOnQuote($quote, $freeProductQty);
    }
```

Diese Methode wird unmittelbar dann aufgerufen, wenn das besagte Event in Magento ausgelöst wird. Vereinfacht gesagt, wird über das Objekt `Mage_Sales_Model_Quote` iteriert, um auf Basis der Einträge des Warenkorbs zu entscheiden, ob und wie unsere Gratisartikel-Logik angewendet wird. Sollte ein Produkt den gleichen oder einen höheren Wert haben als der, der in der Konfigurationsdatei angegeben wurde, wird ein ebenfalls vorher definierter Gratisartikel hinzugefügt, anderenfalls nicht. Besonders interessant ist in diesem Zusammenhang die folgende Zeile:

```
$minItemValue = Mage::getStoreConfig('gratis/general/min_item_value');
```

Mit dieser Syntax haben Sie die Möglichkeit, auf definierte Konfigurationswerte zuzugreifen; der String `gratis/general/min_item_value` im Funktionsaufruf gibt dabei an, auf welcher Hierarchiestufe der Konfigurationswert abgelegt wurde.

In den anderen Methoden der Observer-Klasse werden weitere Fallunterscheidungen getroffen, und es wird überprüft, ob es bereits automatisch hinzugefügte Gratisartikel im Warenkorb gibt.

```
    protected function _setFreeProductOnQuote(Mage_Sales_Model_Quote $quote,
                      $qty = 1)
    {
        $sku = Mage::getStoreConfig('gratis/general/free_product_sku');

        ...

        /*
         * Fallbehandlung: Gratisartikel ist noch nicht im Warenkorb vorhanden.
         */
        if ($qty > 0)
```

```
    {
        /*
         * Produkt ist noch nicht im Warenkorb – Hinzufügen des Artikels.
         */
        $product = Mage::getModel('catalog/product')
            ->loadByAttribute('sku', $sku);
        if ($product)
        {
            if ($product->getId())
            {

                /*
                 * Hinzufügen des Artikels zum Warenkorb
                 */
                $quote->addProduct($product, $qty);
                $quoteItem = $quote->getItemByProduct($product);

                /*
                 * Erzwingen eines Preises, auch wenn das Produkt einen
                 * anderen Preis hat
                 */
                $quoteItem->setCustomPrice(0);

                /*
                 * Dem Kunden sollen keine zusätzlichen Versandkosten
                 * durch den Gratisartikel entstehen.
                 */
                $quoteItem->setFreeShipping($qty);

                /*
                 * Setzen einer Custom-Option zur Anzeige eines Zusatztexts
                 */
                $labelText = Mage::getStoreConfig('gratis/general/free_label');
                $options = array(array(
                    'label'       => $labelText,
                    'value'       => '',
                    'print_value' => '',
                ));

                $quoteItem->addOption(array(
                    'code'  => 'additional_options',
                    'value' => serialize($options),
                ));

                ...

            }
        }
    }
}
```

An dem oben gezeigten Ausschnitt aus der Methode `_setFreeProductOnQuote()` können Sie erkennen, wie auf die Systemkonfiguration mittels `Mage::getStoreConfig('gratis/general/free_product_sku')` zugegriffen wird. Außerdem sehen Sie, wie ein Produkt durch den PHP-Code `$quote->addProduct($product, $qty)` zum Warenkorb hinzugefügt wird.

Um einem Artikel im Warenkorb einen Preis zuzuweisen, der sich von dem des Produkts unterscheidet, wird die Methode `$quoteItem->setCustomPrice($neuerPreis);` genutzt. Als kleines Extra wird ein Optionswert dynamisch einem Quote-Item zugewiesen, sodass er als Zusatztext im Warenkorb und später in der Bestellung zu sehen ist. Dazu muss wie im obigen Beispiel ein Array gebaut werden, und dieses muss dann mit `$quoteItem->addOption()` hinzugefügt werden.

Siehe auch

Den vollständigen und ausführlich kommentierten Code finden Sie im Download-Code zum Buch im Archiv *Webkochshop_GratisArtikel-0.1.0.zip*.

6.7 Ein Bestellkommentarfeld einfügen

Problem

Sie möchten Ihren Kunden die Möglichkeit einräumen, während des Bestellvorgangs im Onepage-Checkout einen Bestellkommentar einzutragen.

Lösung

Fügen Sie den Order- und Quote-Models jeweils ein neues Attribut hinzu, in dem dieser Kommentar aufgenommen wird, und füllen Sie dieses mit der Event-Observer-Methode. Diese Extension haben wir für Sie unter dem Namen *Webkochshop_Orderkommentar* vorbereitet.

Diskussion

Als Erstes werden zwei neue Attribute via Installationsskript den Order- und Quote-Models hinzugefügt:

```
<?php

/**
 * @var $installer Mage_Sales_Model_Mysql4_Setup
 */
$installer = $this;
$installer->startSetup();

$installer->addAttribute('quote', 'order_kommentar', array(
    'type'     => 'text',
```

```
    'label'    => 'Bestellkommentar',
    'required' => 0,
));

$installer->addAttribute('order', 'order_kommentar', array(
    'type'     => 'text',
    'label'    => 'Bestellkommentar',
    'required' => 0,
));

$installer->endSetup();
```

Anschließend definieren wir in der *config.xml*, welche Events dazu genutzt werden sollen, das Bestellkommentarfeld zu füllen:

```
<frontend>
    <!-- … -->
    <events>
        <sales_quote_save_before>
            <observers>
                <orderkommentar>
                    <type>singleton</type>
                    <class>orderkommentar/observer</class>
                    <method>salesQuoteSaveBefore</method>
                </orderkommentar>
            </observers>
        </sales_quote_save_before>
        <sales_model_service_quote_submit_before>
            <observers>
                <orderkommentar>
                    <type>singleton</type>
                    <class>orderkommentar/observer</class>
                    <method>salesModelServiceQuoteSubmitBefore</method>
                </orderkommentar>
            </observers>
        </sales_model_service_quote_submit_before>
    </events>
</frontend>
```

Der Observer mitsamt den beiden angemeldeten Methoden sehen Sie in folgendem Codebeispiel:

```
<?php

class Webkochshop_OrderKommentar_Model_Observer
{
    /**
     * Setzen des Kommentars auf dem Quote-Model
     *
     * @param Varien_Event_Observer $observer
     */
    public function salesQuoteSaveBefore($observer)
    {
        if ($this->_checkControllerAction('checkout', 'onepage', 'saveBilling'))
```

```
        {
            $kommentar = Mage::app()->getRequest()->getParam('kommentar');
            if ($kommentar)
            {
                $observer->getEvent()->getQuote()->setOrderKommentar($kommentar);
            }
        }
    }

    /**
     * Eintragen des Kommentars in die Order-History
     *
     * @param Varien_Event_Observer $observer
     */
    public function salesModelServiceQuoteSubmitBefore($observer)
    {
        /*
         * Der Kommentar wurde bereits durch den Eintrag in das Fieldset
         * config/fieldsets/sales_convert_quote auf dem Order-Model
         * erreicht. Hier fügen wir den Kommentar der Order-Historie hinzu.
         */
        $order = $observer->getEvent()->getOrder();
        $order->addStatusHistoryComment(
            Mage::helper('orderkommentar')->__(
                "<b>Bestellkommentar</b>:<br/>\n%s",
                $kommentar
            )
        );
    }
}
```

Die datenmäßige Grundlage für die Bestellkommentare ist mit diesen einfachen Schritten also geschaffen; was jetzt noch fehlt, ist die tatsächliche Anzeige eines neuen Eingabefelds im Onepage-Checkout, in das Ihre Kunden den entsprechenden Kommentar eintragen können. Die dazu nötigen Frontend-Umbauarbeiten beginnen mit dem Einbinden einer eigenen Layoutdatei in die *config.xml* unserer Extension:

```
    <frontend>
        <layout>
            <updates>
                <orderkommentar>
                    <file>orderkommentar.xml</file>
                </orderkommentar>
            </updates>
        </layout>
        <!--...-->
```

Die zugehörige Layoutdatei liest sich wie folgt:

```
<?xml version="1.0"?>
<layout version="0.1.0">
    <checkout_onepage_index>
        <reference name="head">
```

```
            <action method="addItem">
                <type>skin_css</type>
                <name>webkochshop/orderkommentar/css/kommentar.css</name>
            </action>
        </reference>
        <reference name="checkout.onepage.billing">
            <action method="setTemplate">
                <template>webkochshop/orderkommentar/checkout/onepage/billing.phtml
                </template>
            </action>
            <block type="core/template" name="orderkommentar.field"
                   template="webkochshop/orderkommentar/field.phtml"/>
        </reference>
    </checkout_onepage_index>
</layout>
```

Wie Sie sehen, wird hier mithilfe dieses XML-Schnipsels dem Inhaltsblock `checkout.onepage.billing` ein neues Template namens *billing.phtml* zugewiesen. Darüber hinaus erzeugt diese Layoutdatei einen neuen Block namens *orderkommentar.field*, der durch ein eigenes Template *field.phtml* formatiert wird. In diesem Template findet sich unter anderem auch der HTML-Code für das neue Eingabefeld:

```
<div id="orderkommentar">
    <label for="kommentar">
        <?php echo $this->__('Ihr Kommentar zur Bestellung:') ?>
    </label><br/>
    <textarea name="kommentar" id="kommentar" class="kommentar">
        <?php echo $this->htmlEscape($this->getParentBlock()->getQuote()->
getOrderKommentar()) ?>
    </textarea>
</div>
```

Voilà! Wenn Sie diese Schritte nachvollzogen haben, anschließend ein Produkt in den Warenkorb legen und den Bestellprozess beginnen, wird ein neues Eingabefeld wie in Abbildung 6-5 angezeigt.

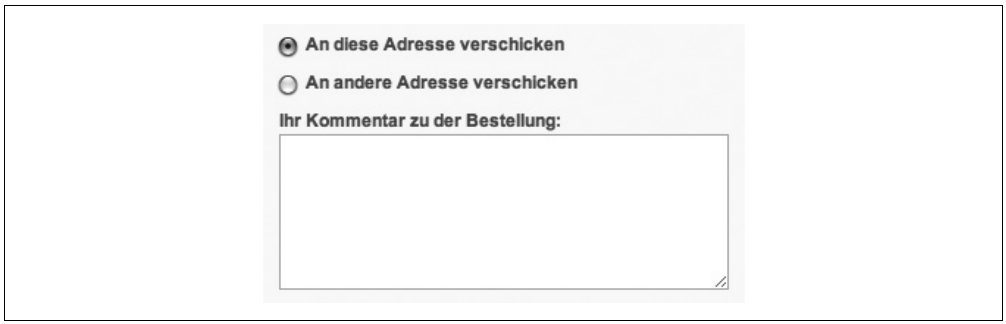

Abbildung 6-5: Ab sofort können Kunden ihrer Bestellung einen Kommentar hinzufügen

Zum vollständigen Shopbetreiberglück fehlt nun nur noch, dass die von den Kunden eingegebenen Kommentare auch entsprechend im Admin-Panel dargestellt werden. Die

Vorgehensweise ist hier die gleiche wie bei den Layoutupdates im Frontend: Es wird eine neue Layoutdatei für das Admin-Panel in der *config.xml* eingebunden, die für die entsprechenden Ausgaben verantwortlich ist.

Siehe auch

Den vollständigen Code zu diesem Rezept finden Sie im Archiv *Webkochshop_Order-Kommentar-0.1.0.zip* im Download-Code.

In diesem Kapitel haben wir uns mit der Frage auseinandergesetzt, wie Sie die Angebote und Bestellungen im Magento-Universum so anpassen können, dass sie für Ihre eigenen Zwecke nutzbar werden. Das nächste Kapitel steht ganz im Zeichen der Systemintegration.