Designheft Online-Bilderservice

22. Januar 2004

Gruppe 3
Daniel Jagszent
Tilmann Kuhn
Alexander Spengler

Version
v1.0

Status
Endgültig

Inhaltsverzeichnis

1 Einführung

Mit diesem Dokument wird der Entwurf der Webanwendung „Online Bilderservice“ festgelegt. Auftraggeber ist die Firma B&S, Auftragnehmer die Gruppe 3 des SWT-Praktikums 2003 der Universität Karlsruhe (TH). Zum Verständnis dieses technischen Dokuments benötigt man fundierte Kenntnisse in der Modellierungssprache UML[1]. Spezieller in folgenden Diagramm-Arten:

- Paketdiagramm
- Klassendiagramm
- Ablaufdiagramm
- Sequenzdiagramm
- Zustandsdiagramm
- Verteilungsdiagramm

Ausgangspunkt ist das gemeinsam erarbeitete Pflichtenheft für den Online-Bilderservice in der Version v0.485, die B&S am 5. Dezember 2003 erhalten hat. Wir werden uns im Folgenden an manchen Stellen direkt auf das Pflichtenheft und die dort eingeführte Nummerierung beziehen, so bezeichnet z.B. /U10/ den Geschäftsprozess „Schnupperbestellung“ oder /F10/ die entsprechende Produktfunktion.

Hinweis

In den Diagrammen dieses Dokuments werden wir die einzelnen Teile (Klassen, Objekte, Pakete etc.) durch nebenstehende Farbkodierung zu den Komponenten Modell, Sicht, Steuerung und der Datenbank zuordnen.

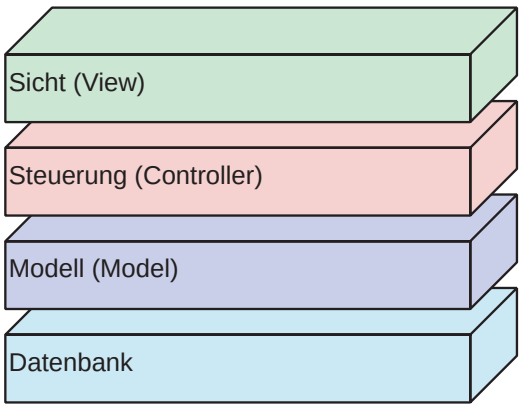

Abbildung 1.1 Farbkodierung in diesem Dokument

1 Unified Modeling Language. Ein Standard der Object Management Group (OMG, http://www.omg.org/) zum graphischen, objektorientierten Modellieren von Geschäftsprozessen und Entwurfsentscheidungen.
Siehe http://www.omg.org/uml/

1.1 Das Model-View-Controller Entwurfsmuster

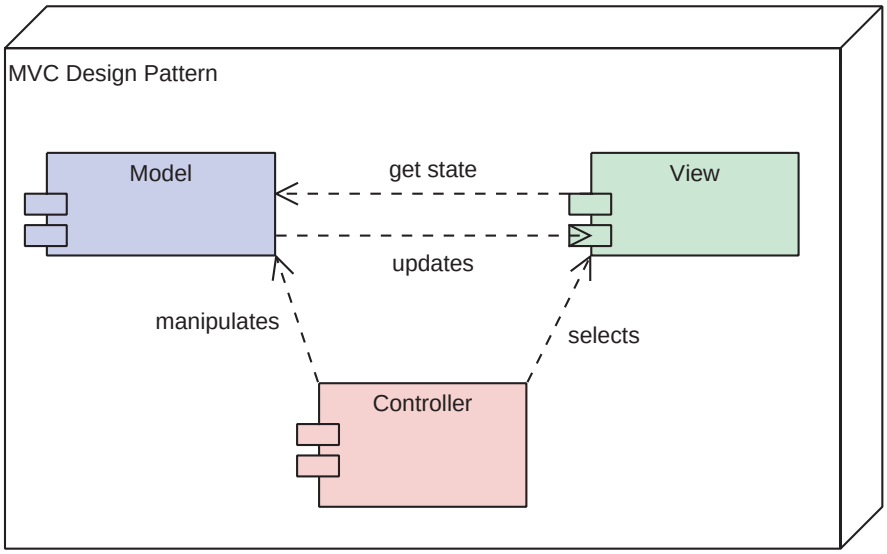

Abbildung 1.2 Grundsätzliches Prinzip des MVC Entwurfsmusters

MVC ist weniger ein Entwurfsmuster an sich, als ein Paradigma bei der Entwicklung von Benutzerschnittstellen. Es zielt darauf ab, die Darstellung eines Geschäftsmodells von der Anzeige (Sicht, View), in der es dargestellt wird, zu entkoppeln, so dass Modell und Sicht unabhängig voneinander ausgetauscht werden können. Dazu wird eine Steuerung (Controller) verwendet, die auch dafür sorgt, dass die Aktionen, die der Benutzer an der Anzeige ausführt, an die Geschäftslogik weitergegeben werden. Realisiert wird das Ganze häufig durch die Kombination geeigneter Entwurfsmuster wie Observer, Command, Adapter, Decorator, Chain of Responsibility.

Das MVC Paradigma bedarf bei unterschiedlichen Anwendungstypen unterschiedliche Ausprägungen und Spezialisierungen. Bei der von uns verwendeten Technologie ist eine solche Ausprägung das „Model 2“, das dem Struts Framework zugrundeliegt.

1.2 Das Struts Framework[2]

Das Struts-Framework ist relativ klein, es beschränkt sich auf ein Minimum an Funktionalität, um die Entwicklung von Webapplikationen nach dem Model 2 Ansatz, einer Variation des Model View Controller -Paradigmas, möglichst einfach zu machen. Seine Leichtgewichtigkeit zeichnet sich auch dadurch ab, dass es sehr einfach einzusetzen ist, z.B. keine Änderungen am Webcontainer erfordert, und mit quasi allen gängigen Java-Web-Technologien zusammen eingesetzt werden kann.

2 Teile dieses Abschnittes wurden der Seminararbeit „Entwicklung von Webapplikationen mit den Struts und Expresso Frameworks – ein Vergleich“ von Boudigué Alioum und Tilmann Kuhn entnommen.
Siehe http://www.ipd.uka.de/~oosem/WAWS03/ausarbeitung/BoudigueAlioum-TilmannKuhn-Ausarbeitung.pdf

1.2.1 Struts und der Modell 2 Ansatz

Modell 2 ist eine Übertragung des MVC-Modells auf Webanwendungen, die auf J2EE-Technologien basieren. Der Model 2 Ansatz bietet vor allem für größere Webanwendungen eine sauberere Umsetzung des MVC-Paradigmas als das Model 1. Model 1 geht davon aus, dass eine Anfrage seitens des Clients direkt an die benötigte JSP-Seite gerichtet ist, die dann wiederum für das Ausführen der gewünschten Aktionen und die anschließende Darstellung der Ergebnisse zuständig ist. Im Model 2 dagegen gehen alle Anfragen, die Aktionen bewirken sollen, an ein Controller-Servlet, das dann die Geschäftslogik bereitstellt oder gar aktiviert und die zuständigen View-Komponenten selektiert. Der View-Teil, meist eine JSP-Seite, bekommt seine darzustellenden Informationen über eine JavaBean zur Verfügung gestellt. Abbildung 1.3 veranschaulicht das Modell und die Ablaufreihenfolge.

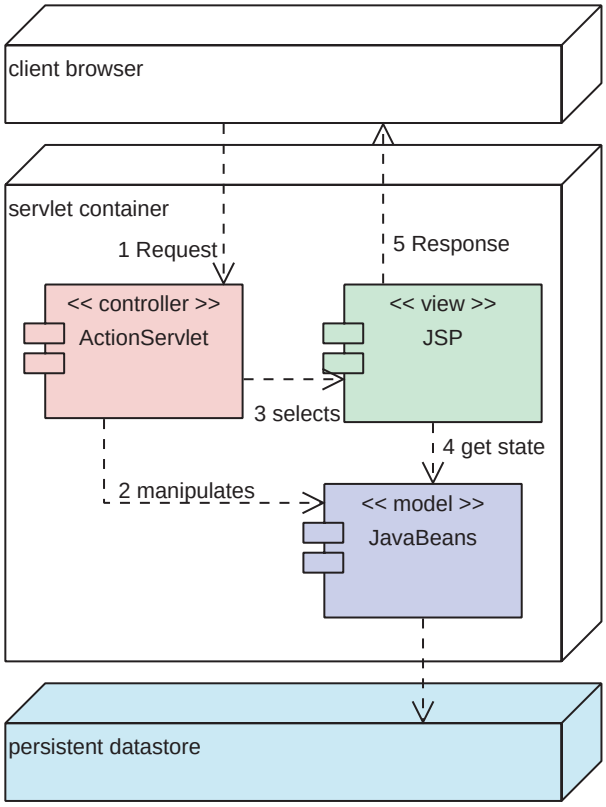

Abbildung 1.3 Struts Modell und Ablaufreihenfolge

Die Struts-Architektur implementiert im Wesentlichen den Controller-Teil dieses Modells bis auf die Teile, die von Anwendung zu Anwendung unterschiedlich sein müssen, um die eigentlichen Aktionen auszuführen. Struts bietet auch reichlich Unterstützung für die Implementierung der View- und Modellkomponenten. Zentraler Zugangspunkt zur Webapplikation bietet bei Struts ein Controller-Servlet das `ActionServlet` heißt.

1.2.2 Struts Controller-Komponenten

Das ActionServlet wird über Browserrequests angesprochen, die einem bestimmten Schema genügen. Dafür wird in der Webapplikation etwa eine bestimmte URL-Klasse auf das ActionServlet gemappt, sodass z.B. alle URLs, die auf „.do“ enden oder deren Lokalteil mit „/do/“ beginnt, durch das ActionServlet bearbeitet werden. Erhält das Servlet so einen Request, so betrachtet es den Rest der URL und sucht in seiner Konfigurationsliste ein `ActionMapping` das zu dieser URL gehört. ActionMappings sind Tupel aus einem URL-Lokalteil, einer zugehörigen `Action` und noch weiteren Informationen, auf die z.T. später eingegangen wird. Actions sind vom Entwickler zu implementierende Objekte, die Nachrichten an die Geschäftslogik schicken, Ergebnisse berechnen und endgültig entscheiden, an welche View-Komponente die Ausgabe delegiert wird. Nach der Erzeugung einer eventuell zur Action gehörenden FormBean wird die Kontrolle zunächst an die Action übergeben und anschließend an die von der Action bestimmte View-Komponente, die dann für die Bedienung der Anfrage sorgt. Das Struts ActionServlet ist somit eine geeignete Implementierung des „Service to Worker“ Entwurfmusters[3].

ActionMappings und ActionForwards In einem ActionMapping wird festgelegt, wie URLs auf Actions abgebildet werden. Zur Veranschaulichung betrachten wir die XML-Definition eines ActionMapping in der Strutskonfiguratinsdatei wie sie in einer Beispielanwendung stehen könnte:

```
<action path="/addUrlDocument"
type="de.tkuhn.searchwebapp.action.AddUrlDocumentAction"
name="addUrlDocumentForm" scope="request" validate="true" input="/index.jsp">
<forward name="fail" path="/fail.jsp" />
</action>
```

Das Path-Attribut gibt an, welche URLs gemappt werden. In unserem Fall wäre das in etwa eine URL wie: `http://localhost:8080/search-webapp/addUrlDocument.do`. Mit Type wird bestimmt, welches die zu diesem Mapping gehörende Action-Klasse ist, die später die Nachfrage behandelt. Mit Name, Scope und Validate wird festgelegt, ob Struts für dieses Mapping eine ActionFormBean verwenden soll, die die Benutzereingaben aufnimmt, in welchem Kontext diese abgelegt werden soll und ob die Eingaben auf Richtigkeit überprüft werden sollen. Unter Input ist eine JSP-Seite angegeben, von der aus auf diese URL verwiesen wird. Wird bei der Überprüfung der Benutzereingaben festgestellt, dass diese nicht in Ordnung sind, so wird sofort diese Seite bemüht, um die aufgetretenen Fehler anzuzeigen und den Benutzer zu einer Korrektur aufzufordern, ohne dass die Action selbst überhaupt aufgerufen werden muss.

Zum Schluss des Mappings wird noch ein sogenanntes `ActionForward` definiert. Diese Forwards sind ein wichtiger Bestandteil des Struts-Frameworks, um Actions unabhängig von den tatsächlichen View-Elementen zu halten. Ein ActionForward verknüpft einen logischen Namen in diesem Beispiel „fail“ mit einer View-Instanz. Das Forward

3 Eine Kombination aus einem Controller einem View-Dispatcher und weiteren Helferobjekten, bei der der Controller nach einer Anfrage zunächst veranlasst, dass die entsprechende Geschäftslogik aufgerufen wird und die Ergebnisdaten zur Verfügung stehen. Danach wird über den Dispatcher die zuständige View-Komponente ermittelt, die dann mit der Darstellung beauftragt wird.

hat, so definiert, nur in Verbindung mit diesem ActionMapping Gültigkeit. Schlägt man in der Action `AddUrlDocumentAction` also nach einem Forward namens „fail“ nach, erhält man als Ergebnis „/fail.jsp“, falls die Action über dieses Mapping aktiviert wurde. Damit wird deutlich, dass man durchaus über verschiedene URLs die gleiche Action aktivieren kann, aber mit verschiedenen Forwards dann unterschiedliche Darstellungen der Ergebnisdaten erhält. Eine Anwendung dafür wäre z.B., Informationen aus einer Anwendung einmal für einen Browser und einmal für ein Wap-Gerät darzustellen. Über die mappinggebundene Definition hinaus lassen sich auch noch globale ActionForwards definieren, die in allen Actions immer sichtbar sind, wenn sie nicht von einem lokalen Forward überlagert werden.

Actions Eine Action ist der Teil von Struts, durch den letztendlich die Arbeit am Geschäftsmodell der Webanwendung veranlasst wird. Wie der Name der Klasse schon sagt, handelt es sich bei Actions also um aktive Vorgänge, sodass sie ideal geeignet sind, um die Usecases der Applikation abzubilden. Struts Actions entsprechen nur teilweise dem „Command“ Entwurfsmuster, da sie weder in Warteschlangen organisiert werden können, noch die Fähigkeiten besitzen, direkt rückgängig gemacht werden zu können. Actions werden vom Programmierer dadurch erstellt, dass er von `org.apache.struts.action.Action` eine Klasse ableitet und die Methode

```
ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
request, HttpServletResponse response)
```

implementiert. Neben dem Mapping, das für die Weiterleitung verantwortlich war, und den typischen HTTP-Parametern erhält man auch noch die `ActionForm`, die die Benutzereingabedaten enthält. Innerhalb der execute-Methode kann man dann die notwendigen Geschäftslogikaufgaben direkt berechnen oder alternativ, was auch von den Struts-Entwicklern empfohlen wird, nochmals getrennte Geschäftslogikbeans dazu bemühen. Dies hat mehrere Vorteile: Zum Einen kann man dann auf einfache Weise auf die gleiche Geschäftslogik aus unterschiedlichen Actions zugreifen, zum Anderen hat man dadurch die Möglichkeit die Geschäftslogik mittels des Entwurfmusters „Facade“ zu kapseln und damit ohne großen Aufwand austauschbar zu machen. Als Rückgabe erwartet das ActionServlet ein ActionForward, welches man in der Regel vom ActionMapping bekommt, über die Methode `ActionForward findForward(String name)`, der man den in der Konfiguration festgelegten logischen Namen gibt. In seltenen Fällen, wird die Anfrage auch direkt von der Action direkt bedient, so dass man hier auch `null` zurückgeben kann. Zur Kommunikation mit JSPs gibt es von einer Action aus im Wesentlichen zwei Möglichkeiten: Man kann Daten in Form von Javaobjekten, die man der Sicht zur Darstellung geben möchte, unter einem Schlüssel in den Kontext des Requests einhängen, wo die JSP-Seite sie dann später abrufen kann. Dies würde man mit dem Aufruf von `request.setAttribute("BeispielSchlüssel", datenObjekt)` erreichen. Zur Datenübergabe gibt es noch zwei weitere Kontexte, die in 1.2.4 kurz beschrieben sind. Darüber hinaus bietet Struts noch eine Funktion an, um so genannte `ActionMessages` oder auch für noch schlimmere Fälle, die davon abgeleiteten `ActionErrors`, an eine JSP-Seite zu übergeben. Dabei handelt es sich um internationalisierte Textnachrichten, die über spezielle Tags in der JSP sichtbar gemacht werden können.

ActionForms sind JavaBeans, die mit ihren verschiedenen Properties die Benutzereingaben in die Webapplikation aufnehmen. Um dies zu erreichen, erzeugt das ActionServlet die ActionForm im entsprechenden Kontext, falls noch nicht vorhanden, und füllt ihre Properties mit den Werten der gleichnamigen Requestparameter der HTTP-Anfrage. Darüber hinaus besitzen ActionForms noch Methoden zum Zurücksetzen und zum Validieren der Werte. Bei letzterem überprüft die FormBean, ob die Inhalte ihrer Properties den Erwartungen entspricht, und liefert, falls nicht, als Ergebnis der Operation eine Reihe ActionErrors zurück. Diese sagen dem ActionServlet wiederum, dass etwas nicht in Ordnung war, so dass es dem Benutzer die Eingabeseite erneut anbieten kann. Dabei ist es über eine Struts Tag-Bibliothek sehr einfach möglich, in einer JSP-Seite die FormBean mit einem HTML-Formular zu verknüpfen und die alten Eingabewerte und die Fehlermeldungen in die neue Eingabeaufforderung zu übernehmen. ActionForms sind also eine Art erweitertes „Value Object“, das zum Datenaustausch zwischen den View- und Controller-Komponenten eingesetzt wird.

1.2.3 Sichtkomponenten

Am häufigsten mit Struts verwendete View-Komponenten sind Java Server Pages, wenn auch es mit Struts möglich ist, auch XML eventuell zusammen mit XSL zu verwenden. Selbstverständlich können auch beide kombiniert werden. Struts bietet verschiedene Hilfen bei der Erstellung von JSP-Seiten.

Internationalisierung (i18n) ist ein wichtiger Aspekt in den heutigen Applikationen, vor allem in einem universell verfügbaren Medium wie dem Internet. Struts trägt diesem Trend Rechnung, indem es die gängigen Java-i18n-Methoden von Java nutzt und diese für Webanwendungen verfügbar macht. Es bietet die Möglichkeiten, über `ResourceBundles` verfügbare internationalisierte Zeichenketten auszugeben und sogar Platzhalter in der Zeichenkette durch aktuelle Parameter zu ersetzen. Dies geschieht aus den Actions über ein `MessageResources` Objekt. Aus den FormBeans über die `ActionErrors` und für die JSP-Seiten stellt Struts ein eigenes „message“-Tag zur Verfügung. Die verwendeten `ResourceBundles` lassen sich über die Strutskonfiguration genau einstellen.

HTML-Form-Interaktion. Die Eingabe von Informationen durch den Benutzer über HTML-Formulare wird mit Struts zum Kinderspiel, verwendet man die Struts-HTML-Tag-Bibliothek. In ihr findet man spezielle Tags für HTML-Formulare und für alle gängigen HTML-Input-Felder, die man, einfach versehen mit den Property-Namen einer `ActionForm`, in die JSP-Seite einfügt. Jetzt muss man nur noch im Submit-Tag angeben, welches `ActionMapping` verwendet werden soll und schon hat man in der ausgeführten Action alle Eingabewerte in der entsprechenden FormBean.

Zusammengesetzte Sichten sind Sichten, die über Schablonen aus verschiedenen anderen Sichten gemäß dem GoF-Pattern „Composite“ zusammengesetzt sind. Hierfür bieten sich im Struts-Framework gleich zwei Bibliotheken an, eine komplexere und eine einfachere.

Die Template Taglib ist recht einfach gehalten und erlaubt das einfache Verschachteln von Templates und HTML-Fragmenten. Sie ist aber zu Gunsten der anderen Bibliothek in Struts 1.1 bereits deprecated.

Die Tiles Bibliothek ist ein relativ komplexes, aber dennoch leicht einzusetzendes Framework zur Erzeugung bis ins Detail konfigurierbarer Sichten. Ein Sichtteil, der hier Tile heißt, kann auf unterschiedliche Art und Weise definiert werden. Tiles können auch voneinander erben und ergänzen. Man kann auch erreichen, dass die Webapplikation nur die Grobstruktur und die zur Verfügung stehenden Tiles festlegt und der Benutzer selbst festlegen kann, was er genau sehen will. Die Auswahl dargestellter Tiles kann man auch vom jeweiligen HTTP-Request abhängig machen, wenn man auf Benutzersessions, Benutzerrechte, Benutzer-Locale oder den Browsertyp eingehen will. Insgesamt bietet Tiles also genug Flexibilität, um auch anspruchsvolle Webanwendungen zu realisieren. Tiles ist außerdem voll kompatibel zur Template Taglib.

Tagbibliotheken, die in Struts enthalten sind, bieten Lösungen für viele JSP-Alltagsprobleme. Diese werden aber inzwischen auch schon zu einem großen Teil von der standardisierten JSTL[8] abgedeckt. Hier nicht aufgeführt sind die Template- und Tiles-Bibliotheken.

Bean Tags: Erzeugen von Beans aus Cookies, Ressourcen, Requestparametern, Headerwerten oder aus Struts-Konfigurationsobjekten. Ausgabe von Bean-Properties und internationalisierten Nachrichten.

HTML Tags: Viele Tags zur Interaktion mit Struts ActionForms. Anzeigen von (Fehler) Meldungen aus Struts Actions. Link und URL Tags für einfaches Benutzersessionmanagement. Die Tags können sowohl für HTML-, als auch für XHTML-Generierung verwendet werden.

Logic Tags: Tags zum Vergleichen, Prüfen auf (Un-)Gleichheit, Vorhandensein etc. verschiedener Werte und davon abhängiges Generieren von Ausgaben. Iterieren über eine Menge von Elementen.

Nested Tags: Weiten die anderen Struts Tags auf verschachtelte Beaninstanzen aus.

1.2.4 Modellkomponenten

Im Modellbereich des MVC-Paradigmas bietet Struts wenig über die von Java Servlets hinausgehende Unterstützung für bestimmte Techniken und Technologien. Es macht aber auch keine Einschränkungen. Empfohlen wird von den Struts-Entwicklern jedoch, für jeden Bereich von MVC eigene JavaBeans zu verwenden, um eine möglichst gute Wiederverwendbarkeit der einzelnen Komponenten zu gewährleisten, so auch für den Modellbereich.

JavaBeans und ihr Kontext: In JSP-Webanwendungen und damit auch mit dem Struts-Framework können JavaBeans in mehreren verschiedenen Kontexten zur späteren Verwendung abgelegt werden.

- **Page** Beans im Pagekontext sind nur für die aktuelle JSP-Seite sichtbar.

- **Request** Requestbeans haben Gültigkeit für die Dauer des aktuellen Http-Requests und sind damit zur Kommunikation zwischen View- und Controller-Bereich geeignet.
- **Session** Beans im Sessionkontext gelten für die aktuelle Benutzersession über mehrere Requests hinweg.
- **Application** Applicationbeans bestehen, bis die Webanwendung als ganze beendet wird.

ActionForm-Beans Struts-ActionForm-Beans können für den Request- und den Sessionkontext definiert werden, sollten aber immer als Controller-Objekte verstanden werden und sie sollten keine Geschäftslogik enthalten.

Systemzustandbeans sind Beans, die den aktuellen Zustand der Anwendung widerspiegeln. Das wären z.B. Informationen über einen angemeldeten Benutzer und dessen Einkaufswageninhalt. Diese Beans befinden sich oft im Sessionkontext und werden auch oft in Datenbanken o.ä. persistent gemacht.

Geschäftslogikbeans kapseln die Geschäftslogik und auch oft Datenpersistenz einer Anwendung von den Controller- und View-Komponenten ab und werden meist im Application-Kontext der Anwendung abgelegt. Die geschäftslogischen Vorgänge in solchen Beans können einfach sein oder sich auch über komplexe Vorgänge auf EJBs und Datenbanken erstrecken.

Plugins Bei Struts ist ein Plugin einfach eine Klasse, die dem `PlugIn` Interface genügt, welches lediglich eine Methode zur Initialisierung und eine zum Beenden des Plugins definiert. Pluginklassen können verwendet werden, um leicht Geschäftslogikbausteine beim Start der Webanwendung verfügbar zu machen und beim Beenden wieder abzubauen. Bei der Initialisierung bekommt man eine Referenz auf das ActionServlet und etwaige Konfigurationsparameter für das Plugin mitgeliefert, die in der Strutskonfiguration eingetragen werden können, sodass man Plugins, wenn sie richtig geschrieben sind, auf einfache Weise mit verschiedenen Einstellungen für unterschiedliche Webapplikationen verwenden kann.

DataSources Zur Unterstützung von Geschäftslogik, die auf externe Datenquellen, wie z.B. relationale Datenbanken angewiesen ist, kann man durch einfache Einträge in der Strutskonfiguration Datenquellen zur Verwendung in den Actions einblenden, die dem standardisierten Interface `javax.sql.DataSource` entsprechen. Dabei wird für diese Datenbank sogar über eine Bibliothek aus dem Jakarta-Commons-Projekt ein Verbindungspool aktiviert. Die Struts Entwickleranleitung empfiehlt allerdings für den Fall, dass ein mächtigeres Werkzeug für die Datenbankabstrahierung und das Verbindungsmanagement, wie z.B. eine JNDI-Implementierung, zur Verfügung steht, lieber diese zu verwenden. Benutzt man also den Tomcat Webcontainer, so sind die Struts-DataSources hinfällig.

1.3 Systemüberblick

Wir benutzen folgende Paradigmen, Frameworks und Libraries:

- **MVC Entwurfsmuster**
  Eine Einführung ist im Kapitel 1.1 zu finden.
  http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/web-tier/web-tier5.html
- **Struts**
  Eine Einführung ist im Kapitel 1.2 zu finden.
  http://jakarta.apache.org/struts/
- **Hibernate**
  Die von uns verwendete Datenbankabsttraktionsschicht. Hibernate stellt ein Mapping von Objekt- zu Relationalschema zur Verfügung und unterstützt transparente Persistenz von Java Objekten in SQL Datenbanken. Es unterstützt 15 DBMS[4] und bringt eine eigene Datenbankabfragesprache (HQL) mit, die eine objektorientierte Erweiterung zu SQL ist.
  Siehe Kapitel 2.8
  http://www.hibernate.org/
- **Struts-Workflow-Extension**
  Eine Erweiterung, die Struts mit Workflow Unterstützung bereichert. Wir verwenden sie, um unzulässige Aufrufe und Aufrufkombinationen von Benutzern zu verhindern. So ist es z.B. im Schnupperbestellungsprozess nach der Eingabe der Adressdaten nur gestattet, zur Seite zu wechseln, die die Zahlungsweise abfragt. Ruft der Benutzer per Hand andere Seiten auf, wird dies unterbunden.
  http://www.livinglogic.de/Struts/

4 Database Management System. z.B. MySQL, Oracle, DB2

1.3.1 Paketeinteilung

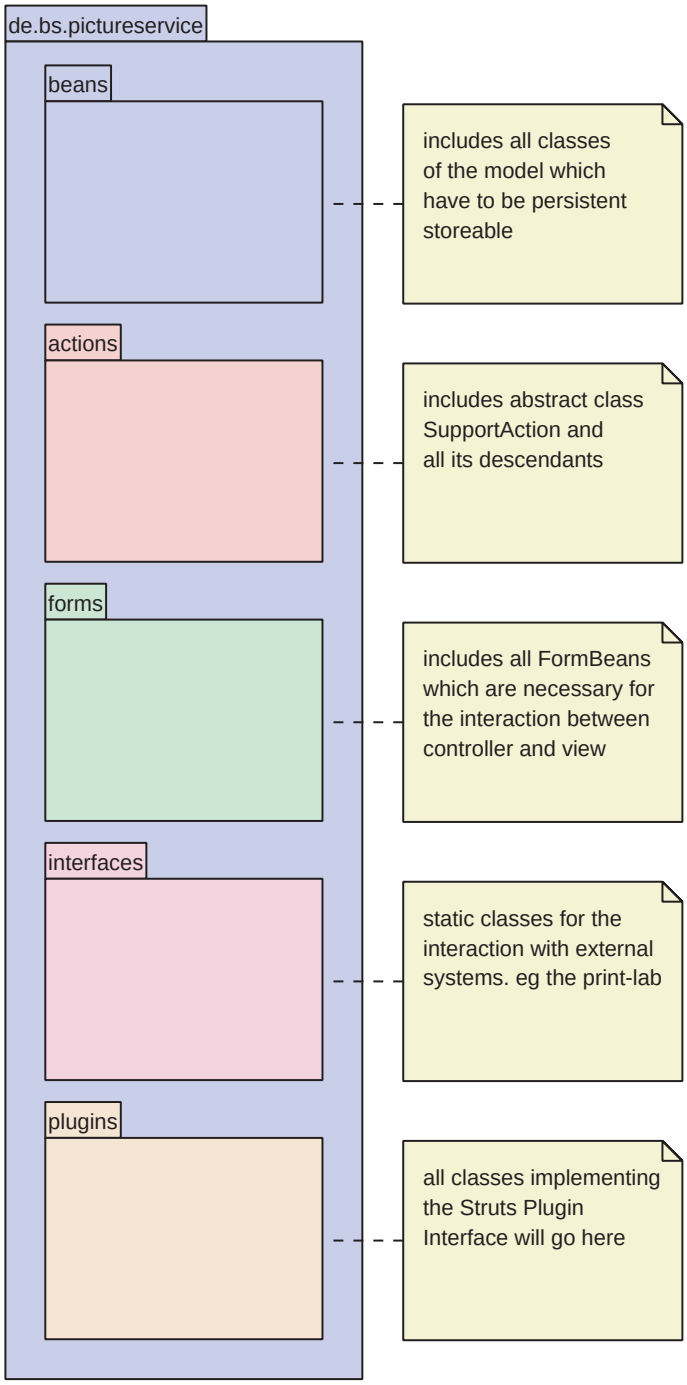

Abbildung 1.4 Paketeinteilung

Paket de.bs.pictureservice

Dieses Paket ist das Hauptpaket der Webanwendung. In ihm sind alle weiteren Pakete enthalten. Außerdem enthält dieses Paket Klassen des Modells, die nicht persistent speicherbar sein müssen. (`ShoppingCart`, `PriceList`, `FormatList`, `FileManager`, `ImageUtils`)

Paket de.bs.pictureservice.beans

Enthält alle Klassen des Modells, die persistent speicherbar sind.

Paket de.bs.pictureservice.actions

Enthält die Controller-Klasse `SupportAction` und alle von ihr abgeleiteten Action-Klassen.

Paket de.bs.pictureservice.forms

Enthält alle `ActionForm`s (auch FormBeans genannt).

Paket de.bs.pictureservice.interfaces

Hier sind statische Klassen angesiedelt, die den Informationsaustausch mit externen Systemen regeln.

Paket de.bs.pictureservice.plugins

Alle von unserer Webanwendung benötigten Struts-Plugins sind hier eingeordnet.

1.4 Überblick über die verbleibenden Kapitel

Kapitel 2 erläutert zunächst die aus dem Pflichtenheft abgeleitete Geschäftslogik (Model Komponente). Des Weiteren finden sich hier auch Beschreibungen der wichtigesten Algorithmen, wie beispielsweise die Kreditkarten-Plausibilitätsprüfung. Hier wird auch die Datenbankanbindung mittels Hibernate erläutert.

In Kapitel 3 wird dann näher auf die Verwaltung der Oberfläche (View, Sichtkomponente) eingegangen. Schwerpunkt bilden dabei ein Automat nach Harel, der die Seitenübergänge innerhalb der Webapplikation festlegt sowie die Beschreibung der für die Benutzerinteraktion benötigten FormBeans (`ActionForm`s).

In Kapitel 4 wird der prinzipielle Aufbau der Steuerungskomponente (Controller) genauer spezifiziert. Dabei wird ebenfalls auf die Beziehung dieser Komponente zur Geschäftslogik und zur Sichtkomponente dargelegt.

Kapitel 5 beschreibt dann die in der Webapplikation eingesetzen `Action`s, wobei das Augenmerk auf der Beschreibung des dynamischen Ablaufmodells besteht. Ausgehend von den im Pflichtenheft definierten Geschäftsprozessen und Produktfunktionen werden daher verschiedenste Szenarien entwickelt und unter anderem mit Hilfe der `Action`s visualisiert.

Kapitel 6 schließlich beinhaltet Dokumente und deren Typ-Definitionen. Beispielsweise werden Beispiel-XML-Dateien für Bonitätslisten und Abrechnungslisten vorgestellt.

2 Die Model Komponente

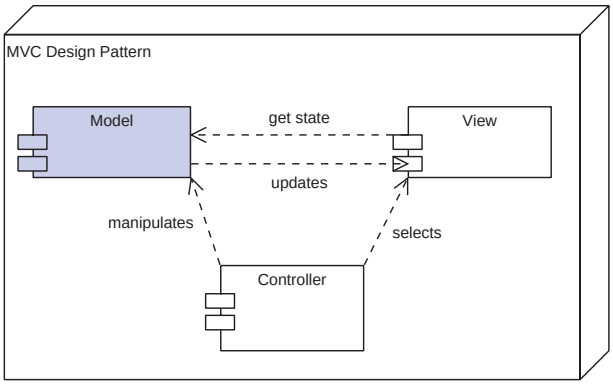

Abbildung 2.1 Einordnung in das MVC Entwurfsmuster

Die Model Komponente des MVC Entwurfsmusters kapselt die Geschäftslogik einer Webapplikation von den Sicht- und Steuerungskomponenten (siehe Abbildung 2.1). Außerdem stellt sie in aller Regel das Bindeglied zu einer Datenbank dar, falls Objekte persistent gespeichert werden müssen. In diesem Kapitel werden wir uns zunächst darauf beschränken, das statische Modell der Geschäftslogik des Online-Bilderservice mit Hilfe von UML-Klassendiagrammen zu beschreiben.[5] Dabei wurde darauf Wert gelegt, die im Pflichtenheft definierten Entitäten und Prozesse so gut wie möglich in das Software-Design zu übertragen. Diese Kontinuität des Designs der Webapplikation wird sich sowohl explizit (durch Referenzen auf das Pflichtenheft) als auch implizit (in der Konstruktion des Modells selbst) in dem hier vorliegenden Designheft widerspiegeln.

Im Folgenden werden wir demnach auf alle Klassen der Geschäftslogik eingehen. Dabei wird das ganze Modell aus Gründen der Übersichtlichkeit und Darstellbarkeit in mehrere zusammenhängende Teile aufgespalten. Des Weiteren werden wir die Beschreibung kurz und verständlich halten ohne jedoch wichtige Feinheiten aus dem Auge zu verlieren.

Hinweise zur Visualisierung: Alle in diesem Kapitel beschriebenen Klassen werden der Geschäftslogik zugerechnet. Jedoch werden immer nur die momentan zu erläuternden Klassen blau eingefärbt und vollständig dargestellt sein. Klassen, die nicht eingefärbt sind, werden an einer anderen Stelle dieses Kapitels beschrieben. Die Diagramme sind so gehalten, dass sie zu einem vollständigen Klassendiagramm zusammengefügt werden könnten. Es sind demnach auch alle Assoziationen zwischen diesen Klassen enthalten. Namespaces sind grundsätzlich so kurz wie möglich dargestellt. Alle vollständig dargestellten Klassen, denen eine explizite Angabe des Namespaces fehlt, befinden sich im Paket `de.bs.pictureservice.beans`. Der Stereotyp `<<entity>>` besagt, dass Objekte dieser Klasse persistent gespeichert werden müssen.

5 Eine genauere Beschreibung des dynamischen Modells findet sich weiter unten, da hierfür zumindest die Controller Komponente einbezogen werden sollte; ohne diese macht eine Beschreibung des dynamischen Modells wenig Sinn.

2.1 Benutzer

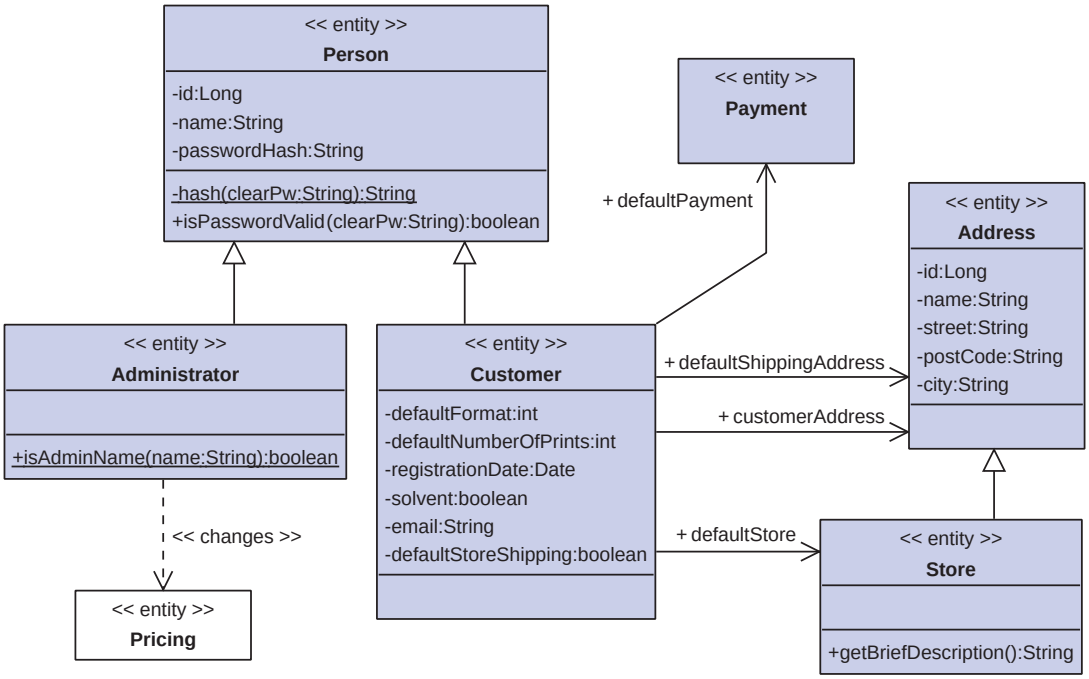

Abbildung 2.2 Dieses Klassendiagramm zeigt die Klassen, die Kunden und Administrator modellieren.

Prinzipiell gibt es zwei Arten von Benutzern der Webapplikation: Kunden und einen Administrator. Beide gemeinsam besitzen einen Benutzername (`Person.name`) und ein Kennwort. Letzteres wird gehasht gespeichert (`Person.passwordHash`). Wie ein Kennwort von der Klartext-Version in die zu gehashte Variante umgewandelt wird, wird weiter unten in diesem Abschnitt erläutert.

Zu jedem Kunden müssen darüber hinaus weitere personenbezogene Daten gespeichert werden.[6] Dazu zählen hauptsächlich die Standard-Kundenadresse (`Customer.customerAddress`), die Standard-Lieferadresse (`Customer.defaultShippingAddress`), Standard-Bildformat (`Customer.defaultFormat`) und Anzahl (`Customer.defaultNumberOfPrints`), die Standard-Filiale (`Customer.defaultStore`) und die Standard-Zahlungsweise (`Customer.defaultPayment`, siehe Abschnitt 2.2) sowie das Bonitätskennzeichen (`Customer.solvent`).

Erwähnenswert ist weiterhin, dass der Administrator die Möglichkeit besitzt, Preise und Sonderaktionen festzulegen.[7] Dies geschieht über Objekte des Typs `Pricing` (siehe Abschnitt 2.7).

6 Das sogenannte Kundenprofil (vgl. Pflichtenheft /D10/).
7 Siehe Pflichtenheft /F60/.

Kryptografisches Hashen von Passwörtern. Der wichtigste Algorithmus für diesen Teil der Geschäftslogik ist das Hashen der Passwörter (`Person.hash(clearPw: String):String`), wofür das SHA1[8]-Verfahren verwandt wird. Angedeutet ist die Verwendung dieses Verfahrens in folgendem Java-Quelltext:

```
java.security.MessageDigest md = MessageDigest.getInstance("SHA");
md.update(theTextToDigest);
byte[] digest = md.digest();
```

Da das Byte-Array `digest` Sonderzeichen enthalten kann, und diese mit der Datenbankimplementierung Schwierigkeiten machen könnten, wird der Byte-Array `digest` noch im BASE64-Format[9] kodiert, um ihn als normale ASCII-Zeichenkette speichern zu können.

2.2 Zahlungsweise

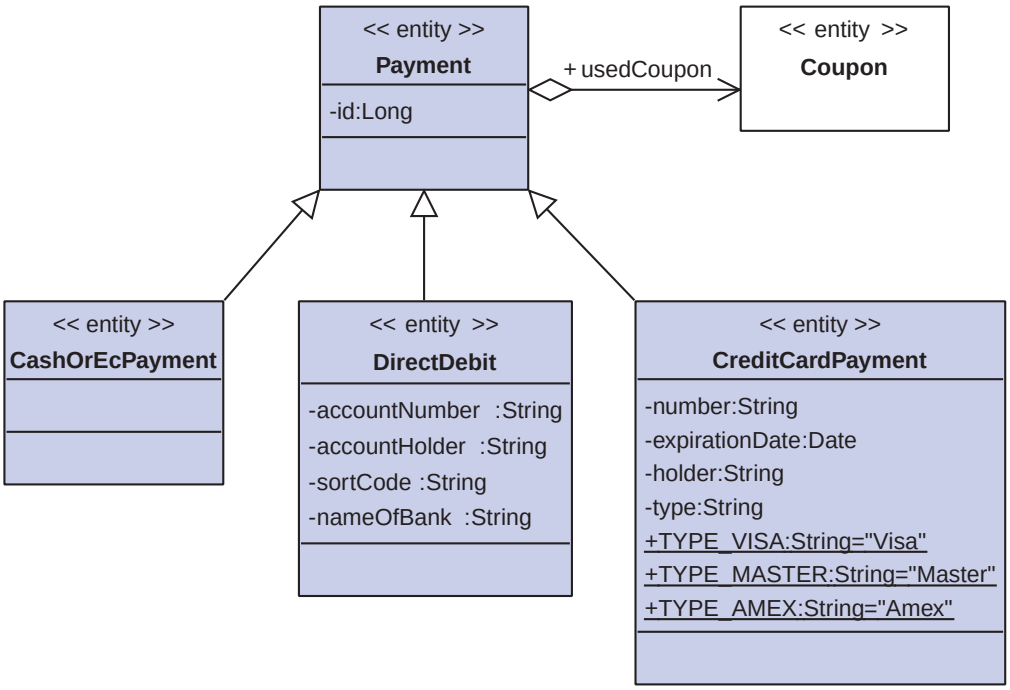

Abbildung 2.3Dieses Klassendiagramm zeigt die möglichen Zahlungsweisen.

Im Grunde hat jeder Kunde die Möglichkeit zwischen vier verschiedenen Zahlungsweisen zu wählen: Zahlung per Kreditkarte (`CreditCardPayment`), Zahlung per Lastschriftverfahren (`DirectDebitPayment`), Barzahlung oder Zahlung per EC-Karte (`CashOrEcPayment`). Da die beiden Letzteren nur bei einer Filiallieferung eine Rolle spielen und keine extra Informationen benötigen, können sie hier zusammengefasst behandelt werden.

8 Siehe http://www.w3.org/PICS/DSig/SHA1_1_0.html
9 Siehe http://www.freesoft.org/CIE/RFC/1521/7.htm

Nicht zu vernachlässigen sind im Übrigen auch Gutscheine (`Coupon`s, siehe Abschnitt 2.3), die als Zahlungsmittel in Anspruch genommen werden können, solange sie gültig sind. Es ist nicht möglich, mehr als einen Gutschein je Bestellung einzulösen.

Kreditkarten Plausibilitätsprüfung. Sie basiert zunächst auf den Unterschieden der verschiedenen Kartentypen:

Mastercard:

Länge der Kreditkartennummer: 16 Ziffern.

Beginnt mit: 51 oder 55.

Visa:

Länge der Kreditkartennummer: 13 oder 16 Ziffern.

Beginnt mit: 4.

American Express:

Länge der Kreditkartennummer: 15 Ziffern.

Beginnt mit: 34 oder 37.

Diners Club:

Länge der Kreditkartennummer: 14 Ziffern.

Beginnt mit: 300, 305, 36 oder 38..

Die Validierung läuft dann wie folgt ab:

1. Überprüfe die Kreditkartennummer auf ungültige Zeichen.
2. Überprüfe die Länge und den Beginn der Nummer nach obigen Angaben.
3. Berechne die Prüfsumme nach der LUHN-Formel[10] mit Hilfe der Modulo-10 Operation:
    1. Ausgehend von der zweiten Ziffer von rechts, verdopple den Wert jeder zweiten Ziffer der Kreditkartennummer.
    2. Addiere die nicht verdoppelten Ziffernwerte zu den in Schritt 3.1 gewonnenen Werten.
    3. Die in Schritt 3.2 errechnete Summe muss durch 10 teilbar sein. D.h. die letzte Ziffer muss eine Null sein.
4. Überprüfen, ob das Gültigkeitsdatum der Kreditkarte in der Zukunft liegt.

10 Siehe http://www.beachnet.com/~hstiles/cardtype.html.

2.3 Produkte[11]

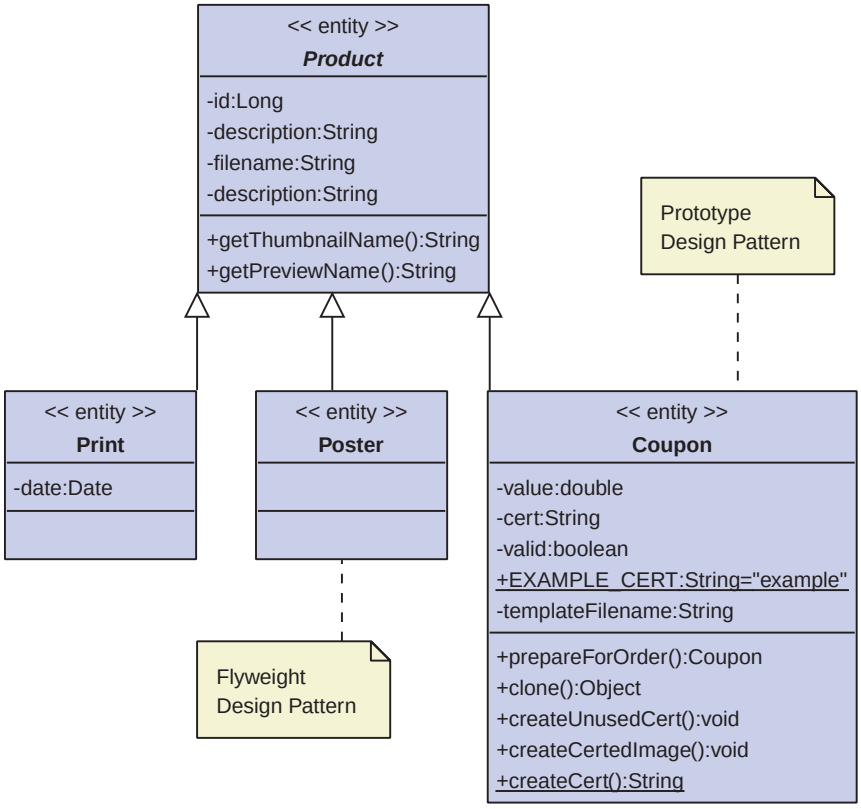

Abbildung 2.4 Dieses Klassendiagramm zeigt die Typen der Produkte, die von einem Kunden innerhalb der Webapplikation Online-Bilderservice erworben werden können.

Produkte, die über die Webapplikation Online-Bilderservice erstanden werden können, sind Ausdrucke von hochgeladenen Fotos eines Kunden (`Print`), Poster (`Poster`) und Gutscheine (`Coupon`).

Gutscheine haben selbstverständlich einen Gegenwert (`Coupon.value`) und ein Zertifikat (`Coupon.cert`), das zur Identifizierung dient. Die Gültigkeit des Gutscheins muss ebenfalls gespeichert werden (`Coupon.valid`). Ein solches Zertifikat ist eindeutig und muss daher bei jedem Erwerb wie unten beschrieben erstellt werden (`Coupon.createUnusedCert()`). Ebenso muss für jeden Gutschein ein entsprechendes Bild mit dem integrierten Zertifikat erzeugt werden (`Coupon.createCertedImage()`) Gutscheine entstehen endsprechend dem Prototyp Entwurfsmuster aus den Gutscheinvorlagen (`Coupon.clone()`).

Ausdrucke von Fotos sind zusätzlich mit einem Datum assoziiert, damit alte Fotos zeitgesteuert gelöscht werden können.[12]

11 Siehe Pflichtenheft /D30/, /D40/, /D80/
12 Siehe Pflichtenheft /F70/,

Für Poster wird das Fliegengewicht Entwurfsmuster verwendet; in allen Bestellungen werden die selben Posterobjekte referenziert.

Erzeugen von Gutscheinzertifikaten. Zertifikate, also eindeutige Nummern, die das Einlösen eines Gutscheines ermöglichen, werden wie nachfolgend beschrieben, erzeugt:

1. Erzeuge eine Zufallszahl.
2. Hashe diese Zahl nach dem SHA1-Verfahren.
3. Kodiere das Ergebnis im BASE64-Format.
4. Extrahiere eine 20 Zeichen große Teilmenge dieses SHA1-gehashten Wertes im BASE64-Format (bspw. die ersten 20 Zeichen).
5. Überprüfe, ob dieses Zertifikat bereits genutzt wird. Falls ja, gehe zu Schritt 1.

2.4 Bestellungen

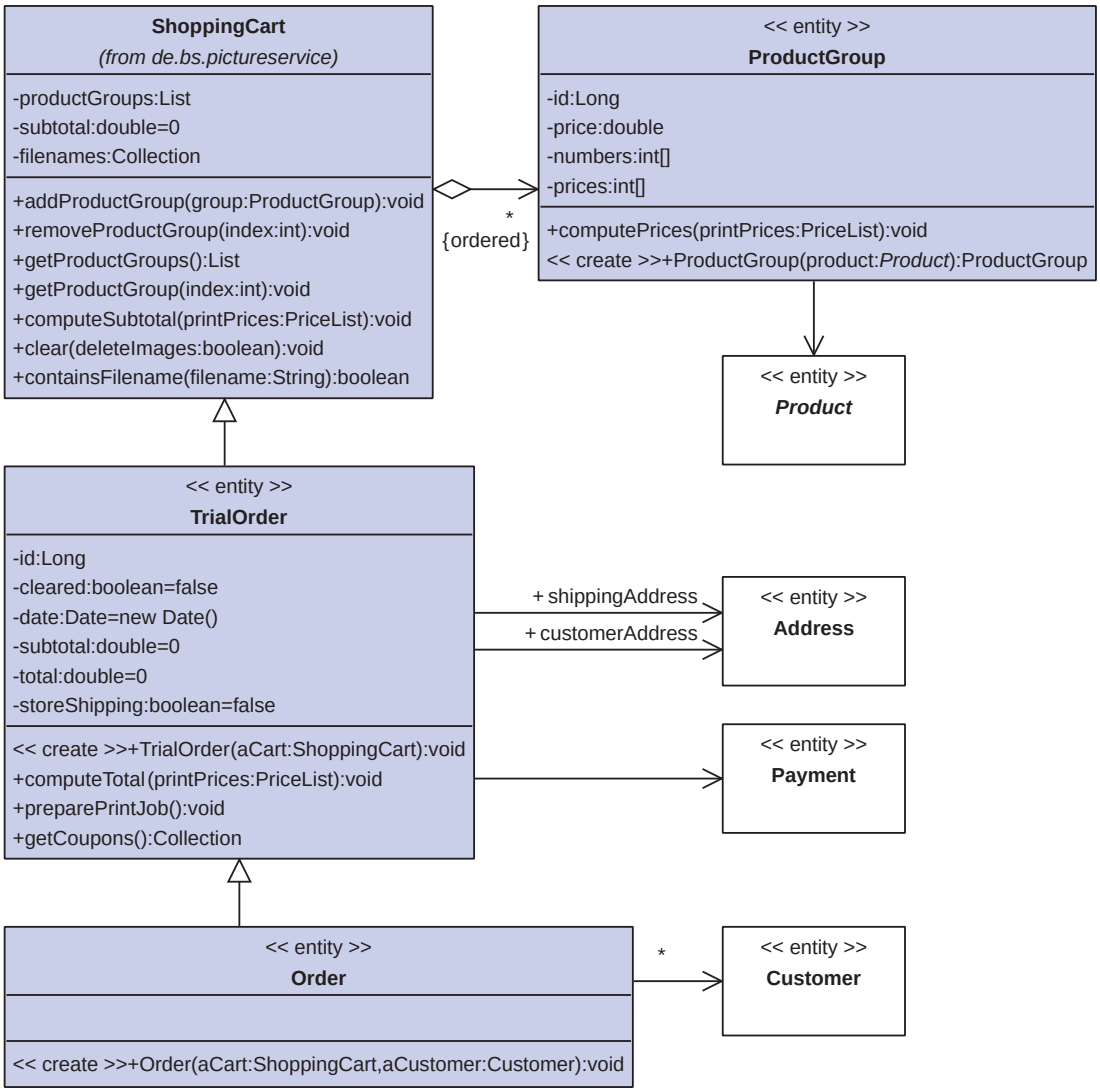

Abbildung 2.5 Dieses Klassendiagramm illustriert die im Bestellprozeß anfallenden Entitäten.

Der wichtigste Teil der Geschäftslogik ist die Modellierung des Bestellprozesses. Da der zu entwickelnde Online-Bilderservice über das Alleinstellungsmerkmal einer Bestellung ohne Login verfügt, gibt es normale Bestellungen (`Order`) sowie Schnupperbestellungen (`TrialOrder`).[13] Da während einer Bestellung, bei der der Kunde eingeloggt ist, zusätzlich zu den bei einer Schnupperbestellung anfallenden Daten noch dessen Standardeinstellungen verfügbar sein müssen, ist `Order` als eine Generalisierung von `TrialOrder` modelliert.

13 Siehe Pflichtenheft /F10/ und /F20/, /D60/

Bei beiden Bestellarten müssen Adressdaten (`shippingAddress` und `customerAddress`) ebenso wie die Zahlungsweise gespeichert werden. Diese werden bei einer Bestellung, bei der der Kunde bereits eingeloggt ist, mit dessen Standardeinstellungen vorbelegt. Des Weiteren sind eine Auftragsnummer, das Auftragsdatum und die Bestellsumme (`TrialOrder.total`) festzuhalten. Das Attribut `TrialOrder.cleared` zeigt an, ob die Bestellung bereits von der automatischen Systempflege in die Abrechnungsliste übertragen wurde.

Selbstverständlich müssen aber auch sämtliche Produkte, die der Kunde bestellt, vermerkt werden. Damit es möglich ist, für eine Bilddatei mehrere Formate zu ordern ohne sie mehrfach hochzuladen, speichert `ProductGroup` für jede Bilddatei die gewünschten Formate mit deren Anzahl von Abzügen. Darüber hinaus werden die Preise gesichert, damit sie jederzeit (auch nach Preisänderungen) eingesehen werden können. Dies geschieht über ein Objekt vom Typ `PriceList`, welches eine Kopie der Preise aller Formate während einer Kundensitzung enthält (siehe Abschnitt 2.7). Dadurch wird gewährleistet, dass ein Kunde während einer Session immer die gleichen Angebote wahrnehmen kann.

Der Kunde nutzt für die Bestellung im Wesentlichen einen Warenkorb (`ShoppingCart`) der einzelne Positionen (Fotos, Gutscheine, Poster) und deren Eigenschaften (Formate, Anzahlen) enthält.

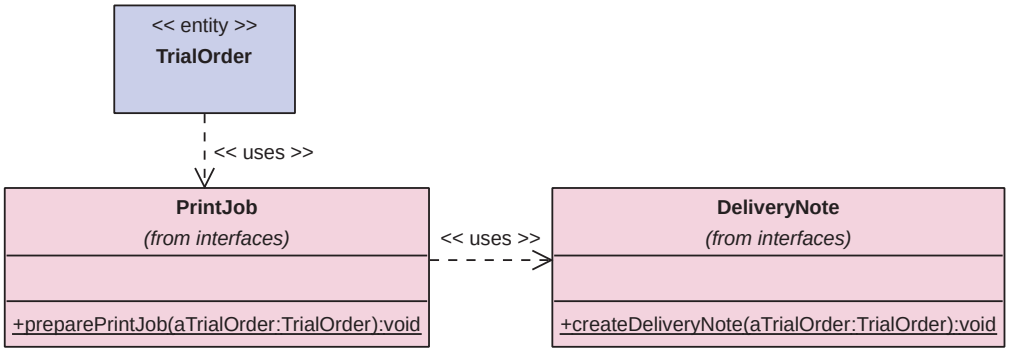

Abbildung 2.6 Erzeugen eines Lieferscheins.

Auch wenn hier nicht auf die Abläufe innerhalb der Webapplikation eingegangen werden soll, ist es vielleicht hilfreich, kurz zu erwähnen, wann welche Objekte im Bestellprozeß erzeugt werden.

Nach einem eventuellen Hochladen von Fotos werden diese sofort als `ProductGroup`s in den Warenkorb eingefügt. Erst beim Bestellen des Warenkorbinhaltes wird ein Objekt vom Typ `TrialOrder` (`Order`) erzeugt. Erst nach einer weiteren Bestätigung wird dann ein Lieferschein erstellt und mit den entsprechenden Fotos/Postern/Gutscheinen persistent gespeichert.

Ist der Kunde mit seiner Auswahl an Produkten zufrieden und sendet die Bestellung endgültig ab, so wird über `PrintJob` ein Lieferschein[14] (`DeliveryNote`) erzeugt (siehe auch Abbildung).

2.5 Systemdienste

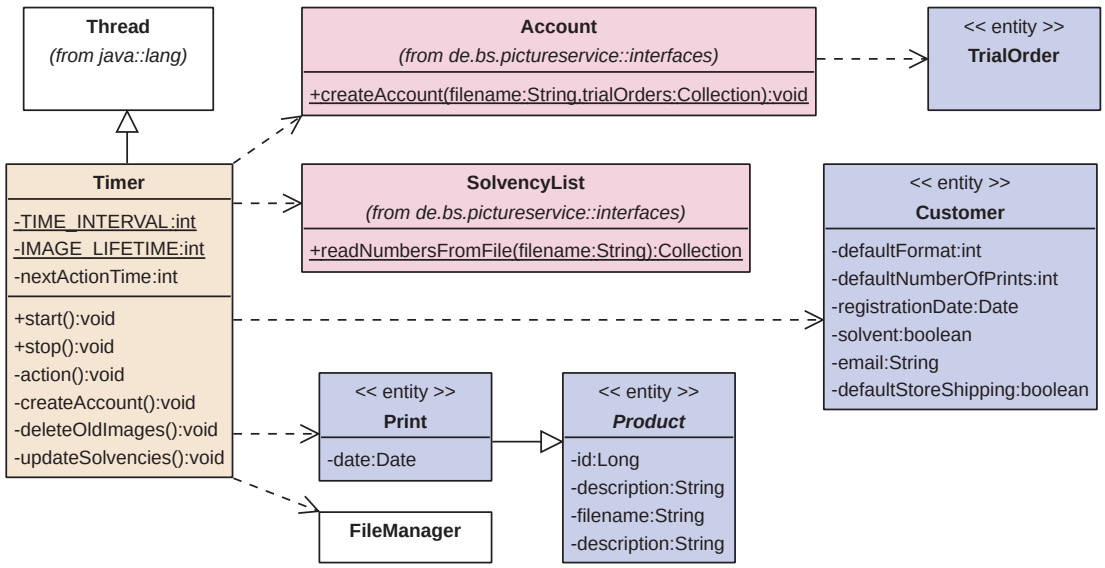

Abbildung 2.7 Automatische Systemdienste.

Die im Pflichtenheft unter Produktfunktion /F70/ beschriebenen Systemdienste sind wie in Abbildung 2.7 dargestellt modelliert. Das tägliche Erzeugen einer Abrechnungsliste (`Account`), das Einlesen der Bonitätsliste (`SolvencyList`) und das Löschen alter Fotos werden über einen Zeitgeber (`Timer`), der bei Applikationsstart initialisiert wird, gesteuert. [15]

14 Das Format eines Lieferscheins ist in Kapitel 6 als XML-Schema definiert.
15 Siehe Pflichtenheft /D50/

2.6 Formate

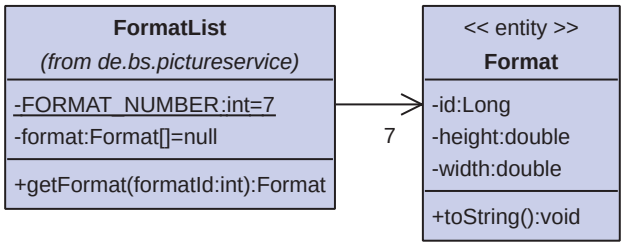

Abbildung 2.8Modellierung der Bildformate

Druckformate, die von der Anwendung zu verwalten sind gibt es genau sieben Stück[16]. Diese werden von der `FormatList` gemanaged. Das Format selbst besitzt eine Höhe (`height`) und eine Breite (`width`).[17]

2.7 Preise

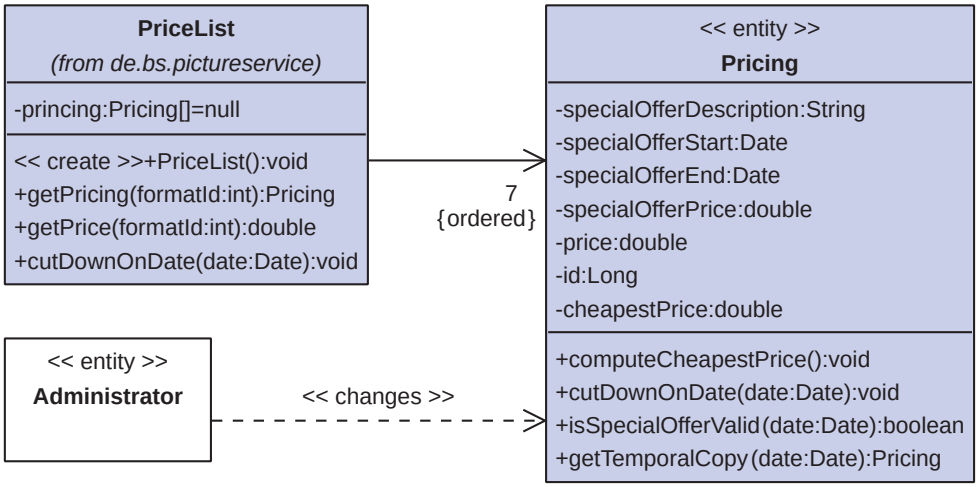

Abbildung 2.9 zeigt die Modellierung der Preise, Sonderaktionen und einer Managementklasse

In diesem Abschnitt wollen wir uns kurz damit beschäftigen, wie Preise für einen Kunden während einer Session konstant gehalten werden können, ohne dem Administrator Einschränkungen bezüglich der Zeitpunkte von Preisänderungen auferlegen zu müssen.

Die einfache Lösung dieses Problems besteht darin, dass ein zentral gespeicherter Satz von Preisen (ein Preis für jedes Format) existiert, der vom Administrator jederzeit geändert werden darf. Von diesem zentral gespeicherten Satz von Preisen wird dann für jede Kundensitzung eine Kopie angelegt, die dem Kunden einheitliche Preise während seines Besuchs des Bilderservices garantiert. Dazu dienen die Methoden

16 Anhang des Pflichtenheftes
17 Siehe Pflichtenheft /D20/

`cutDownOnDate(date:Date)` in `PriceList` und `Pricing` sowie `getTemporalCopy (date:Date)` in `Pricing`.
Der Satz von Preisen (einer für jedes der sieben verschiedenen Formate) wird durch `PriceList` modelliert (siehe Abbildung 2.9). `Pricing` hingegen enthält alle nötigen Preisinformationen für ein bestimmtes Format. Dazu zählen insbesondere eine Beschreibung (`specialOfferDescription`), Start (`specialOfferStart`) und Ende (`specialOfferEnd`) einer möglicherweise aktiven Sonderaktion. Die Gültigkeit von Sonderaktionen wird mittels `isSpecialOfferValid(date: Date)` überprüft (Aktuelles Datum liegt zwischen Start- und Endzeitpunkt). Die Methode `computeCheapestPrice()` wird verwendet um den momentan günstigsten Preis für das Format zu berechnen.[18]

18 Siehe Pflichtenheft /D20/

2.8 Datenbank

Obwohl wir Hibernate für die Datenbankzugriffe benutzen, haben wir beschlossen, eine zentrale Datenbankfassade, die Klasse `DatabaseAccessor`, zu entwicklen. Sie enthält alle vom Modell und der Steuerungskomponente benötigten Methoden zum Laden und Speichern von Objekten, welche mit dem Stereotyp `<<entity>>` versehen sind. Und nur diese Fassade benutzt Hibernate zum persistenten Speichern von Objekten. Damit entkoppeln wir die Webanwendung von der benutzten Datebankabstraktionsschicht, so dass man diese bei Bedarf leicht auswechseln könnte.

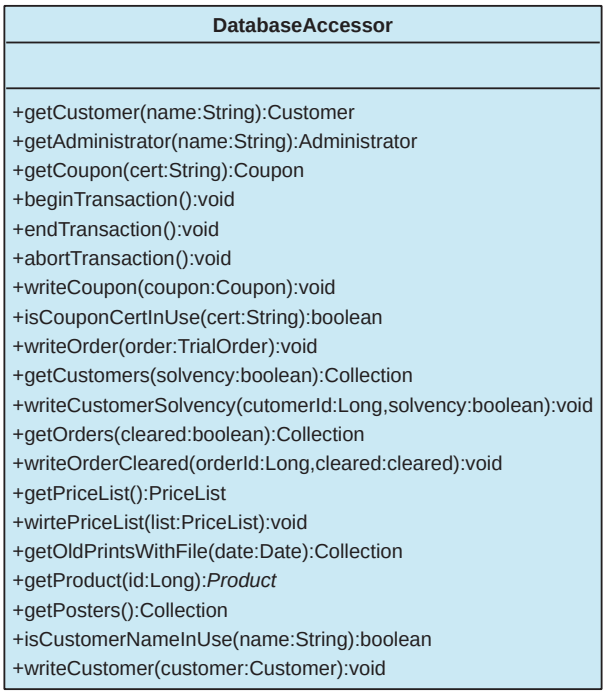

Abbildung 2.10 Die Fassade DatabaseAccessor

Methode	***Beschreibung***
`public Customer getCustomer(String name)`	Gibt den Kunden mit dem Benutzernamen `name` zurück. Wenn dieser nicht vorhanden ist, wird `null` zurückgegeben.
`public Administrator getAdministrator(String name)`	Gibt den Administrator mit dem Benutzernamen `name` zurück. Wenn dieser nicht vorhanden ist, wird `null` zurückgegeben.
`public Coupon getCoupon(String cert)`	Gibt den Gutschein mit dem Zertifikat `cert` zurück. Wenn dieser nicht vorhanden ist, wird `null` zurückgegeben.

Methode	***Beschreibung***
`public void beginTransaction()`	Startet eine Datenbank-Transaktion. Wird von den meisten Methoden automatisch aufgerufen. Sollte schon eine Transaktion geöffnet sein, wird diese benutzt.
`public void endTransaction()`	Beendet eine Datenbank-Transaktion und macht die Daten persistent.
`public void abortTransaction()`	Bricht eine Datenbank-Transaktion ab.
`public void writeCoupon(Coupon coupon)`	Schreibt den Gutschein `coupon` in die Datenbank.
`public boolean isCouponCertInUse(String cert)`	Gibt zurück, ob das Gutscheinzertifikat `cert` schon von einem anderen Gutschein benutzt wird.
`public void writeOrder(TrialOrder order)`	Schreibt eine Schnupperbestellung oder Bestellung `order` in die Datenbank.
`public Collection getCustomers(boolean solvency)`	Gibt eine `Collection` mit allen Kunden Objekten zurück, die entweder Bonität besitzen (`solvency = true`) oder keine Bonität in diesem System haben (`solvency = false`).
`public void writeCustomerSolvency(Long cutomerId, boolean solvency)`	Setzt die Bonität des Kunden mit der Kundennummer `id` auf den Wert von `solvency`.
`public Collection getOrders(boolean cleared)`	Gibt eine `Collection` aller Bestellungen zurück mit dem Abarbeitungsstatus `cleared`.
`public void writeOrderCleared(Long orderId, cleared cleared)`	Setzt den Abarbeitungsstatus der Bestellung mit der Identifikation `orderId` auf `cleared`.
`public PriceList getPriceList()`	Liest alle `Pricing` Objekte aus der Datenbank und erstellt mit ihnen eine `PriceList`.
`public void wirtePriceList(PriceList list)`	Schreibt alle in `list` enthaltenen `Pricing` Objekte in die Datenbank.

Methode	***Beschreibung***
`public Collection getOldPrintsWithFile(Date date)`	Gibt alle `Print` Objekte zurück, die älter als `date` sind und noch eine Bilddatei mit sich assoziiert haben.
`public Product getProduct(Long id)`	Liest das Produkt (Eigenes Bild, Poster, Gutschein) mit der Identifikation `id`. Wenn dieser nicht vorhanden ist, wird `null` zurückgegeben.
`public Collection getPosters()`	Gibt eine `Collection` mit allen `Poster` Objekten zurück, die in der Datenbank gespeichert sind.
`public boolean isCustomerNameInUse(String name)`	Gibt zurück, ob es schon einen Kunden mit dem Namen `name` gibt.
`public void writeCustomer(Customer customer)`	Speichert den Kunden `customer` in der Datenbank.

Beispiel für das Speichern eines Objektes:

```
public void writeCustomer(customer: Customer)
{
     beginTransaction();
     currentSession().save(customer);
     endTransaction();
     closeSession();
}
```

Beispiel für das Laden eines Objektes:

```
public Customer getCustomer(name: String){
     beginTransaction();
     List customers = currentSession().find(
          "from Customer as customer " +
          "where customer.name = ?",
          name,
          Hibernate.STRING
     );
     endTransaction();
     closeSession();
     if (customers.isEmpty()) return null;
     else return (Customer) customers.get(0);
}
```

Das Datenbankschema ist durch folgende Hibernate-Konfigurationsdatei implizit festgelegt. Für das Erstellen, Abändern und Befüllen dieses Schemas ist Hibernate zuständig.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping SYSTEM "http://hibernate.sourceforge.net/hibernate-
mapping-2.0.dtd" >
<hibernate-mapping>

	<class name="de.bs.pictureservice.beans.Person"
		table="PERSONS">
		<id name="id" type="long" unsaved-value="null" >
			<generator class="native"/>
		</id>
		<property name="name" not-null="true" type="string" length="20"/>
		<property name="passwordHash" not-null="true" type="string">
			<column
				name="passwordHash"
				sql-type="CHAR(80)"/>
		</property>
		<joined-subclass name="de.bs.pictureservice.beans.Administrator"
			table="ADMINISTRATORS">
			<key column="USER"/>
		</joined-subclass>
		<joined-subclass name="de.bs.pictureservice.beans.Customer"
			table="CUSTOMERS">
			<key column="USER"/>
			<property name="registrationDate" not-null="true"
type="date"/>
			<property name="solvent" not-null="true" type="boolean"/>
			<property name="defaultStoreShipping" not-null="true"
type="boolean"/>
			<property name="email" not-null="true" type="string"
length="255"/>
			<property name="defaultFormat" not-null="true"
type="integer"/>
			<property name="defaultNumberOfPrints" not-null="true"
type="integer"/>
			<many-to-one name="defaultPayment"
				class="de.bs.pictureservice.beans.Payment"/>
			<many-to-one name="defaultStore"
				class="de.bs.pictureservice.beans.Store"/>
			<many-to-one name="defaultShippingAddress"
				class="de.bs.pictureservice.beans.Address"/>
			<many-to-one name="customerAddress"
				class="de.bs.pictureservice.beans.Address"/>
		</joined-subclass>
	</class>

	<class name="de.bs.pictureservice.beans.Product"
		table="PRODUCTS">
		<id name="id" type="long" unsaved-value="null">
			<generator class="native"/>
		</id>
		<property name="description" type="string" length="100" not-
null="true"/>
		<property name="filename" type="string" length="255" not-
null="true"/>
		<joined-subclass name="de.bs.pictureservice.beans.Print"
			table="PRINTS">
```

```
                    <key column="PRODUCT"/>
                    <property name="date" type="date" not-null="true"/>
             </joined-subclass>
             <joined-subclass name="de.bs.pictureservice.beans.Poster"
                    table="POSTERS">
                    <key column="PRODUCT"/>
             </joined-subclass>
             <joined-subclass name="de.bs.pictureservice.beans.Coupon"
                    table="COUPONS">
                    <key column="PRODUCT"/>
                    <property name="value" type="currency" not-null="true"/>
                    <property name="cert" not-null="true"
                           type="string">
                           <column
                                  name="certificate"
                                  sql-type="CHAR(20)"/>
                    </property>
                    <property name="valid" type="boolean" not-null="true"/>
             </joined-subclass>
      </class>


      <class name="de.bs.pictureservice.beans.Payment"
             table="PAYMENTS">
             <id name="id" type="long" unsaved-value="null">
                    <generator class="native"/>
             </id>
             <discriminator type="string"/>
             <set name="usedCoupons" inverse="true">
                    <key column="CouponId"/>
                    <one-to-many class="de.bs.pictureservice.beans.Coupon"/>
             </set>
             <joined-subclass
name="de.bs.pictureservice.beans.CreditCardPayment"
                    table="CREDITCARDPAYMENTS">
                    <key column="PAYMENT"/>
                    <property name="number" not-null="true" type="string"
length="16"/>
                    <property name="expirationDate" not-null="true"
type="string" length="5"/>
                    <property name="holder" not-null="true" type="string"
length="100"/>
                    <property name="type" not-null="true" type="string"
length="6"/>
             </joined-subclass>
             <joined-subclass name="de.bs.pictureservice.beans.CashOrEcPayment"
                    table="CASHORECPAYMENTS">
                    <key column="PAYMENT"/>
             </joined-subclass>
             <joined-subclass name="de.bs.pictureservice.beans.DirectDebit"
                    table="DIRECTDEBITS">
                    <key column="PAYMENT"/>
                    <property name="accountNumber" not-null="true"
type="string" length="10"/>
                    <property name="accountHolder" not-null="true"
type="string" length="100"/>
                    <property name="sortCode" not-null="true" type="string"
length="8"/>
                    <property name="nameOfBank" not-null="true" type="string"
length="100"/>
             </joined-subclass>
```

```
	</class>


	<class name="de.bs.pictureservice.beans.Address"
		table="ADDRESSES"
		discriminator-value="N">
		<id name="id" type="long" unsaved-value="null">
			<generator class="native"/>
		</id>
		<discriminator type="char" column="isStore"/>
		<property name="name" not-null="true" type="string" length="100"/>
		<property name="street" not-null="true" type="string"
length="100"/>
		<property name="postCode" not-null="true" type="string"
length="5"/>
		<property name="city" not-null="true" type="string" length="100"/>
		<subclass name="de.bs.pictureservice.beans.Store"
			discriminator-value="Y">
		</subclass>
	</class>

	<class name="de.bs.pictureservice.beans.TrialOrder"
		table="ORDERS"
		discriminator-value="N">
		<id name="id" type="long" unsaved-value="null">
			<generator class="native"/>
		</id>
		<discriminator type="char" column="isCustomerOrder"/>
		<property name="date" not-null="true" type="date"/>
		<property name="cleared" not-null="true" type="boolean"/>
		<property name="subtotal" not-null="true" type="currency"/>
		<property name="total" not-null="true" type="currency"/>
		<set name="productGroups" inverse="true" cascade="all-delete-
orphan">
			<key column="OrderId"/>
			<one-to-many
class="de.bs.pictureservice.beans.ProductGroup"/>
		</set>
		<many-to-one name="shippingAddress"
			class="de.bs.pictureservice.beans.Address"/>
		<many-to-one name="customerAddress"
			class="de.bs.pictureservice.beans.Address"/>
		<many-to-one name="payment"
			class="de.bs.pictureservice.beans.Payment"/>
		<subclass name="de.bs.pictureservice.beans.Order"
			discriminator-value="Y">
			<many-to-one name="customer"
				class="de.bs.pictureservice.beans.Customer"/>
		</subclass>
	</class>

	<class name="de.bs.pictureservice.beans.ProductGroup"
		table="PRODUCTGROUPS">
		<id name="id" type="long" unsaved-value="null">
			<generator class="native"/>
		</id>
		<property name="numbers"
type="de.bs.pictureservice.beans.ProductGroupNumbers">
			<column name="numbers0"/>
			<column name="numbers1"/>
```

```
                        <column name="numbers2"/>
                        <column name="numbers3"/>
                        <column name="numbers4"/>
                        <column name="numbers5"/>
                        <column name="numbers6"/>
                </property>
                <property name="prices"
type="de.bs.pictureservice.beans.ProductGroupPrices">
                        <column name="prices0"/>
                        <column name="prices1"/>
                        <column name="prices2"/>
                        <column name="prices3"/>
                        <column name="prices4"/>
                        <column name="prices5"/>
                        <column name="prices6"/>
                </property>
                <many-to-one name="product"
                        class="de.bs.pictureservice.beans.Product"/>
        </class>

        <class name="de.bs.pictureservice.beans.Pricing"
                table="PRICINGS">
                <id name="id" type="long" unsaved-value="null">
                        <generator class="native"/>
                </id>
                <property name="specialOfferDescription" not-null="true"
type="string"/>
                <property name="specialOfferStart" not-null="true" type="date"/>
                <property name="specialOfferEnd" not-null="true" type="date"/>
                <property name="specialOfferPrice" not-null="true"
type="currency"/>
                <property name="price" not-null="true" type="currency"/>
        </class>

        <class name="de.bs.pictureservice.beans.Format"
                table="FORMATS">
                <id name="id" type="long" unsaved-value="null">
                        <generator class="native"/>
                </id>
                <property name="height" not-null="true" type="double"/>
                <property name="width" not-null="true" type="double"/>
        </class>


</hibernate-mapping>
```

3 Die Präsentationsschicht (View)

In diesem Kapitel gehen wir auf die View-Schicht des MVC Entwurfsmusters ein. In einer auf Struts aufbauenden Anwendung besteht diese Schicht üblicherweise aus JSP[19]-Seiten. Wir werden die JSP-Erweiterung Tiles (siehe 1.2.3) einsetzen, um Erweiterbarkeit zu gewährleisten und Redundanzen zu minimieren. Diese Tiles Integration werden wir aber im Folgenden nicht weiter behandeln, sondern die JSP-Seiten als „Black-Box“ betrachten.

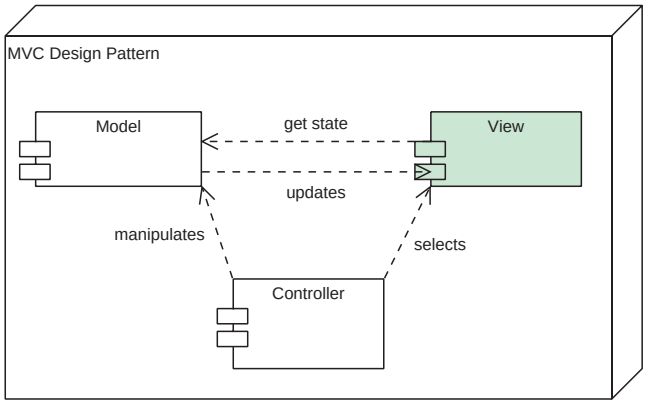

Abbildung 3.1 Einordnung in das MVC Entwurfsmuster

Außerdem werden in diesem Kapitel auch die von unserer Struts Anwendung benötigten `ActionForms` vorgestellt und eine Verbindung zwischen den JSP und den `ActionForms` gezogen.

Hinweise zur Visualisierung

Die in den anderen Kapiteln gültige Farbkodierung der Diagramme wird hier nicht eingehalten. Alle Diagramme in diesem Kapitel beziehen sich nur auf die Sicht (View) Komponente des MVC Entwurfsmusters und benutzen Farben, um einzelnen Teilbereiche dieser Schicht zu kennzeichnen.

Der Stereotyp `<<action>>` besagt, dass dieser Zustand oder Zustandsübergang als Struts-Action realisiert wird. Der Stereotyp `<<jsp>>` identifiziert Zustände und Zustandsübergänge, die als JSP-Seiten realisiert werden.

3.1 Übersicht

Abbildung 3.2 stellt die grobe Struktur der GUI dar, wie sie in dem Pflichtenheft vereinbart wurde. Die farbigen Bereiche in dieser Abbildung sind jene Bereiche der GUI, die sich bei den verschiedenen HTML-Seiten unterscheiden können. Alle anderen Bereiche sind konstant. Die variablen Seitenbereiche sind:

19 JavaServer Pages. JSP ist eine Technologie, die es erlaubt, dynamische HTML-Seiten zu erstellen. Dazu wird die HTML-Sprache durch einige Elemente erweitert, um Programmlogik direkt in der HTML-Seite einzubetten.
Siehe http://java.sun.com/products/jsp/

1. Seitentitel, Seiteninhalt, Seitennavigation und der Link zum Warenkorb
2. Login-Bereich
3. Landmarks wie z.B. Preise oder AGBs anzeigen

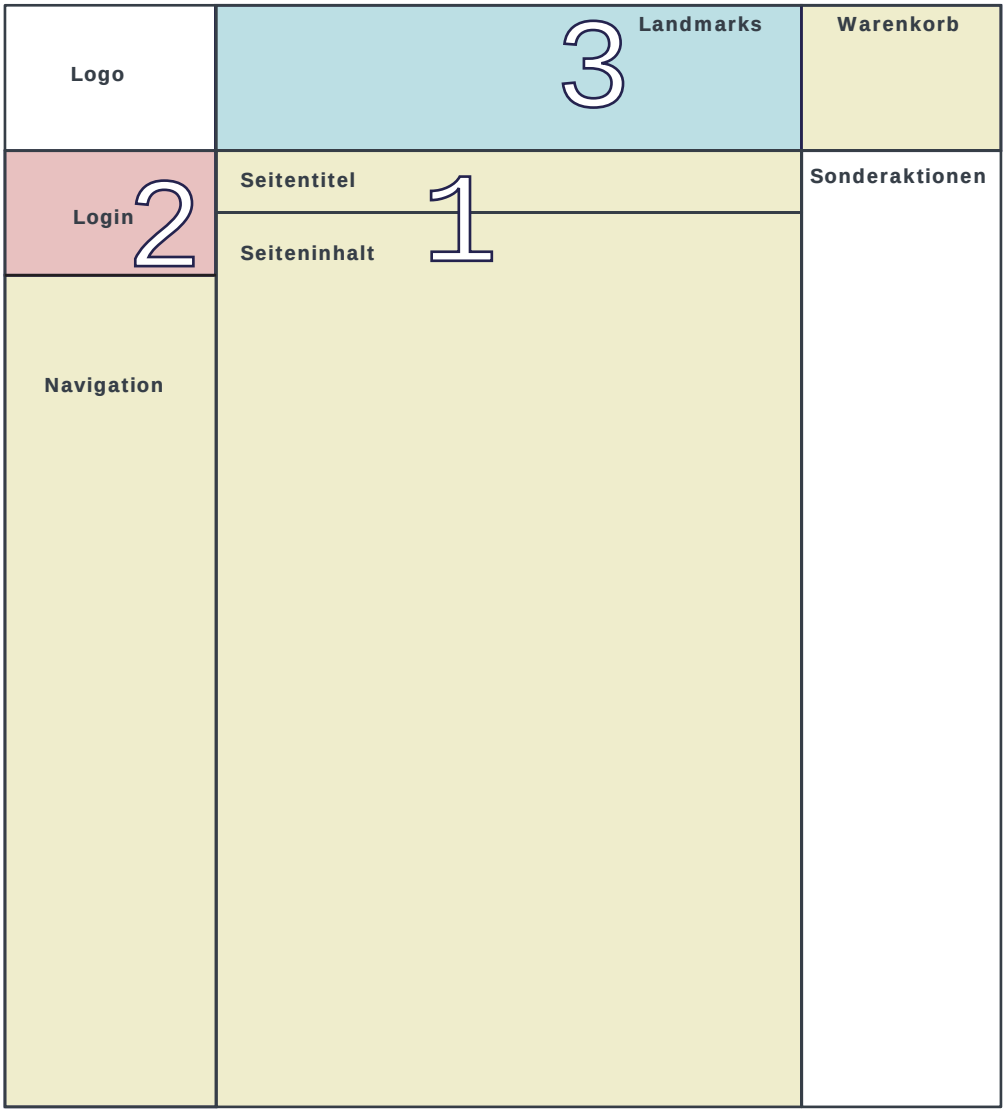

Abbildung 3.2 Struktur der GUI - Identifikation und Gruppierung der variablen Seitenteile

Anhand eines UML Zustandsdiagramms wollen wir eine Übersicht über die möglichen Zustände der GUI geben und daraus die benötigten Sicht(View)-Elemente identifizieren und spezifizieren. Wir fangen mit einem Übersichtsdiagramm an und verfeinern es in nachfolgenden Abschnitten sukzessive. Abbildung 3.4 ist ein dreifach nebenläufiger Automat, der die Nebenläufigkeit der GUI modelliert.

Vereinfachungen in dem Übersichtsdiagramm:

- Im ersten Teil sind alle Logout-Zustandsübergänge weggelassen. Wenn ein Logout möglich ist (siehe Teil 2), möge man sich einen Logout-Übergang zu dem Startzustand denken.
- Die vier Zustände Start/Profile, Product Selection, Order und Trial Order sind zusammengesetzte Zustände, die in folgenden Abschnitten detaillierter betrachtet werden. In diesem Übersichtsdiagramm betrachten wir sie als Einheit.
- Die Zustände, die für die Administrator-GUI notwendig sind, werden hier noch nicht berücksichtigt.
- Für jeden Übergang, der mit dem Stereotyp `<<action>>` gekennzeichnet ist, muss man sich noch einen Übergang vorstellen, der den Fehlerfall modelliert. D.h. dieser Übergang wird genommen, wenn während der Ausführung der Action ein Fehler auftritt (z.B. wenn eine Fehleingabe des Benutzers erkannt wurde). Abbildung 3.3 veranschaulicht dies. Die Fehlerfallübergänge werden werden wir auch in der Detailbetrachtung weiter unten nicht explizit berücksichtigen.

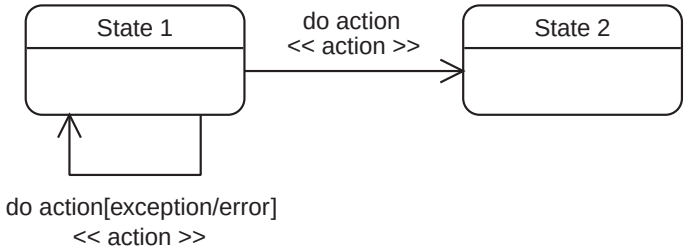

Abbildung 3.3 Beispiel eines impliziten Fehlerzustandsüberganges

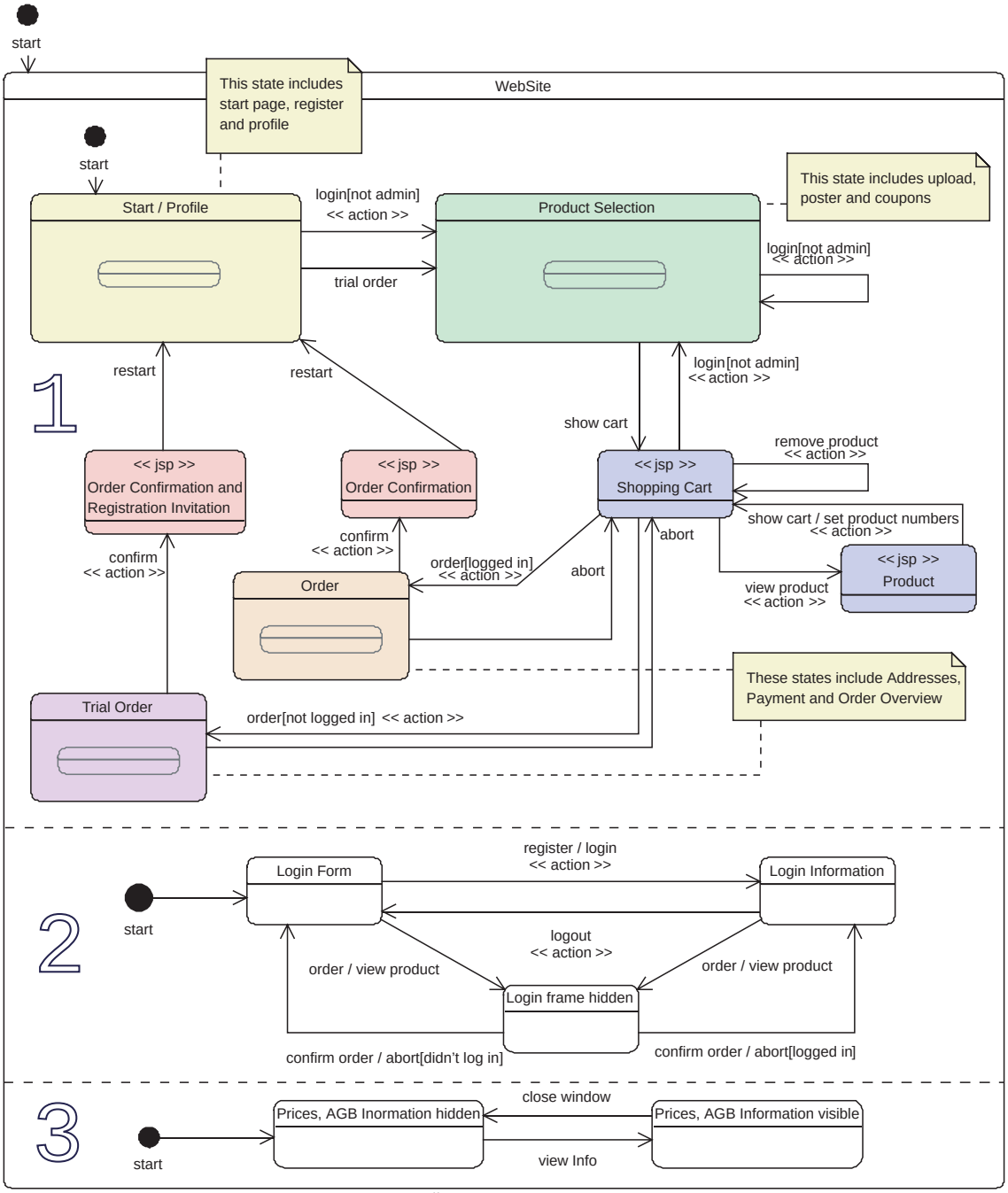

Abbildung 3.4 Zustandsdiagramm der GUI (Übersicht)

3.2 Im Detail

3.2.1 Start/Profile

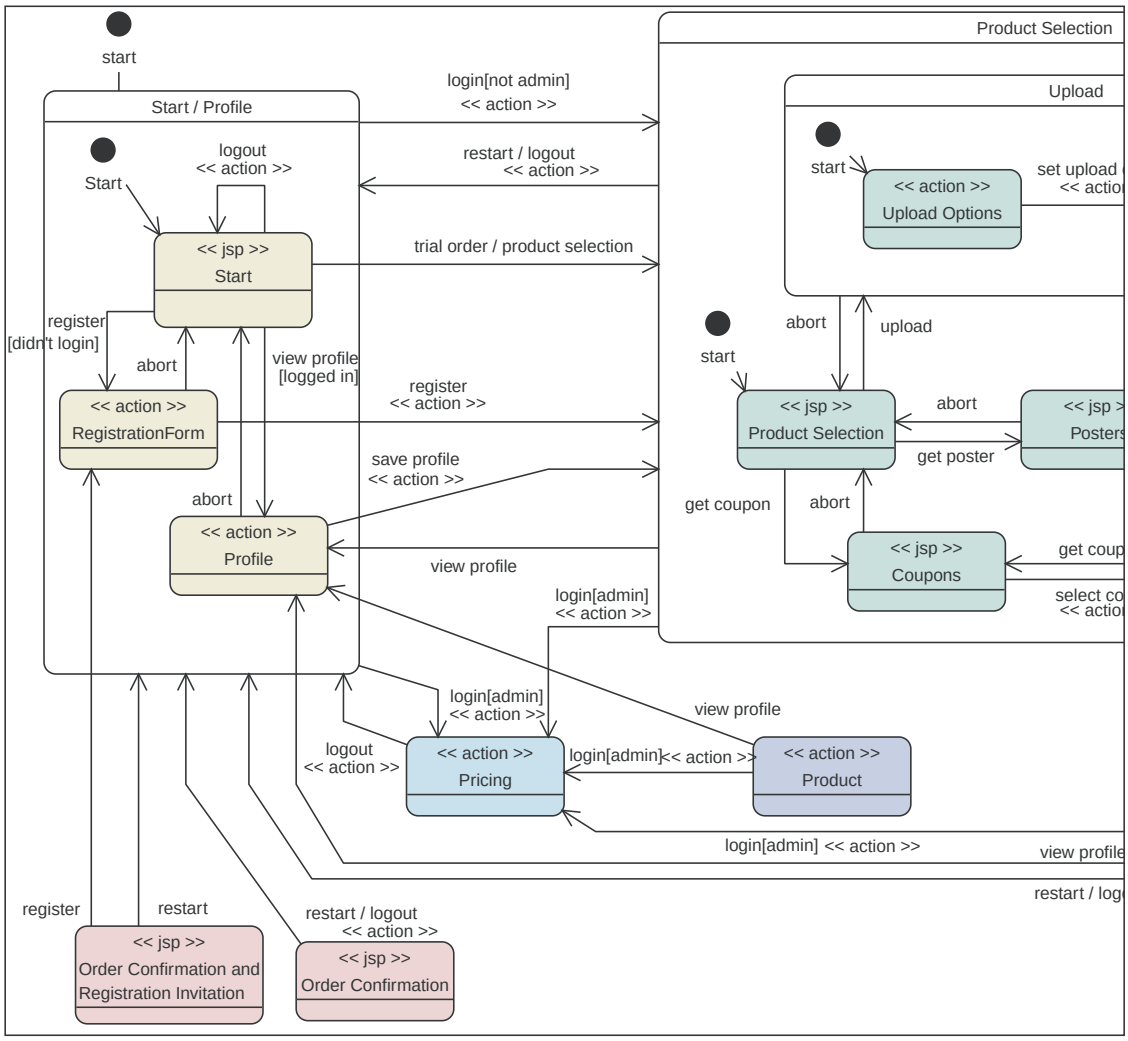

Abbildung 3.5 Detailausschnitt: Start/Profile

Der Ausschnitt zeigt den zusammengesetzten Zustand „Start/Profile“ und seine Beziehung zu anderen Zuständen.

Der Zustand ist Startzustand, d.h. Er wird als erstes aktiviert, wenn der Benutzer die Webanwendung startet. Außerdem wird der Zustand aktiviert, wenn der Benutzer sich ausloggt, oder nochmals von Vorne anfangen möchte (`restart`).

Start

Repräsentiert die Homepage. Hier kann der nicht eingeloggte Benutzer sich einloggen, eine Schnupperbestellung starten, sich registrieren.

Wenn der Benutzer schon eingeloggt ist, kann er zur Produktübersicht wechseln, sein Profil ansehen/ändern und sich ausloggen.

Siehe Abbildung 7.1

Profile

Hier kann der Benutzer sein Profil ändern. Nach der Profiländerung kommt der Benutzer gleich zur Produktübersicht/Auswahl.

Siehe `ProfileForm` (Seite 44)

RegistrationForm

Bei der Registrierung kann der Neukunde den Vorgang abbrechen (`abort`) oder ihn fortfahren (`register`). Nach erfolgreicher Registrierung kommt der Benutzer gleich zur Produktübersicht/Auswahl.

Siehe `RegistrationForm` (Seite 44)

3.2.2 Product Selection

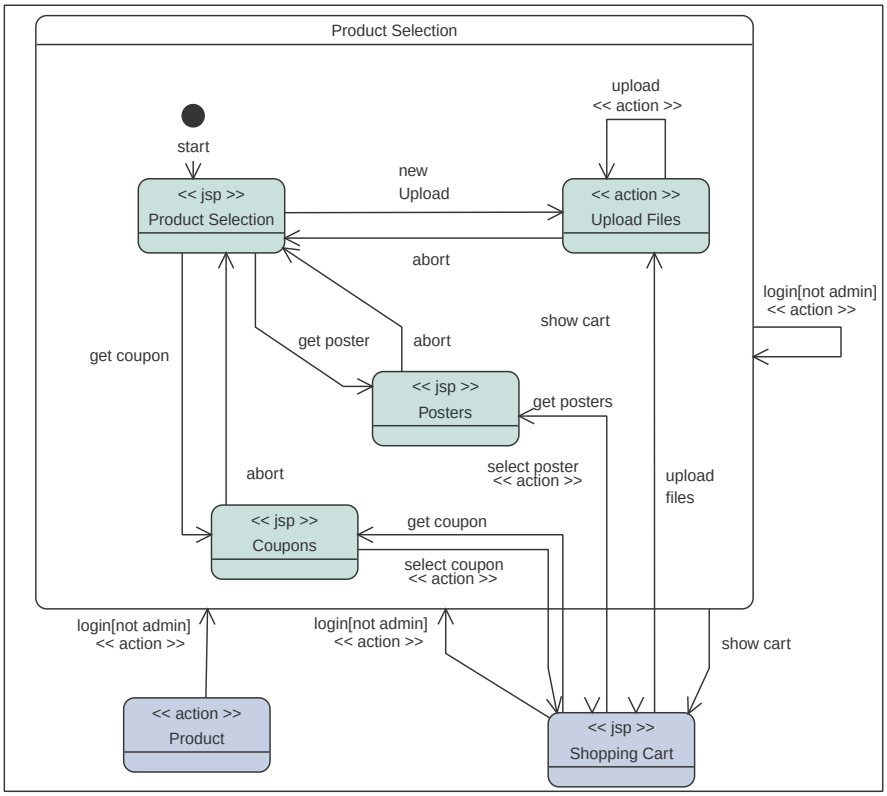

Abbildung 3.6 Detailausschnitt: Product Selection

Der Ausschnitt zeigt den zusammengesetzten Zustand `Product Selection` und seine Beziehung zu anderen Zuständen.

Dieser Zustand fasst alle Produkte ein, die über die Webanwendung bestellbar sind.

Folgende Zustandsübergänge können in allen Unterzuständen ausgeführt werden:

- Wenn der Benutzer noch nicht eingeloggt ist, kann er das mit `login` tun und gelangt dann, wenn er Kunde ist, wieder zum Startzustand von `Product Selection`. Der Administrator kann sich hier auch einloggen und gelangt danach zu seinem Administrationsbereich (`Pricing`)
- Ein eingeloggter Benutzer kann sich auch wieder ausloggen (`logout`), er gelangt dann zum Startzustand
- `show cart` zeigt den Warenkorb an
- `view profile` wechselt zum Kundenprofil (nur gültig, wenn Benutzer sich eingeloggt hat)

Product Selection

Startzustand des zusammengesetzten Zustandes. Hier kann der Benutzer auswählen, welche Produktart er in den Warenkorb legen möchte – Poster (`get poster`), Gutscheine (`get coupon`) oder eigene hochgeladene Bilder (`upload`).

Siehe Abbildung 7.2 und Abbildung 7.12

Upload Files

Wenn der Benutzer auswählt, dass er eigene Bilder in seinen Warenkorb laden möchte, kommt er in diesen Zustand. Hier wählt der Benutzer wie viele Abzüge er jeweils von den Bildern in welchem Format haben möchte und die ersten zehn Dateien, die er hochladen möchte, von seiner lokalen Festplatte aus und kann diese dann auch hochladen. Nach dem Hochladeprozess wird entweder der Warenkorb angezeigt oder nochmals diese Seite, wenn der Benutzer angegeben hat, noch mehr Dateien hochladen zu wollen. Alle Bilder, die der Benutzer bis dahin hochgeladen hat, werden hier am Ende der Seite angezeigt (in umgegehrter Reihenfolge ihres Hochladezeitpunktes), so dass er den Überblick behält, welche Bilder schon hochgeladen wurden. Außerdem kann er den Hochladeprozess abbrechen (`abort`).

Siehe `UploadForm` (Seite 45), Abbildung 7.3 und Abbildung 7.13.

Poster

In diesem Zustand wird dem Benutzer eine Liste mit allen Postern angezeigt, er kann eines auswählen und in den Warenkorb legen. Außerdem kann er abbrechen (`abort`) und kommt dann wieder zu dem Zustand `Product Selection`.

Siehe `SelectProductForm` (Seite 45)

Coupons

In diesem Zustand wird dem Benutzer eine Liste mit allen Gutscheinvorlagen angezeigt, er kann einen auswählen, einen Gutscheinbetrag festlegen und den Gutschein in den Warenkorb legen. Außerdem kann er abbrechen (`abort`) und kommt dann wieder zu dem Zustand `Product Selection`.

Siehe `SelectCouponForm` (Seite 45)

3.2.3 Shopping Cart & Product

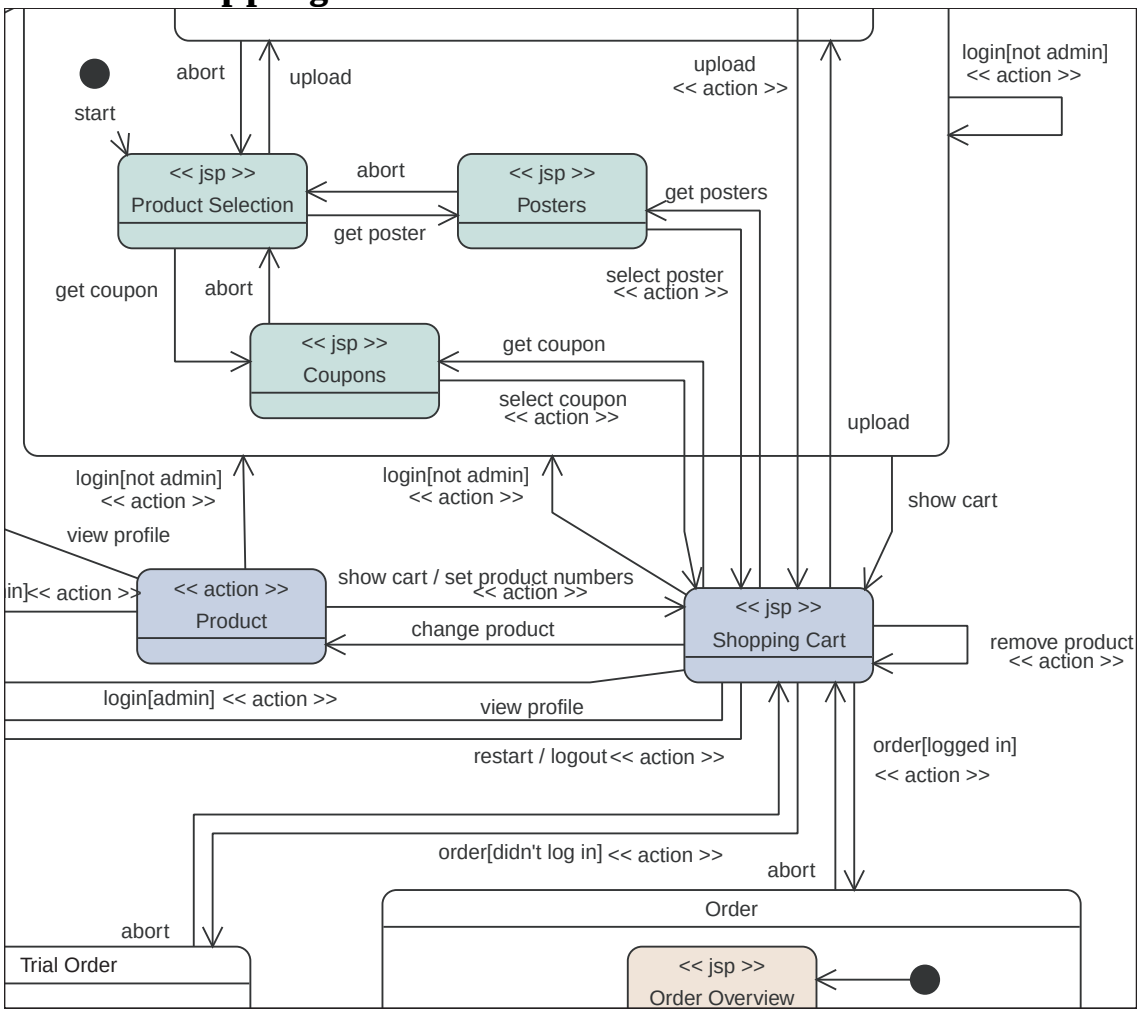

Abbildung 3.7 Detailausschnitt: Shopping Cart & Product

Shopping Cart

(siehe Abbildung 7.4 und Abbildung 7.14)

Von der Warenkorbansicht aus, kann der Kunde weitere Produkte zu seinem Warenkorb hinzufügen (`get coupon`, `get posters`, `upload`), die schon hinzugefügten Produkte wieder entfernen (`remove product`) oder die Anzahl und die Formate für ein im Warenkorb befindlichen Produktes ändern (`change product`).

Außer diesen Möglichkeiten zum Verändern seines Warenkorbs, hat der Benutzer noch folgende Auswahl: Er kann sich einloggen - wobei er daraufhin zur Produktselektion (`Product Selection`) oder zur Administrationsoberfläche (`Pricing`) kommt. Er kann, falls er eingeloggt ist, sein Kundenprofil einsehen und ändern (`view profile`). Oder er kann den Bestellprozess starten (`order`).

In dem Zeitpunkt, in dem der Kunde seinen Bestellwunsch äußert, wird entschieden, ob er eine Schnupperbestellung oder eine Bestellung mit Kundenprofil tätigt. Ist der Kunde zu diesem Zeitpunkt eingeloggt, so wird es eine Bestellung mit Kundenprofil (`Order`), ansonsten tätigt er eine Schnupperbestellung (`Trial Order`).

Product

(siehe Abbildung 7.5 und Abbildung 7.15)

Ist der Benutzer in diesem Zustand, kann er bei einem bestimmten Produkt, welches sich in seinem Warenkorb befindet, die Anzahl und die Formate ändern, in denen er dieses Produkt bestellen möchte. (`set product numbers`)

Außerdem kann er auch ohne Änderungen zu tätigen wieder zum Warenkorb zurückkehren (`show cart`), sich hier entscheiden, sich einzuloggen (`login`) oder sein Kundenprofil einsehen und ändern (`view profile`).

Siehe `SetProductNumbersForm` (Seite 45)

3.2.4 Trial Order & Order

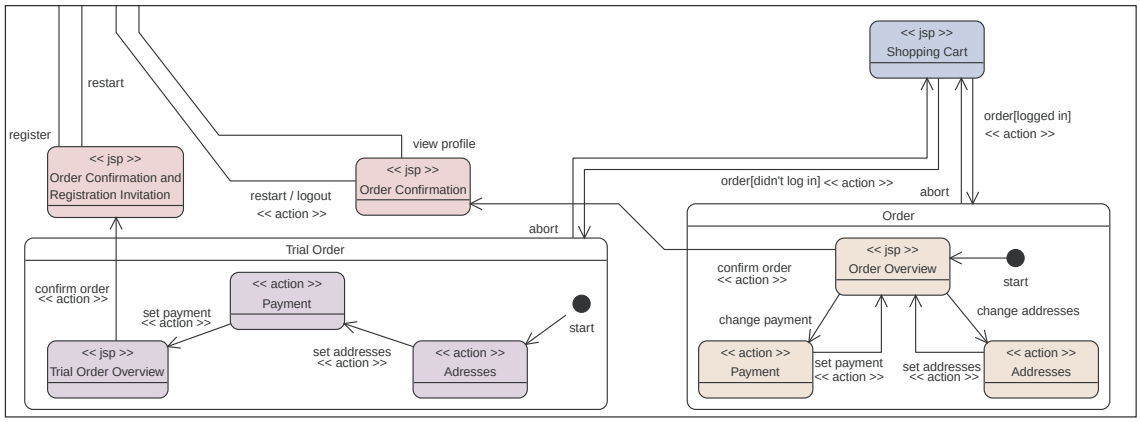

Abbildung 3.8 Detailausschnitt: Trial Order & Order

Siehe `OrderForm` (Seite 45)

Trial Order

Dieser Zustand besteht aus den Unterzuständen `Addresses` (siehe Abbildung 7.6), `Payment` (siehe Abbildung 7.7) und `Order Overview` (siehe Abbildung 7.8). Die Unterzustände werden der Reihe nach durchgegangen, d.h. Erst wird der Benutzer nach der Liefer- und Kundenadresse gefragt, danach wird die Bezahlung geklärt und zum Abschluss bekommt der Kunde eine Übersicht der Bestellung, die er bestätigen muss. Nach der Bestätigung der Bestellung (`confirm order`), wird der bei der Bestellungseingangsbenachrichtigung (`Order Confirmation`,siehe Abbildung 7.9 und Abbildung 7.10) noch zur Registrierung aufgefordert. Je nachdem, ob der Kunde diese Aufforderung befolgt, kommt er entweder auf die Registrierungsseite (`register`, siehe Abbildung 7.11), oder auf die Startseite zurück (`restart`).

In den Unterzuständen `Addresses`, `Payment` und `Order Overview` kann der Benutzer jeweils durch `abort` den Bestellvorgang abbrechen. Er kommt dann wieder auf die Warenkorbseite, wo er seinen gefüllten Warenkorb vorfindet. Seine Bestellung und die von ihm dort eingegebenen Daten werden allerdings gelöscht und er muss sie bei einem erneutem Bestellversuch nochmals eingeben.

Order

Der `Order` Zustand ist dem `Trial Order` Zustand ähnlich, er enthält die gleichen Unterzustände mit der gleichen Semantik. Hier wird allerdings zuerst der Zustand `Order Overview` (siehe Abbildung 7.16) aktiviert, denn ein eingeloggter Kunde hat ja schon Standardwerte für Liefer- und Kundenadresse und seine Standard-Bezahlweise angegeben. Diese Standards kann er gegebenenfalls abändern (`change payment` und `change addresses`) und kommt nach der Änderung wieder direkt zum `Order Overview` zurück. Dem Benutzer werden in den Zuständen `Payment` (siehe Abbildung 7.10) und `Addresses` (siehe Abbildung 7.17) nur jeweils die Möglichkeiten zur Auswahl angegeben, die gerade zulässig sind. So kann er z.B. nicht Barzahlung/EC-Karte auswählen, wenn er nicht auch eine Filiale als Lieferadresse angegeben hat.

Nach der Bestätigung der Bestellung (`confirm order`) erhält der Benutzer eine Bestellungseingangsbenachrichtigung (`Order Confirmation`, siehe Abbildung 7.19), von der aus er wieder auf die Startseite gelangen kann (`restart`/`logout`) oder sein Kundenprofil ändern kann (`view profile`).

Auch hier kann der Benutzer in den Unterzuständen `Addresses`, `Payment` und `Order Overview` jeweils durch `abort` den Bestellvorgang abbrechen.

3.2.5 Administrator Zugang

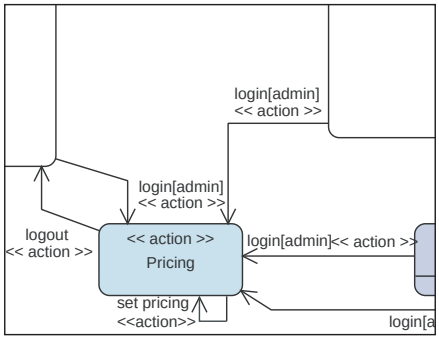

Abbildung 3.9 Detailausschnitt: Pricing

Pricing

Wenn beim Einlogg-Vorgang erkannt wird, dass sich der Administrator eingeloggt hat, wird in den Zustand `Pricing` gewechselt. Dies ist der einzigste Zustand, den der Administrator, wenn er eingeloggt ist, aufrufen kann. Der Administrator hat dort folgende Möglichkeiten: Er kann die Preise und Sonderpreise der Webanwendung verändern (`set pricing`) und/oder sich ausloggen (`logout`).
Siehe `SetPricingForm` (Seite 46)

3.3 ActionForms

3.3.1 Übersicht

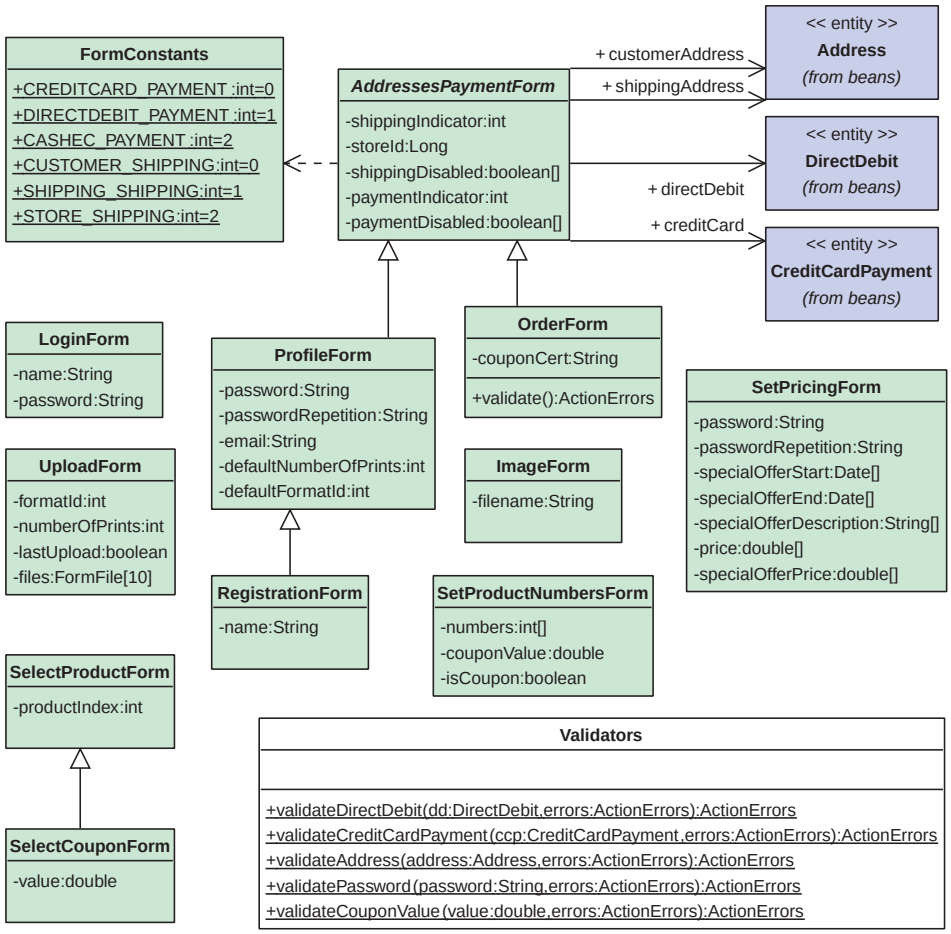

Abbildung 3.10 Übersicht der ActionForms

Die Abbildung 3.10 zeigt ein Klassendiagramm aller ActionForms, die in unserer Webanwendung benötigt werden. Die Validierungsmethode `validate()`, die jede ActionForm hat, wurde hier aus Gründen der Übersichtlichkeit weggelassen. Genauso sind die öffentlichen Zugriffsmethoden für die privaten Eigenschaften ausgeblendet. (Getter und Setter)

3.3.2 Beschreibung der ActionForms

AddressesPaymentForm & FormConstants

`AddressesPaymentForm` ist eine abstrakte ActionForm, die die Gemeinsamkeiten der ActionForms `ProfileForm`, `RegisterForm` und `OrderForm` kapselt.

`FormConstants` ist ein Container für von `AddressPayment` verwendete Konstanten.

Überprüfungen in validate()	*Regulärer Ausdruck*
errors = Validators.validateAddress(`customerAddress`, errors);	
errors = Validators.validateAddress(`shippingAddress`, errors);	

Je nachdem, welche Zahlungsweise der Kunde gewählt hat (erkennbar durch die Interger-Property `paymentIndicator`), wird die entsprechende Validierungsmethode von Validators aufgerufen:
errors = Validators.validateDirectDebit(directDebit, errors);
errors = Validators.validateCreditCardPayment(creditCard, errors);	

`shippingInidcator` und `paymentIndicator` sind 0, 1 oder 2 (die Werte der entsprechenden Konstanten in FormConstants)	
`storeId` ist Null oder eine positive Zahl	

ProfileForm

Diese ActionForm speichert alle Daten, die der Kunde in `Profile.jsp` eingeben/ändern kann.

Überprüfungen in validate()	*Regulärer Ausdruck*
errors = Validators.validatePassword(`password`, errors);	
alternativ kann `password` auch leer gelassen werden, dann wird das Passwort nicht verändert	^$
`password` muss mit `passwordRepetition` übereinstimmen	
`email` muss eine syntaktisch gültige E-Mail Adresse sein	^[^@]+@[a-zA-Z0-9.-]+\.[a-z]{2,4}$
`defaultNumberOfPrints` ist positiv	
`defaultFormatId` ist zwischen 0 und 6	^[0-6]$

RegistrationForm

Abgeleitet von `ProfileForm`. Enthält die Daten der Registrierung eines Kunden. Bis auf den Benutzernamen, den der Kunde nur bei der Registrierung angeben kann, sind diese Daten identisch mit denen der `ProfileForm`.

Überprüfungen in validate()	*Regulärer Ausdruck*
Alle Überprüfungen von `ProfileForm`, wobei hier die `password`-Property *nicht* leer sein darf.	
`name` muss aus mindestens fünf alphanumerischen Zeichen bestehen. Maximal aber nur 20.	^[a-zA-Z0-9-]{5,20}$

LoginForm

Diese ActionForm speichert alle Daten, die der Kunde beim Login-Vorgang eingibt.

Überprüfungen in validate()	*Regulärer Ausdruck*
`name` und `password` müssen den Anforderungen in `RegisterForm` genügen.	

OrderForm

Speichert die Kundeneingaben bei einer Bestellung/Schnupperbestellung.

Überprüfungen in validate()	*Regulärer Ausdruck*
`couponCert` muss entweder leer oder eine zwanzigstellige, BASE64 kodierte Zeichenkette sein.	^$ ^([a-zA-Z0-9+/]{20}){0,1}$

SetProductNumbersForm

In dieser ActionForm wird gespeichert, wie viele Abzüge der Kunde für ein bestimmtes Produkt in den sieben vorhandenen Formaten haben möchte.

Überprüfungen in validate()	*Regulärer Ausdruck*
Alle sieben Einträge in dem Integer-Array `numbers` müssen größer oder gleich Null sein.	

UploadForm

Hier werden die Einstellungen, die der Kunde für den Hochladen-Dialog machen kann, und die Bilder, die er hochlädt, gespeichert.

Überprüfungen in validate()	*Regulärer Ausdruck*
`formatId` muss zwischen 0 und 6 sein	^[0-6]$
`numberOfPrints` und `numberOfFiles` müssen grösser als Null sein.	

SelectProductForm

Speichert die ID des Posters, dass der Kunde in den Warenkorb legen möchte. Sie wird auch bei der Auswahl des Produktes zum Ändern oder zum Entfernen verwendet.

Überprüfungen in validate()	*Regulärer Ausdruck*
`productIndex` muss größer oder gleich Null sein	

SelectCouponForm

Speichert die ID und die Gutscheinhöhe eines Gutscheines, den der Kunde in den Warenkorb legen möchte.

Überprüfungen in validate()	*Regulärer Ausdruck*
`productIndex` muss größer oder gleich Null sein	
`value` muss einen positiven Wert haben	

SetPricingForm

Speichert alle Daten, die der Administrator in der Administrationsoberfläche ändern kann. Die Array-Properties sind jeweils Arrays der Länge sieben für die sieben möglichen Formate der Abzüge.

Überprüfungen in validate()	*Regulärer Ausdruck*
errors = Validators.validatePassword(`password`, errors); *alternativ* kann `password` auch leer gelassen werden, dann wird das Passwort nicht verändert	
`password` muss mit `passwordRepetition` übereinstimmen	
Alle Einträge von `price` und `specialOffer` müssen positiv sein	

ImageForm

Bei verschiedenen Gelegenheiten müssen dem Benutzer Bilder angezeigt werden. So gibt es z.B. in dem Warenkorb Vorschaubilder der Produkte oder auf der Produktdetailseite eine größeres Bild. Alle diese Bilder werden dem Benutzer nicht direkt per URL zugänglich gemacht, sondern sind nur durch einen Aufruf der Action `ImageAction` im Browser des Benutzers anzeigbar. Dies ist nötig, um die Zugriffsrechte des Benutzers zu prüfen, bevor ihm das Bild gezeigt wird. Sonst könnte ein Benutzer durch mutwilliges Verändern der Bild-URL Bilder einsehen, die ein anderer Benutzer gerade hochgeladen hat.

Die `ImageForm` speichert den Dateinamen des Bildes, das durch die `ImageAction` angezeigt werden soll.

Überprüfungen in validate()	*Regulärer Ausdruck*
Es muss sichergestellt sein, dass die Property `filename` einen syntaktisch korrekten Dateinamen enthält. Wir schränken diese Menge aber noch weiter ein, indem wir nur relative Dateinamen der Art `dateiname.jpg` zulassen.	^[a-zA-Z0-9_]+\.jpg$

3.3.3 Validators

Die Klasse Validators enthält folgende statische Methoden. Sie lagern Validierungen, die mehrmals in den ActionForms vorgenommen werden müssen an einen zentralen Ort aus.

+validateDirectDebit(dd: DirectDebit, errors: ActionErrors): ActionErrors

Überprüfungen	*Regulärer Ausdruck*
`dd.accountNumber` muss eine mindestens vierstellige aber maximal zehnstellige Ziffernfolge sein. Führende Nullen sind erlaubt.	^[0-9]{4,10}$

Überprüfungen	*Regulärer Ausdruck*
`dd.accountHolder` ist ein String mit maximal 30 Zeichen	^[a-zA-Z0-9/+ -]{4,30}$
`dd.sortCode` muss aus genau acht Ziffern bestehen	^[0-9]{8}$
`dd.nameOfBank` ist ein String mit maximal 30 Zeichen	^[a-zA-Z0-9/+ -]{4,30}$

+validateCreditCardPayment(ccp: CreditCardPayment, errors: ActionErrors): ActionErrors

Überprüfungen	*Regulärer Ausdruck*
`ccp.number`, `ccp.expirationDate` und `ccp.type` werden durch den Kreditkarten-Plausibilitätsalgorithmus überprüft. Siehe Seite 18.	
`ccp.holder` ist ein String mit maximal 30 Zeichen	^[a-zA-Z0-9/+ -]{4,30}$

+validateAddress(address: Address, errors: ActionErrors): ActionErrors

Überprüfungen	*Regulärer Ausdruck*
`address.name` ist ein String mit maximal 60 Zeichen	^[a-zA-Z0-9/+ -]{4,60}$
`address.street` ist ein String mit maximal 60 Zeichen	^[a-zA-Z0-9/+ -]{4,60}$
`address.city` ist ein String mit maximal 60 Zeichen	^[a-zA-Z0-9/+ -]{4,60}$
`address.postCode` ist eine fünfstellige Ziffernfolge.	^[0-9]{5}$

+validatePassword(password: String, errors: ActionErrors): ActionErrors

Überprüfungen	*Regulärer Ausdruck*
`password` besteht aus mindestens vier und maximal 20 alphanumerischen Zeichen und ausgewählten Sonderzeichen	^[a-zA-Z0-9/+-]{4,20}$

4 Die Controller-Komponente

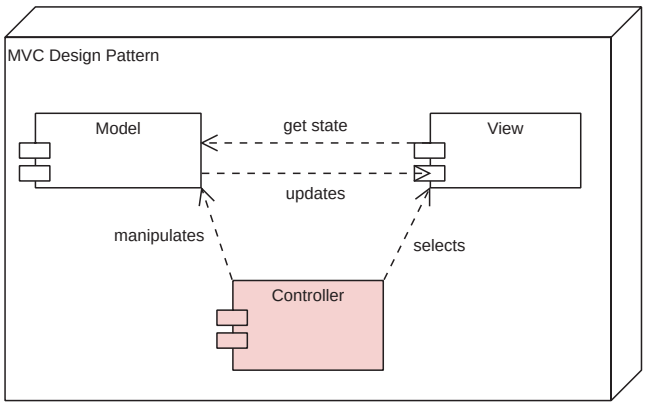

Abbildung 4.1 Einordnung in das MVC Entwurfsmuster

In diesem Kapitel wollen wir die Controller-Komponente genauer unter die Lupe nehmen. Die verwendete Controller-Komponente ist im Wesentlichen die von Struts vorgegebene, die von uns um die `SupportAction` erweitert wird. Wenn man die umfangreichen Klassen zur Konfiguration von Struts außer Acht lässt, wird die Hauptarbeit unter zwei Klassen aufgeteilt. Zum einen das von Struts vorgegebene `ActionServlet`, das für das Weiterleiten der Anfragen des Benutzers an die richtigen Modell-Teile zuständig ist, ebenso wie für die Extraktion und Aufbereitung der Eingabedaten. Zum Anderen die `SupportAction`, die dafür sorgt, dass `Action`s nur aufgerufen werden können, wenn für die jeweilige `Action` bestimmte Vorbedingungen erfüllt sind. Und sie sorgt nach der Aktivierung der Geschäftslogik dafür, dass die Kontrolle an die zuständige JSP weitergeleitet wird.

Um die Reihenfolge der Ausführung der `Action`s zu erzwingen, werden die Struts Workflow Extensions eingesetzt.

4.1 Die SupportAction

Die `SupportAction` ist aus der Sicht von Struts eine ganz normale Action, ist aber abstrakte Erbgrundlage für die von uns verwendeten `Action`s, die die zu erledigenden Manipulationen auf den Geschäftsobjekten implementieren, und verhilft somit zu einer leichteren Trennung zwischen Model und Controller. Dies erreicht sie dadurch, dass sie die Controller-Funktionalitäten implementiert und alles, was zur Geschäftslogik gehört, den erbenden Klassen überlässt. Desweiteren bietet sie den erbenden `Action`s einige Implementierungen häufig verwendeter Methoden an. In Abschnitt 4.2 wird die Einbettung der `SupportAction` zwischen `ActionServlet` und dem Geschäftlogikteil vorgestellt.

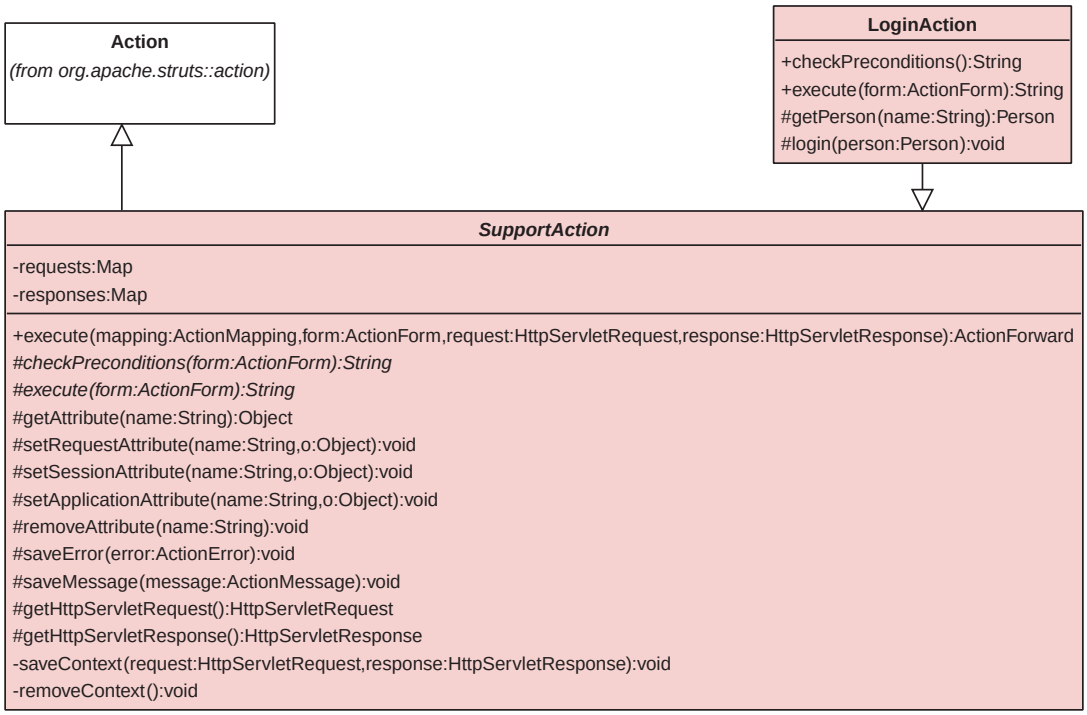

Abbildung 4.2 Die SupportAction

4.1.1 Methoden der SupportAction

Name	***Funktion***
#checkPreconditions(form:ActionForm):String	Überprüft, ob das Ausführen dieser `Action` im Moment zulässig ist und gibt den Namen eines `Forward`s zurück falls nicht.
#execute(form:ActionForm):String	Startet die Geschäftslogik der `Action` und gibt nach Beendigung den Namen eines `Forward`s zurück.
#getAttribute(name:String):Object	Sucht in allen Kontexten nach dem Attribut mit dem gegebenen Namen.
#setRequestAttribute(name:String,o:Object)	Legt das gegebene Attribut im Requestkontext ab.
#setSessionAttribute(name:String,o:Object)	Legt das gegebene Attribut im Sessionkontext ab.
#setApplicationAttribute(name:String,o:Object)	Legt das gegebene Attribut im Applikationskontext ab.

Name	***Funktion***
#removeAttribute(name:String)	Löscht das Attribut mit dem gegebenen Namen aus allen Kontexten, in denen es vorhanden ist.
#saveError(error:ActionError)	Speichert eine Fehlernachricht zur späteren Anzeige.
#saveMessage(message:ActionMessage)	Speichert eine Textnachricht zur späteren Anzeige.
#getHttpServletRequest():HttpServletRequest	Gibt den zum momentanen Aufruf gehörigen `ServletRequest` zurück.
#getHttpServletResponse():HttpServletResponse	Gibt die zum momentanen Aufruf gehörige `ServletResponse` zurück.
-saveContext(request:HttpServletRequest, response:HttpServletResponse)	Speichert den `Request` und die `Response` zur späteren Verwendung
-removeContext()	Entfernt den gespeicherten Kontext.

4.2 Interaktion mit der View-Komponente

4.2.1 Einkommendes Ereignis mit gültigen Eingabedaten

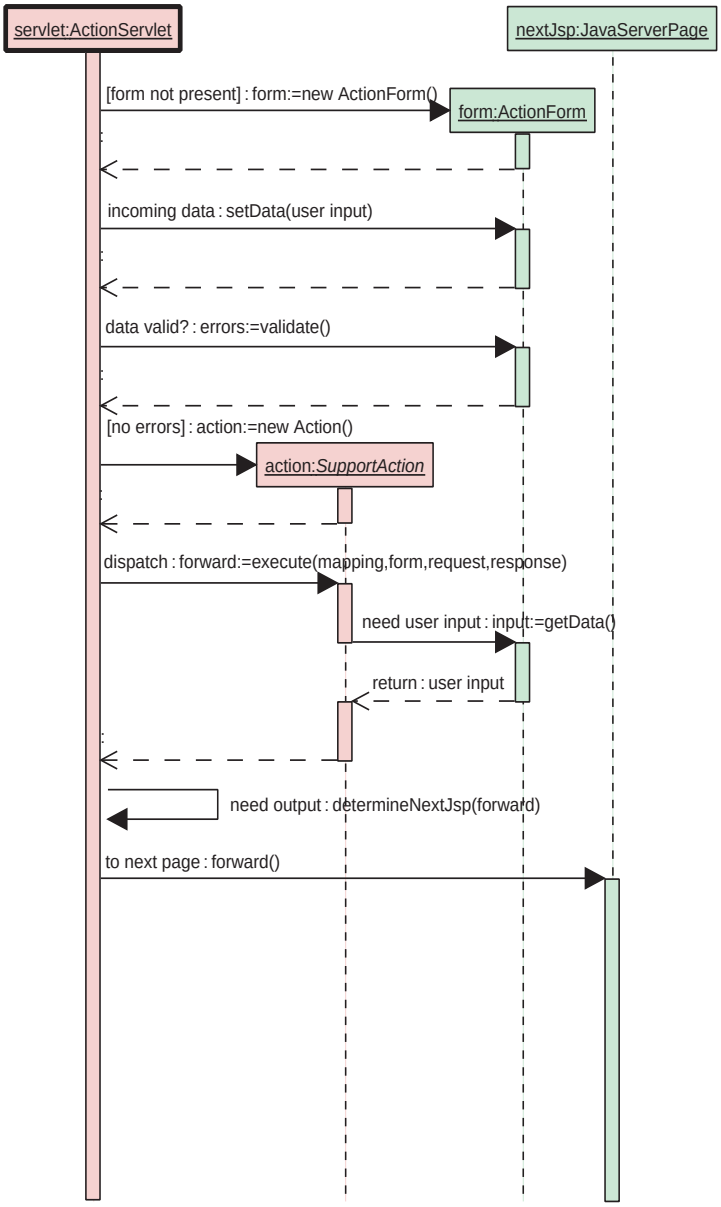

Abbildung 4.3 Einkommendes Ereignis mit gültigen Eingabedaten

- Zunächst erzeugt das `ActionServlet` eine `FormBean`, die dann mit den Daten aus der Anfrage des Browsers gefüllt wird.

- Danach wird die `FormBean` aufgefordert ihre Daten auf Gültigkeit zu überprüfen.
- Wurden dabei keine Fehler festgestellt wird die zuständige `SupportAction`-Instanz ermittelt und gegebenenfalls erzeugt.
- Diese sorgt anschließend für die Ausführung der nötigen Geschäftslogikfunktionen, wobei ein Zugriff auf die Anfragedaten über die FormBean besteht.
- Zum Abschluss wird von der `SupportAction` zurückgegeben, mit welcher JSP es weitergehen soll. Diese wird dann vom `ActionServlet` aktiviert.

4.2.2 Ereignis mit ungültigen Eingabedaten

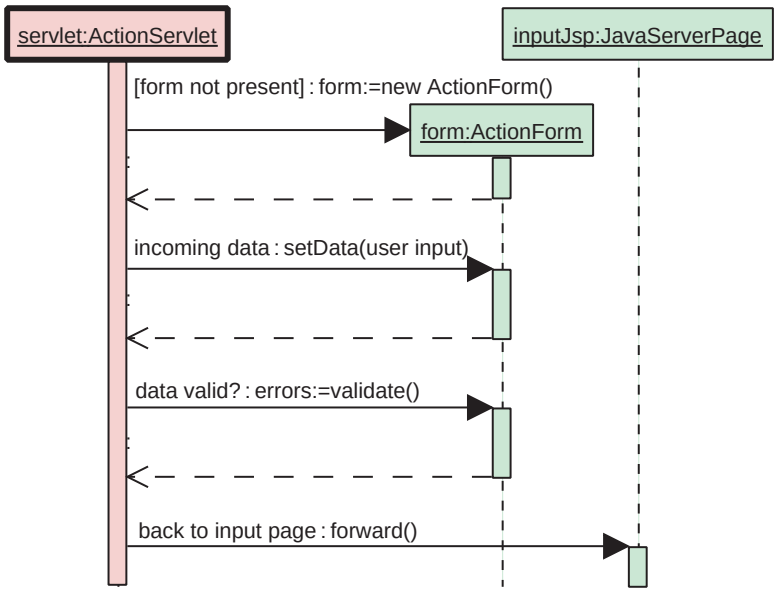

Abbildung 4.4 Ereignis mit ungültigen Eingabedaten

- Die Anfänglichen Schritte sind die selben wie beim Fall der gültigen Eingabedaten.
- Der Aufruf zur Überprüfung der Gültigkeit der Eingaben an die `FormBean` liefert jedoch Fehler zurück.
- Damit weiß das `ActionServlet`, dass die Anfrage nicht bearbeitet werden kann und gibt den Kontrollfluss sofort an die Eingabeseite zurück, damit der Benutzer seine Eingabe korrigieren kann.

4.3 Interaktion mit der Model-Komponente

4.3.1 Aufruf mit erfüllten Vorbedingungen

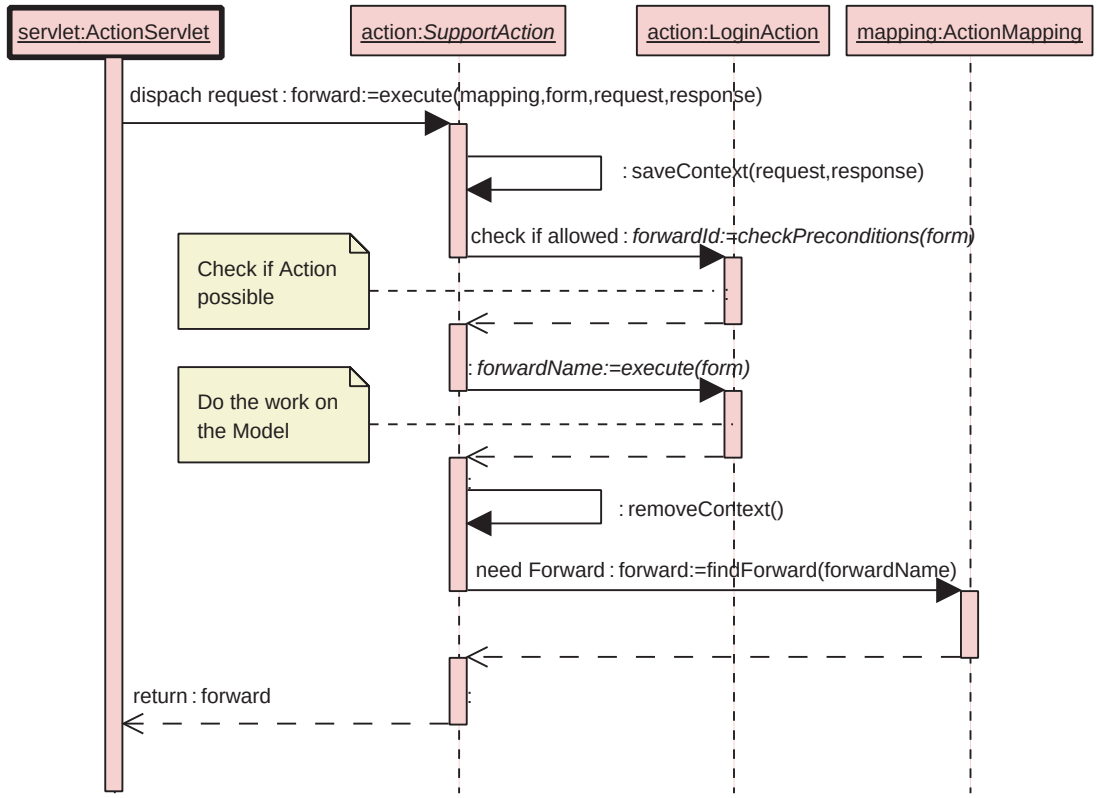

Abbildung 4.5 Aufruf mit erfüllten Vorbedingungen

- Nachdem wie oben beschrieben die `SupportAction` aktiviert wird muss sie zunächst prüfen, ob die Vorbedingungen erfüllt sind. Dazu ruft sie auf sich selbst die abstrakte Methode `checkPreconditions()` auf. Die oben dargestellten Objekte „action“ sind also in Realität nur ein einziges Objekt der abstrakten Klasse `SupportAction` und der abgeleiteten Klasse `LoginAction`, wobei die Aufgaben der Oberklasse `SupportAction` eindeutig zur den Kernfunktionen der Controller-Komponente zugerechnet werden. Die eigentliche Aktion wird in der `LoginAction` ausgeführt so dass sie zur Geschäftslogik zählt.
- Sind die Vorbedingungen erfüllt, so liefert der Aufruf keinen Namen zurück. Nachdem sich herausgestellt hat, dass dies der Fall ist, wird die von der `LoginAction` bereitgestellte Funktionalität aktiviert.
- Danach wird je nach Ausgang der Geschäftsfunktion über ein `ActionMapping` die zuständige JSP ermittelt und das Ergebnis zurückgeliefert.

4.3.2 Aufruf mit unerfüllten Vorbedingungen

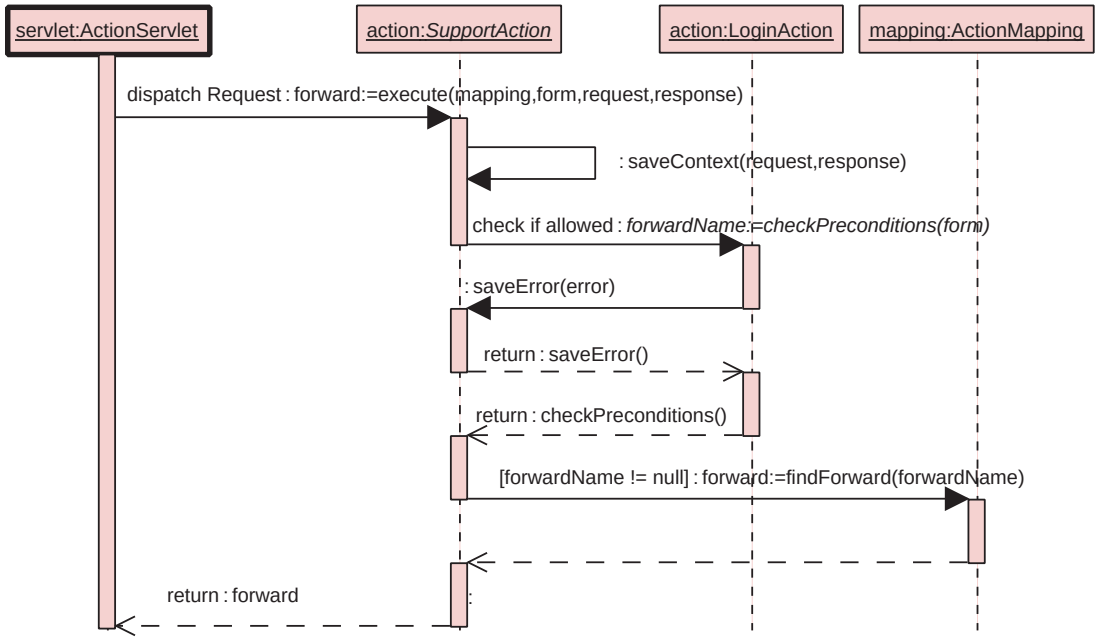

Abbildung 4.6 Aufruf mit unerfüllten Vorbedingungen

- Bei dieser Variante ergibt sich bei der Prüfung der Vorbedingungen ein Fehler, welcher in der aktuellen Fehlerliste gespeichert wird. Außerdem gibt die Methode den Namen einer Weiterleitung zurück, die für die Behandlung des Fehlers zuständig ist.
- Die aufgetretenen Fehler werden dann endgültig gespeichert, damit sie dem Benutzer später angezeigt werden können.
- Zum Schluss wird eine JSP für die Darstellung der Fehler gesucht und eine Verbindung zu dieser zurückgegeben.

4.4 Struts-Plugins & Servlet Container Listener

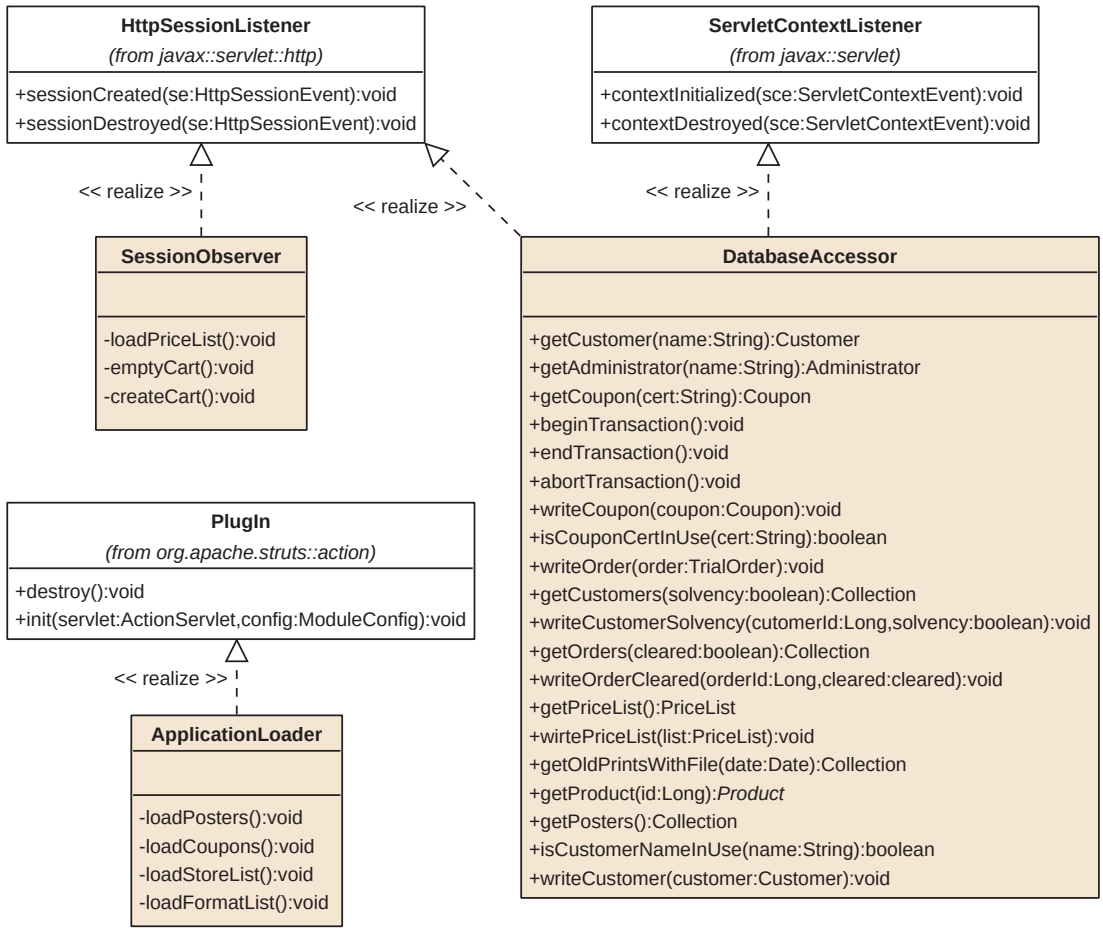

Abbildung 4.7 Die Struts Plugins & Servlet Container Listener unserer Webanwendung

4.4.1 DatabaseAccessor

Die Datenzugriffsfunktionen des `DatabaseAccessor`s betrachten wir in Kapitel 2.8 näher. Hier gehen wir auf die Einbindung des `DatabaseAccessor`s in die Struts Umgebung ein.

Der `DatabaseAccessor` implementiert die beiden Listener `HttpSessionListener` und `ServletContextListener`, um die Hibernate-Konfiguration zu laden, die Datenbankverbindung zu initialisieren und sie in den Application-Context zu legen.

4.4.2 SessionObserver

Der `SessionObserver` ist ein `HttpSessionListener`, der für jede HTTP-Session, die von dem Servlet Container erstellt wird, die nötigen Initialisierungs- und Finalisierungsarbeiten verrichtet. Jede neue Session muss einen neuen Warenkorb erhalten und eine

lokale Kopie der derzeit gültigen Preise. Bei dem Löschen einer Session muss der darin enthaltene Warenkorb wieder kontrolliert gelöscht werden, denn die in dem Warenkorb enthaltenen Bilder, die der Kunde nicht bestellt hat, müssen mit gelöscht werden.

4.4.3 ApplicationLoader

Der `ApplicationLoader` wird beim Programmstart geladen. Hier werden die Objekte aus der Datenbank gelesen, die für alle HTTP-Session Gültigkeit besitzen: Die Poster, die Gutscheine, eine Liste mit Filialen und eine Liste aller Formate.

5 Abbildung der Geschäftsprozesse

5.1 Gemeinsamkeiten Bestellung und Schnupperbestellung

5.1.1 Bilder hochladen

(a) Ablaufdiagramm

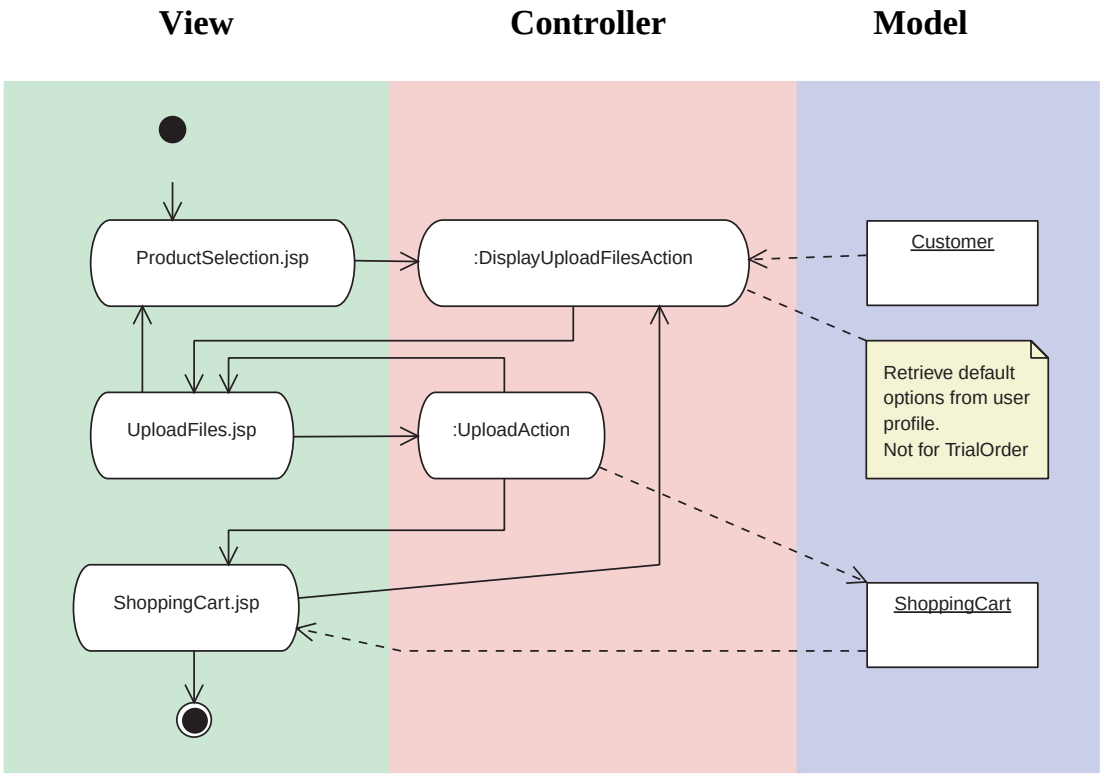

Abbildung 5.1 Bilder hochladen

(b) Beschreibung

- Wählt der Kunde „Bilder hochladen“, so muss er auswählen wieviele Drucke er pro Bild haben möchte und welches Druckformat die Bilder haben sollen. Diese Einstellungen tätigt er ein in der `UploadFiles.jsp`, welche durch die `DisplayUploadFilesAction` mit den Standardwerten der Applikation oder den Standardwerten aus dem Benutzerprofil, falls der Kunde eingeloggt ist, initialisiert wird.
- Ebenfalls auf dieser JSP sucht er bis zu zehn jpg-Dateien aus, die er hochladen möchte. Er kann dann noch wählen, ob er außer diesen danach noch weitere Dateien hochladen möchte, oder ob er dann mit dem Hochladen fertig ist.
- Nach der Auswahl der entsprechenden Dateien durch den Kunden nimmt die `UploadAction` diese entgegen und sorgt für die Eintragung der zugehörigen `ProductGroups` im `ShoppingCart`, die mit den ausgewählten Optionen versehen werden. Darüber hinaus werden die Dateien gespeichert und die nötigen Vorschaubilder erzeugt. (4.1 /F10/ 1 bzw 4.1 /F20/ 1 im Pflichtenheft)
- Je nach vorheriger Wahl des Kunden geht es nun zurück zur `UploadFiles.jsp` oder weiter mit dem Einkaufswagen.

(c) Beispiel: UploadAction

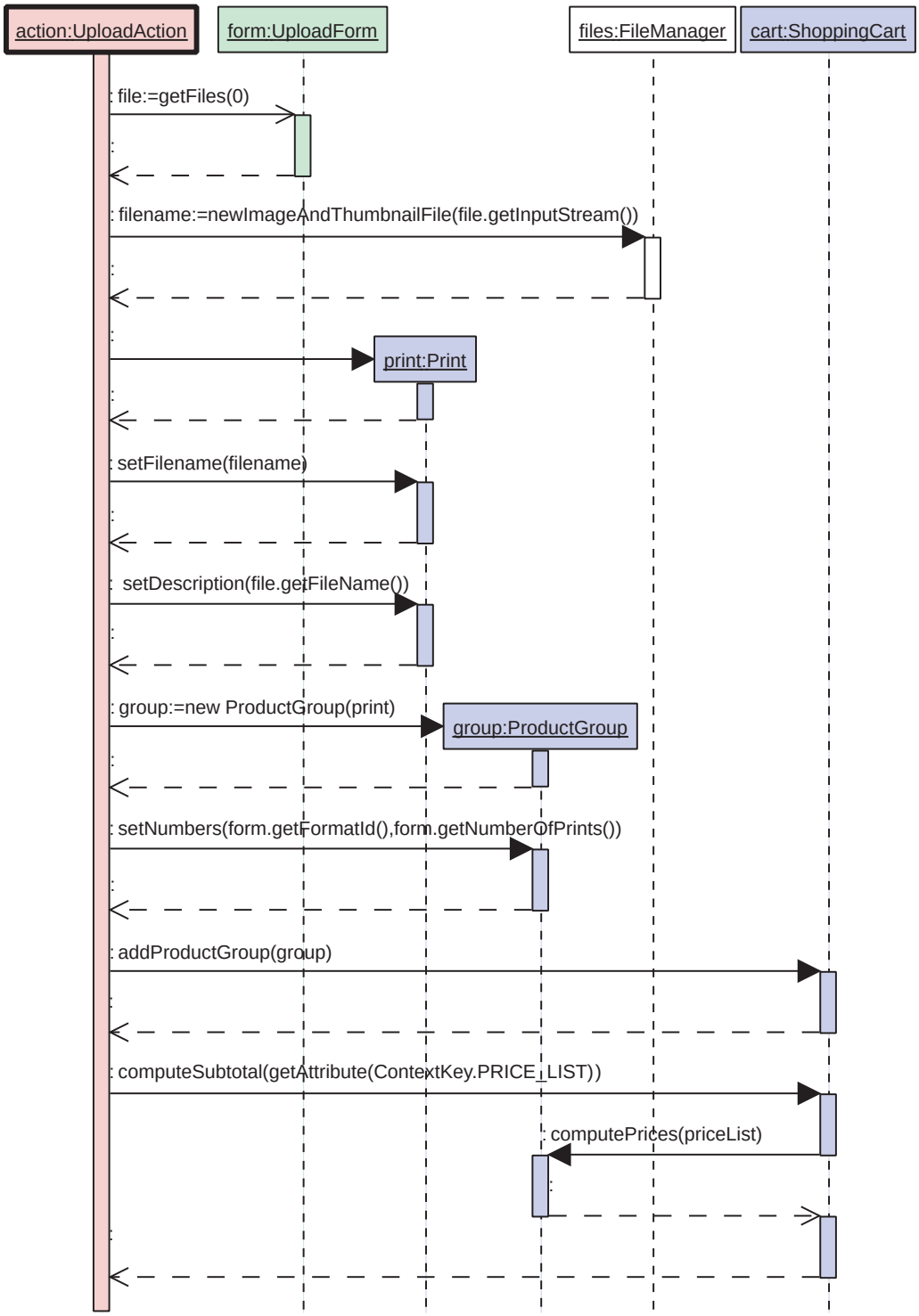

Abbildung 5.2 UploadAction Beispiel: Hochladen eines Bildes

(d) Beteiligte Actions

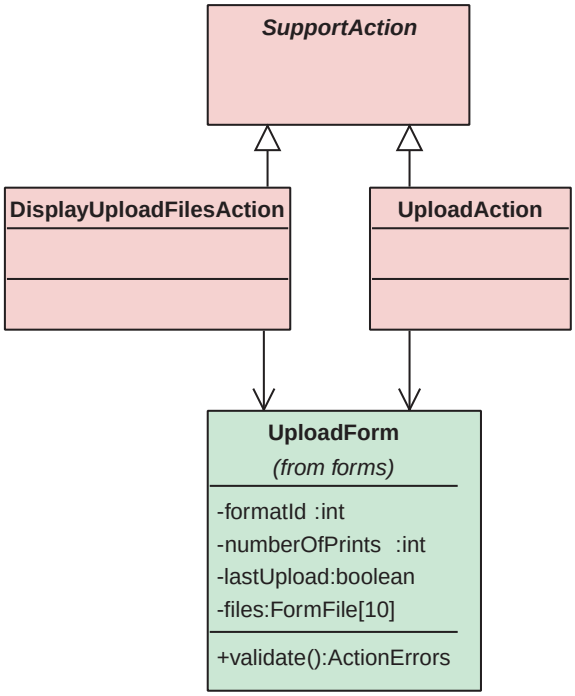

Abbildung 5.3 Beteiligte Actions

5.1.2 Poster auswählen

(a) Ablaufdiagramm

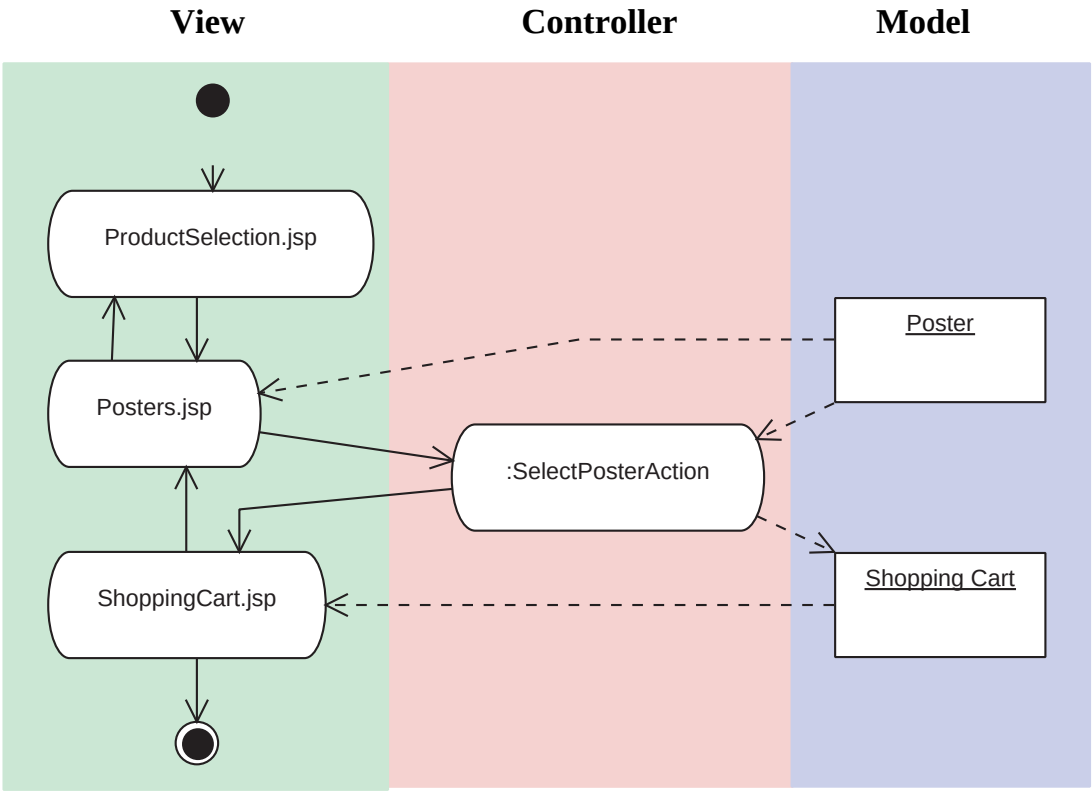

Abbildung 5.4 Poster auswählen

(b) Beschreibung

- Der Kunde entscheidet sich ein Poster zu bestellen. Dann bekommt er mit `Posters.jsp` eine Anzeige aller verfügbarer Poster, von denen er sich eines auswählt.
- Die `SelectPosterAction` sorgt dann dafür, dass das Poster in den Warenkorb gelegt wird. (4.1 /F10/ 1 bzw 4.1 /F20/ 1 im Pflichtenheft)

(c) Beteiligte Actions

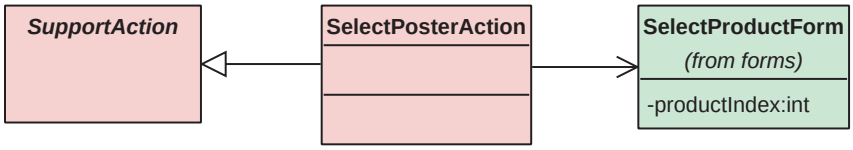

Abbildung 5.5 Beteiligte Actions

5.1.3 Gutschein auswählen

(a) Ablaufdiagramm

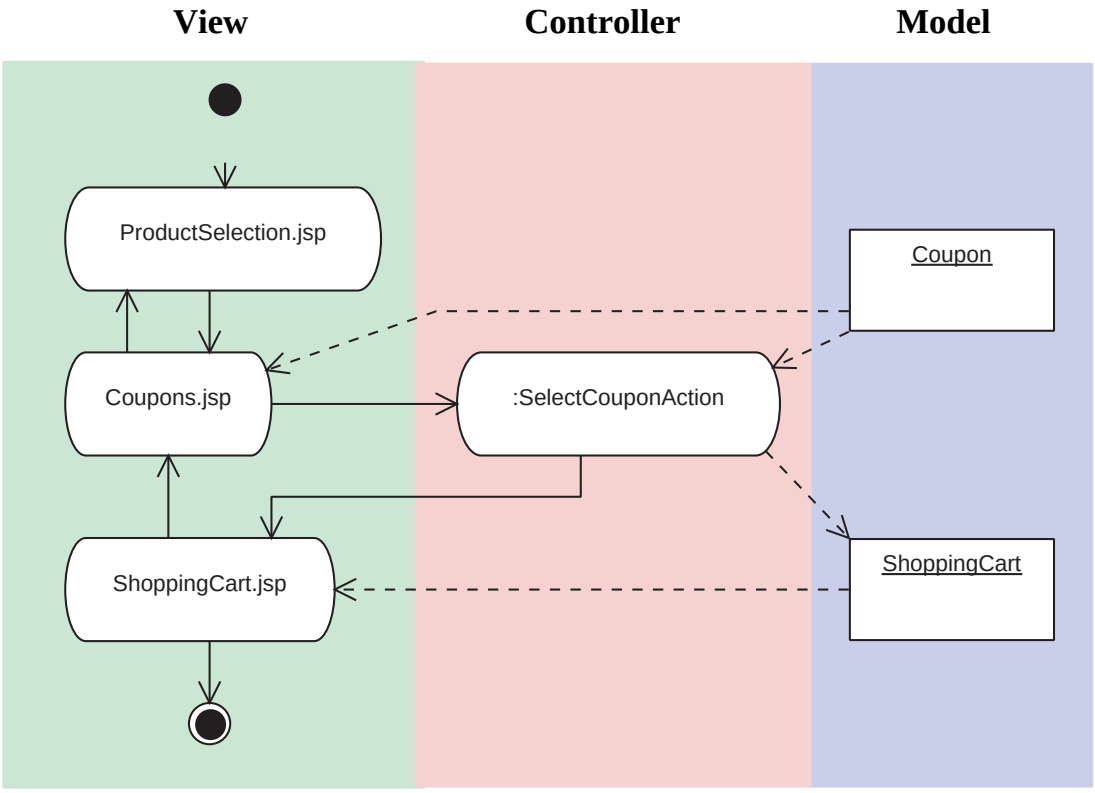

Abbildung 5.6 Gutschein auswählen

(b) Beschreibung

- Der Kunde entscheidet sich einen Gutschein zu bestellen. Dann bekommt er mit `Coupons.jsp` eine Anzeige aller verfügbarer Gutscheinvorlagen, von denen er sich eine auswählt. Zusätzlich gibt er noch den Wert des Gutscheins an.
- Die `SelectCouponAction` sorgt dann dafür, dass eine Kopie des Gutscheins mit dem entsprechenden Wert versehen und in den Warenkorb gelegt wird. (4.1 /F10/ 1 bzw 4.1 /F20/ 1 im Pflichtenheft)

(c) Beteiligte Actions

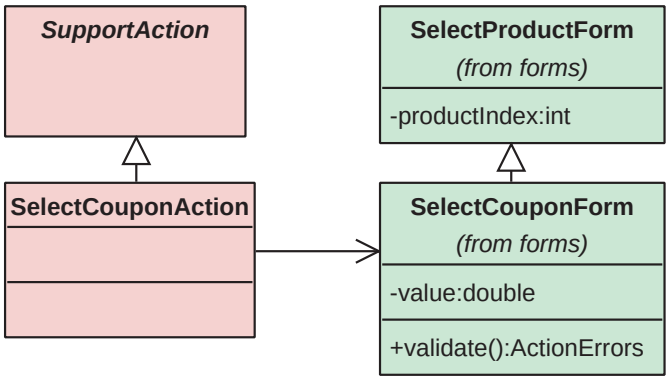

Abbildung 5.7 Beteiligte Actions

5.1.4 Produkt aus dem Warenkorb entfernen

(a) Ablaufdiagramm

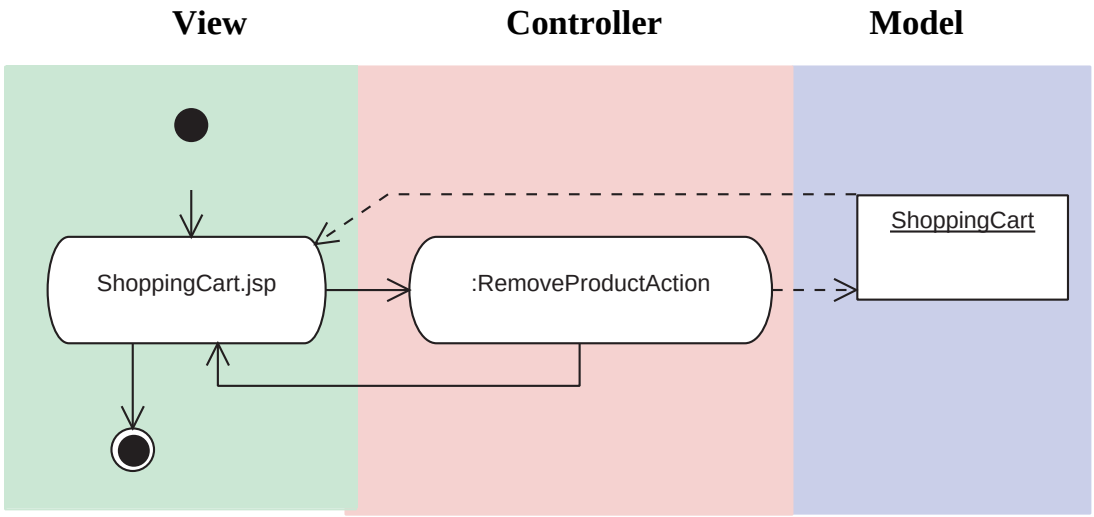

Abbildung 5.8 Produkt aus dem Warenkorb entfernen

(b) Beschreibung

- Der Kunde möchte ein Produkt aus seinem Warenkorb entfernen. Dazu aktiviert er über einen Link die `RemoveProductAction`, die die entsprechende `ProductGroup` aus dem `ShoppingCart` löscht. Dabei wird auch das hochgeladene Bild gelöscht, falls es sich nicht um einen Gutschein oder ein Poster handelt.

(c) Beteiligte Actions

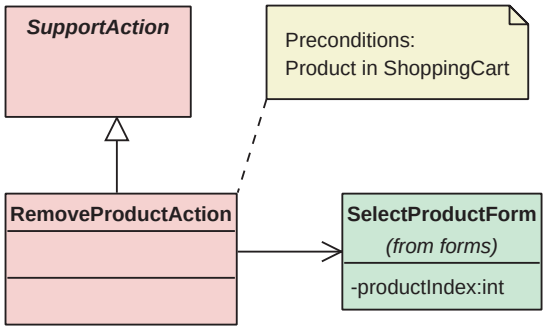

Abbildung 5.9 Beteiligte Actions

5.1.5 Druckanzahl / -Format eines Produktes ändern

(a) Ablaufdiagramm

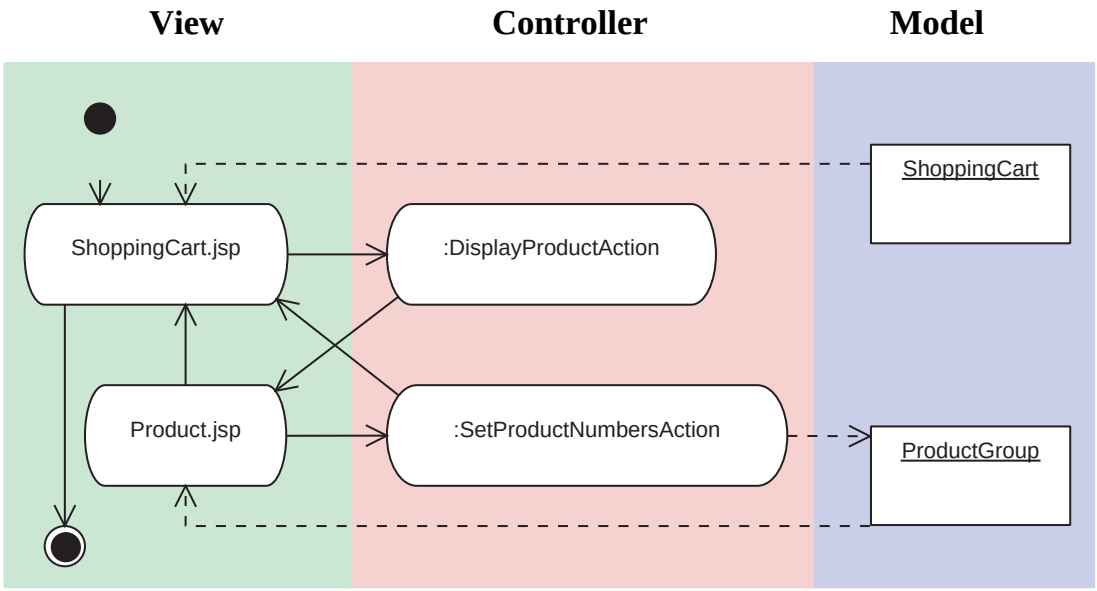

Abbildung 5.10 Druckanzahl / -Format eines Produktes ändern

(b) Beschreibung

- Der Kunde möchte die Anzahl der Drucke der verschiedenen Druckformate für ein einzelnes Produkt oder den Wert der Gutscheine ändern.
- Um dies zu erreichen, wird ihm eine spezielle Seite, die `Product.jsp`, gezeigt, auf der er die Einstellungen vornehmen kann. Ihre Anzeige wird durch die `DisplayProductAction` mit den aktuellen Werten initialisiert.

- Nach der Änderung der Anzahlen der Formate oder des Gutscheinwertes kann er diese speichern. Die hierfür notwendigen Änderungen an der `ProductGroup` werden von der `SetProductNumbersAction` vorgenommen. (4.1 /F10/ 3 bzw 4.1 /F20/ 3 im Pflichtenheft)

(c) Beteiligte Actions

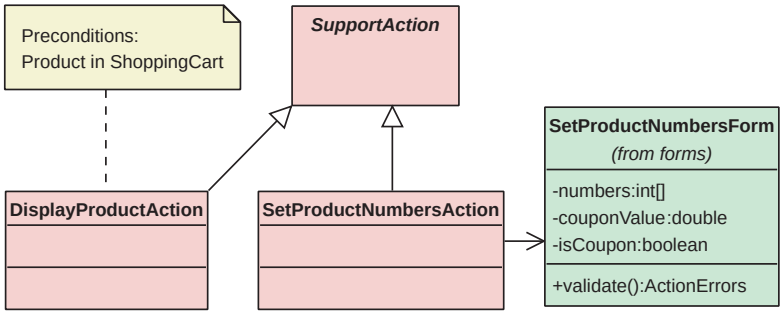

Abbildung 5.11 Beteiligte Actions

5.2 Schnupperbestellung /F10/

(a) Ablaufdiagramm

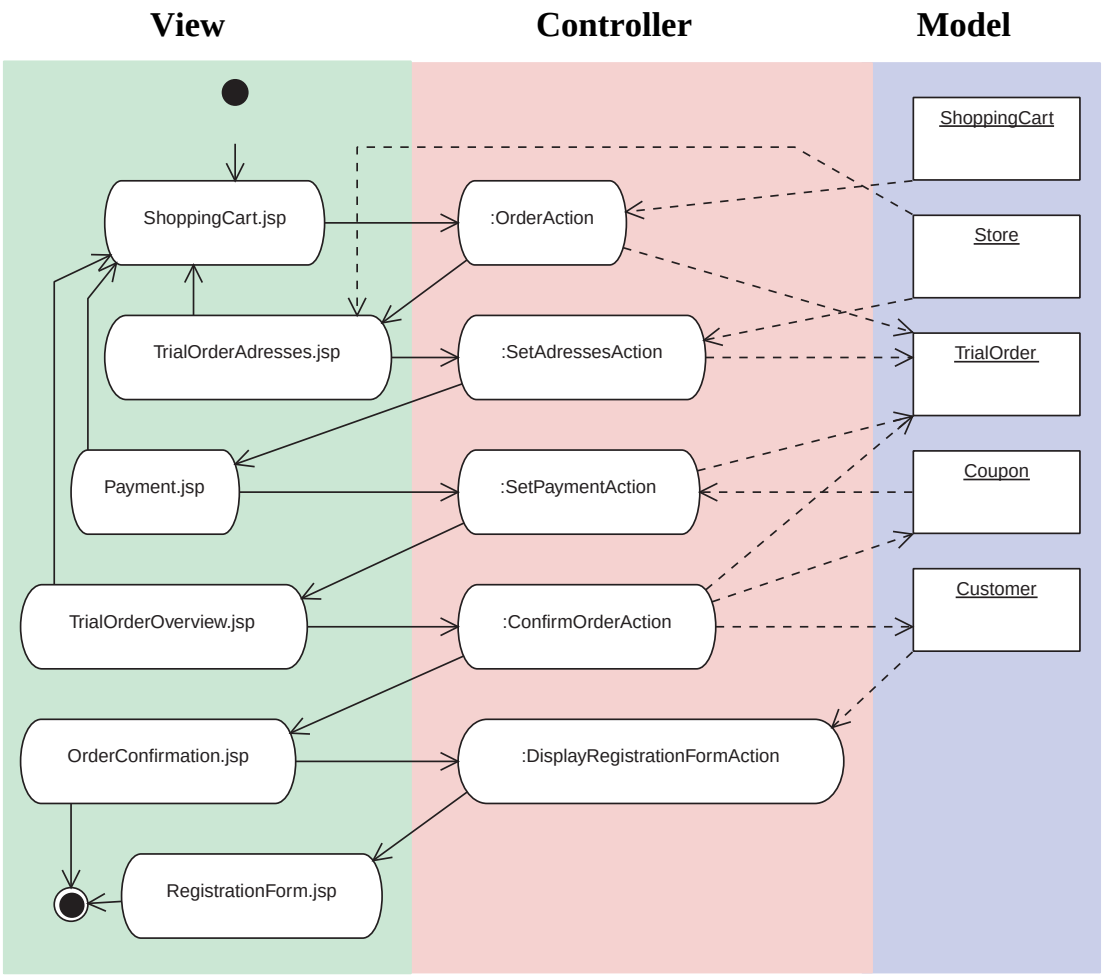

Abbildung 5.12 Schupperbestellung /F10/

(b) Beschreibung

- Die Ausprägung der Schnupperbestellung im Eigentlichen wird erst nach dem Auswählen von Waren deutlich. D.h. Wir betrachten hier den Abschnitt in dem die Produkte, die sich bereits im Warenkorb befinden bestellt werden sollen. Wir sind also zunächst bei der Ansicht des Warenkorbinhaltes. (4.1 /F10/ 2 im Pflichtenheft)
- Initial wird dann die `OrderAction` angestoßen, die ein neues `TrialOrder`-Objekt in der Session erzeugt und die Produkte aus dem Warenkorb dorthin überträgt. (Beispielablauf in Abbildung 5.13)

- Im nächsten Schritt bekommt der Kunde leere Adressfelder präsentiert, in die er seine Adresse und gegebenenfalls eine separate Lieferadresse eingeben kann. Auch die Filiale, in die er geliefert bekommen möchte, kann auf `TrialOrderAddresses.jsp` ausgewählt werden. (4.1 /F10/ 4 im Pflichtenheft)
- Die `SetAddressesAction` sorgt dafür, dass die eingegebenen Adressen und Lieferoptionen korrekt in das `TrialOrder`-Objekt übernommen werden
- Danach muss sich der Kunde für eine Zahlungsart entscheiden und die dafür nötigen Angaben machen. Er erhält mit `Payment.jsp` die entsprechenden Eingabeformulare. Auch den Schlüssel eines Gutscheins kann der Kunde dort eingeben. Dabei werden nur die für die gewählte Lieferart gültigen Möglichkeiten angeboten. (4.1 /F10/ 5 im Pflichtenheft)
- Die `SetPaymentAction` überprüft nochmals ob die gewählten Eingaben zulässig sind und ob der Gutschein, falls eingegeben, gültig ist und speichert die Informationen im `TrialOrder`-Objekt. (Beispielablauf in Abbildung 5.14)
- Jetzt bekommt der Kunde nochmals eine Auftragsübersicht zur Bestätigung. (4.1 / F10/ 6 im Pflichtenheft) Dazu dient die `TrialOrderOverview.jsp`.
- Mit der Bestätigung wird die Bestellung getätigt. Dies wird von der `ConfirmOrderAction` erledigt. Darin wird der Gutschein zur Zahlung nochmals überprüft, die bestellten Gutscheine generiert und die Bestellung in der Datenbank persistent gespeichert. (4.1 /F10/ 7 im Pflichtenheft)
- Ist das erfolgreich verlaufen erhält der Kunde eine Auftragsbestätigung und Aufforderung zur Registrierung durch die `OrderConfirmation.jsp`. (4.1 /F10/ 8 im Pflichtenheft)
- Von dort aus kann der Kunde sich direkt zur Registrierung begeben, um von den vorher eingegebenen Adress- und Zahlungsdaten zu profitieren. Dies erfolgt über die `DisplayRegistrationFormAction`, welche für die Vorabfüllung der nötigen Formulare sorgt. (4.1 /F10/ 9 im Pflichtenheft)

(c) Beispiel: OrderAction

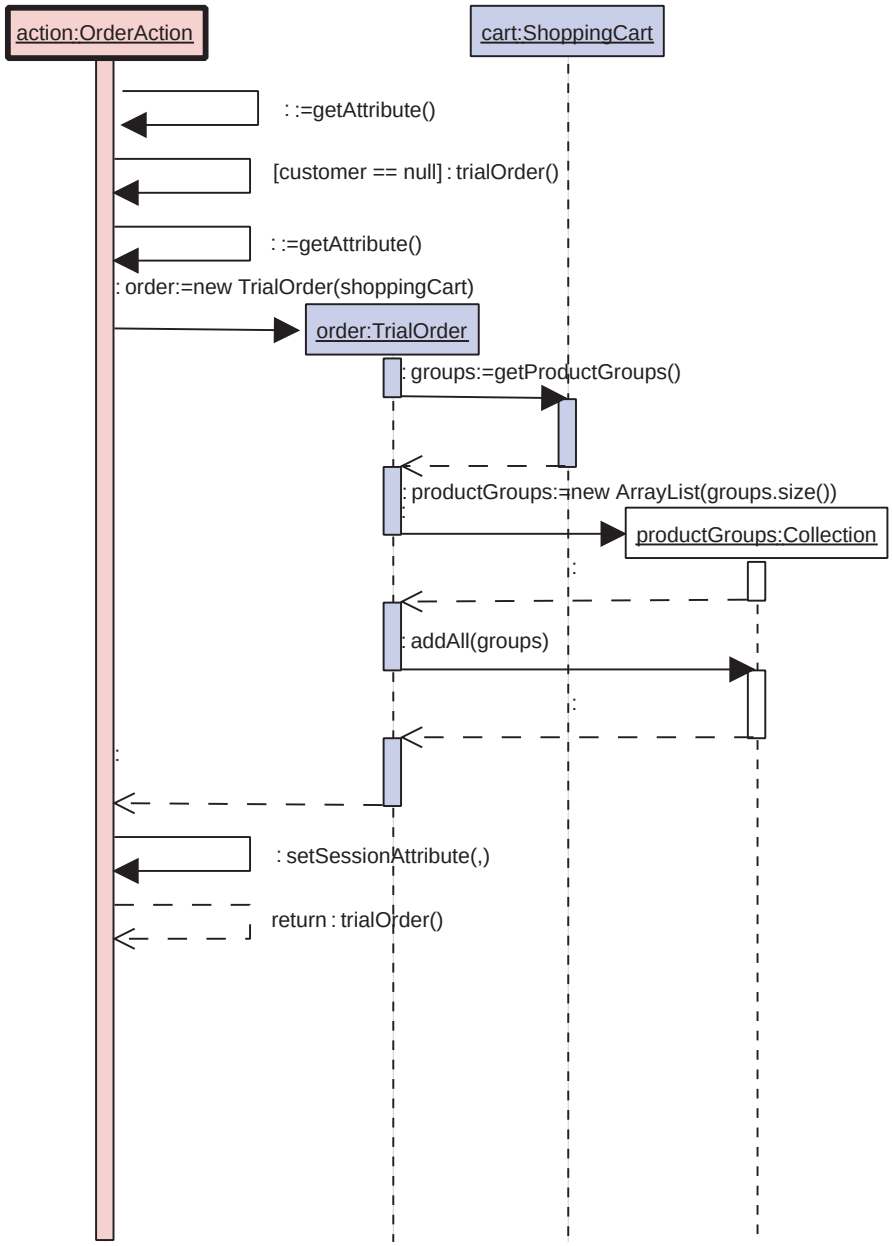

Abbildung 5.13 OrderAction Beispiel: Vorbereitung einer Schnupperbestellung

(d) Beispiel: SetPaymentAction

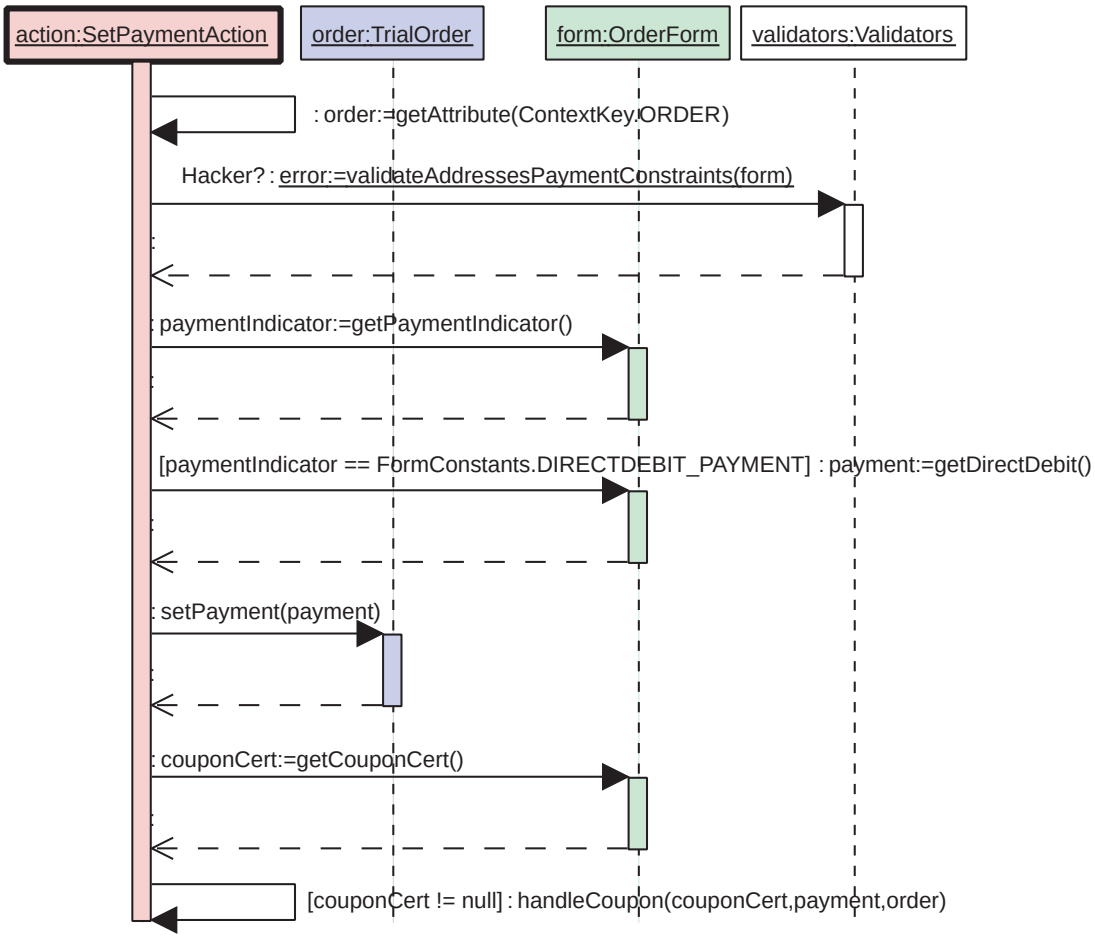

Abbildung 5.14 SetPaymentAction Beispiel mit Zahlung per Einzug und Gutschein

(e) Beteiligte Actions

- **Methoden der OrderAction**

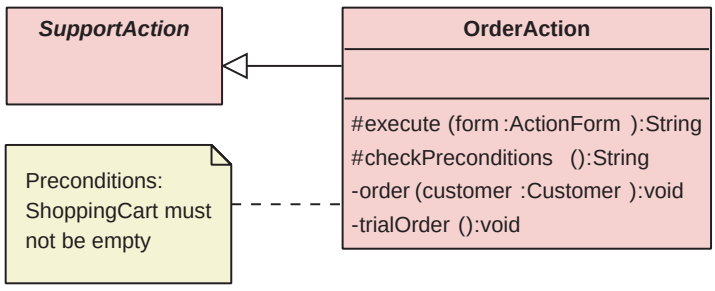

Abbildung 5.15 Methoden der OrderAction

Name	*Funktion*
-order(customer:Customer)	Beginnt eine neue Bestellung
-trialOrder()	Beginnt eine neue Schnupperbestellung

- **Methoden der SetAddressesAction**

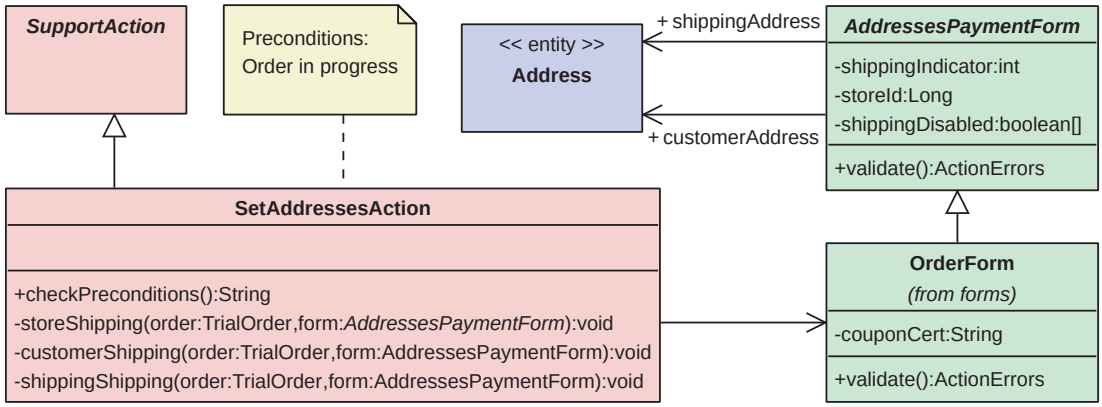

Abbildung 5.16 Methoden der SetAddressesAction

Name	*Funktion*
-storeShipping(...)	Setzt eine Filiale als Lieferadresse.
-customerShipping(...)	Setzt die Kundenadresse als Lieferadresse.
-shippingShipping(...)	Setzt die extra eingegebene Lieferadresse als Lieferadresse.

- **Methoden der SetPaymentAction**

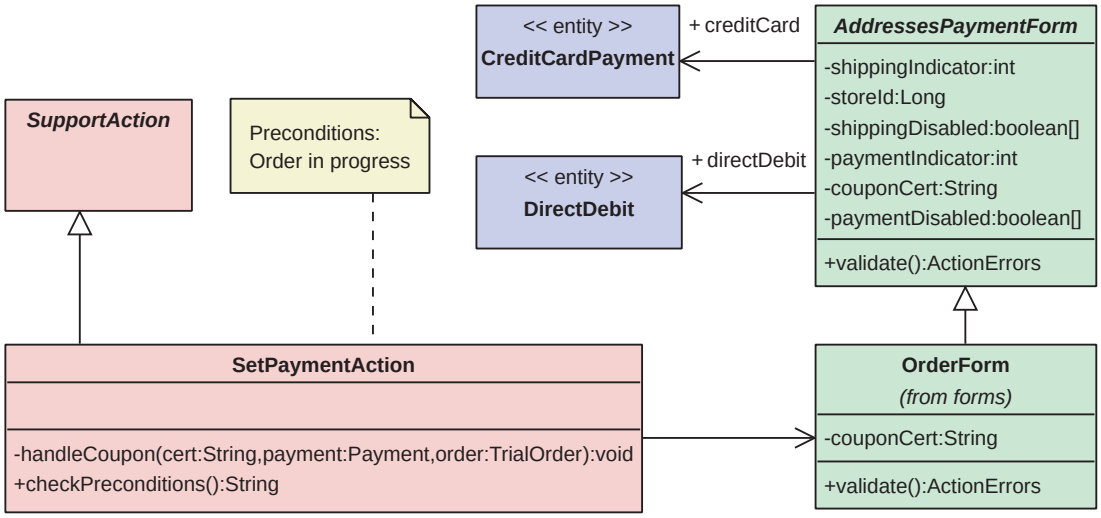

Abbildung 5.17 Methoden der SetPaymentAction

Name	*Funktion*
-handleCoupon(...)	Nimmt einen Gutschein zur Bezahlung entgegen und prüft seine Gültigkeit.

- **Methoden der ConfirmOrderAction**

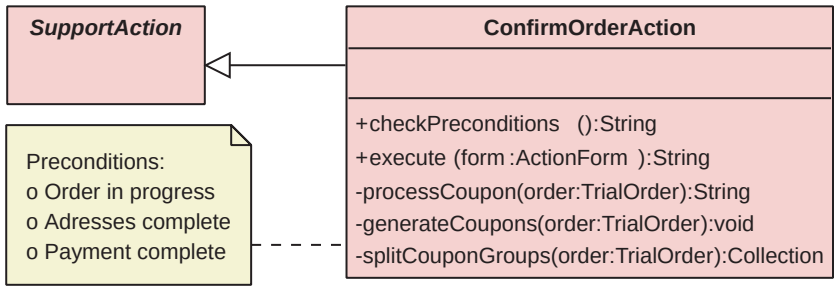

Abbildung 5.18 Methoden der ConfirmOrderAction

Name	*Funktion*
-processCoupon(...)	Überprüft die Gültigkeit eines Gutscheines, der zur Bezahlung angegeben wurde.
-generateCoupons(...)	Bereitet die Gutscheine, die bestellt werden sollen vor, indem neue Zertifikate und Bilder erzeugt werden.
-splitCouponGroups(...)	Teilt Mehrfachbestellungen eines Gutscheins in mehrere einzelne Gutscheine auf.

5.3 Bestellen /F20/

(a) Ablaufdiagramm

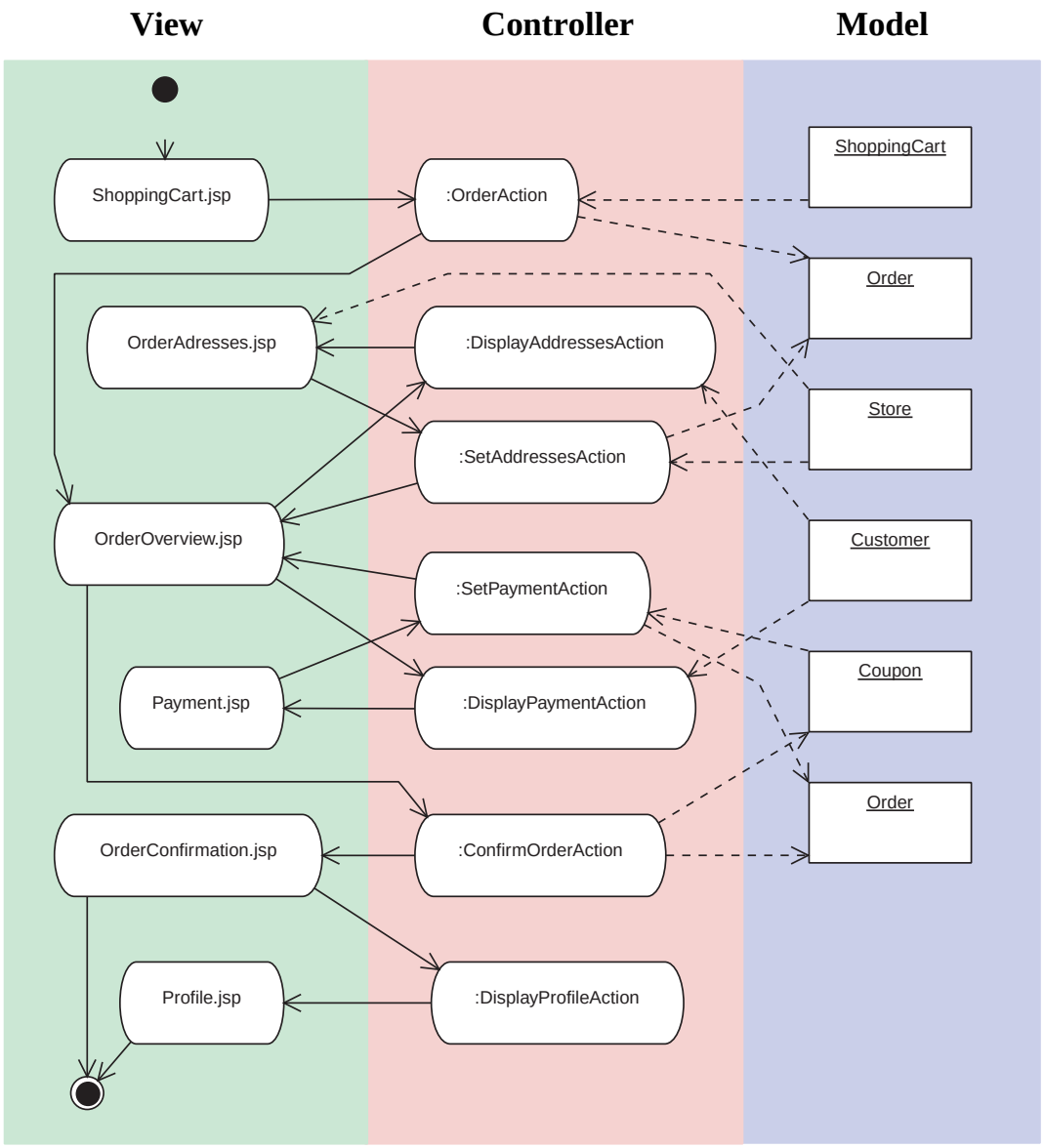

Abbildung 5.19 Bestellen /F20/

(b) Beschreibung

- Bei der eigentlichen Bestellung haben wir den selben Startpunkt wie bei der Schnupperbestellung: Den gefüllten Warenkorb. (4.1 /F20/ 2 im Pflichtenheft)
- Bei der Aktivierung einer Bestellung durch die `OrderAction` wird jetzt im Falle eines eingeloggten Kunden automatisch eine `Order` erstellt und in den Ablauf der Bestellung verzweigt. Die Order enthält dann eine Referenz auf das Kundenobjekt, sowie die Profileinstellungen des Kunden über Adressen und Zahlung und bekommt alle Produkte aus dem Warenkorb mit auf den Weg.
- Zentraler Punkt im normalen Bestellvorgang ist die Auftragsübersicht, die alle Daten des Auftrages zusammenfasst. Sie ist in der `OrderOverview.jsp` verwirklicht und bietet dem Kunden die Möglichkeiten mit einer Adressänderung, einer Änderung der Zahlungsart oder der Bestätigung der Bestellung fortzufahren. (4.1 /F20/ 4 im Pflichtenheft)
- Zur Änderung einer Adresse muss der Kunde auf die `OrderAddresses.jsp` wechseln, die von der `DisplayAddressesAction` die voreingestellt Adressdaten aus dem Kundenprofil übermittelt bekommt. (4.1 /F20/ 5 im Pflichtenheft). Hier kann der Kunde falls nötig eine Lieferadresse ändern oder eine Filiale zur Lieferung auswählen. Die Änderung der Kundenadresse ist hier nicht möglich.
- Die `SetAddressesAction` sorgt dafür, dass die eingegebenen Adressen und Lieferoptionen korrekt in das `Order`-Objekt übernommen werden. (Beispielablauf in Abbildung 5.20)
- Möchte der Kunde die Zahlungsart ändern oder einen Gutschein einlösen, so wechselt er zur Zahlungsansicht, die durch die `Payment.jsp` verwirklicht wird. Auch hier werden über die `DisplayPaymentAction` die Werte aus dem Kundenprofil zur Verfügung gestellt. (4.1 /F20/ 6 im Pflichtenheft)
- Die `SetPaymentAction` überprüft nochmals ob die gewählten Eingaben zulässig sind und ob der Gutschein, falls eingegeben, gültig ist und speichert die Informationen im `Order`-Objekt.
- Bestätigt der Kunde die Auftragsübersicht, so wird die Bestellung getätigt. Dies wird von der `ConfirmOrderAction` erledigt. Darin wird der Gutschein zur Zahlung nochmals überprüft, die bestellten Gutscheine generiert und die Bestellung in der Datenbank persistent gespeichert. (4.1 /F10/ 7 und 8 im Pflichtenheft) (Beispielablauf in Abbildung 5.21)
- Ist das erfolgreich verlaufen erhält der Kunde eine Auftragsbestätigung durch die `OrderConfirmation.jsp`. (4.1 /F10/ 9 im Pflichtenheft). Von dort kann er wenn gewünscht zu seinem Profil wechseln oder auch sich ausloggen.

(c) Beispiel: SetAddressesAction

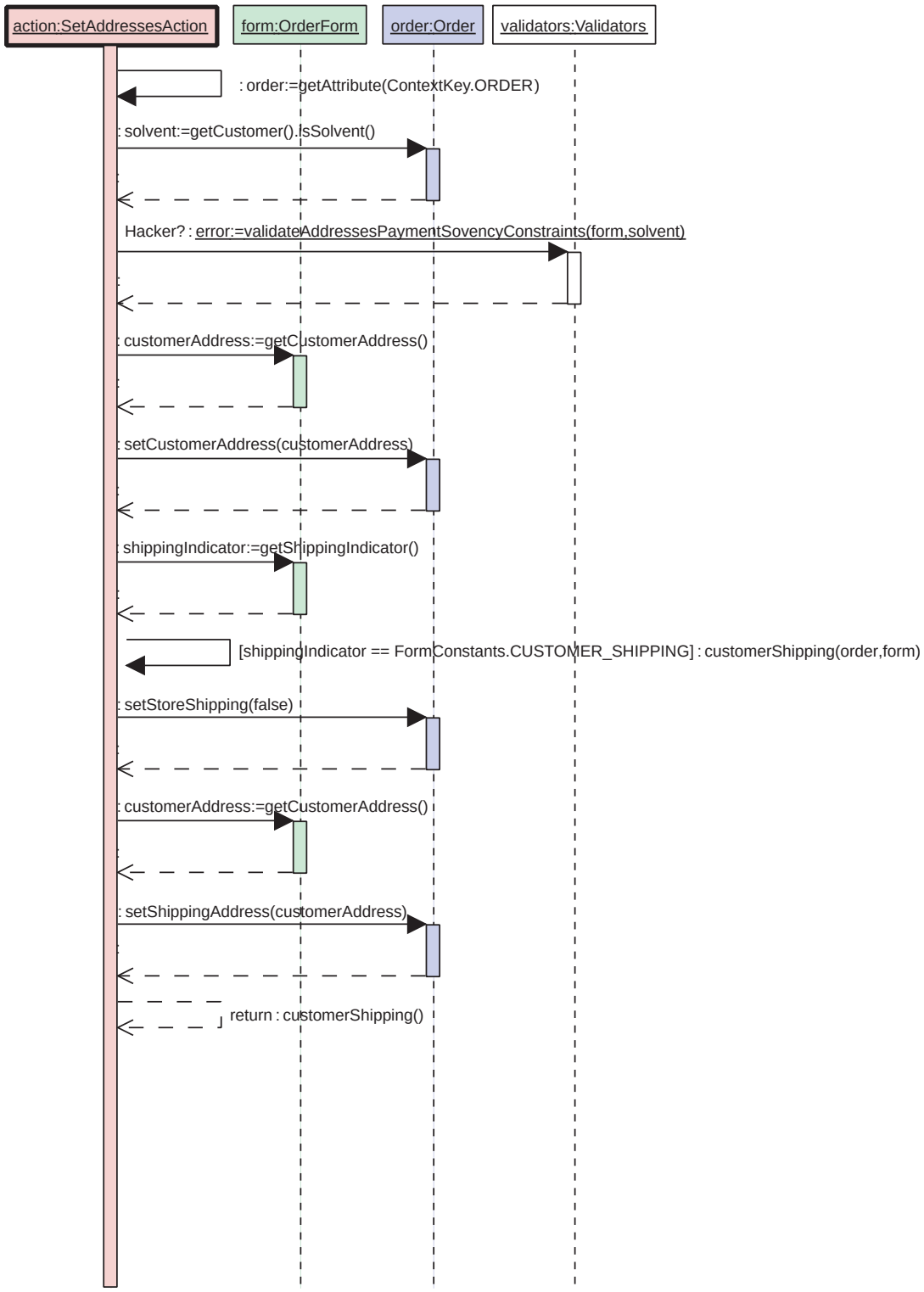

Abbildung 5.20 SetAddressesAction Beispiel mit Lieferung nach Hause

(d) Beispiel: ConfirmOrderAction

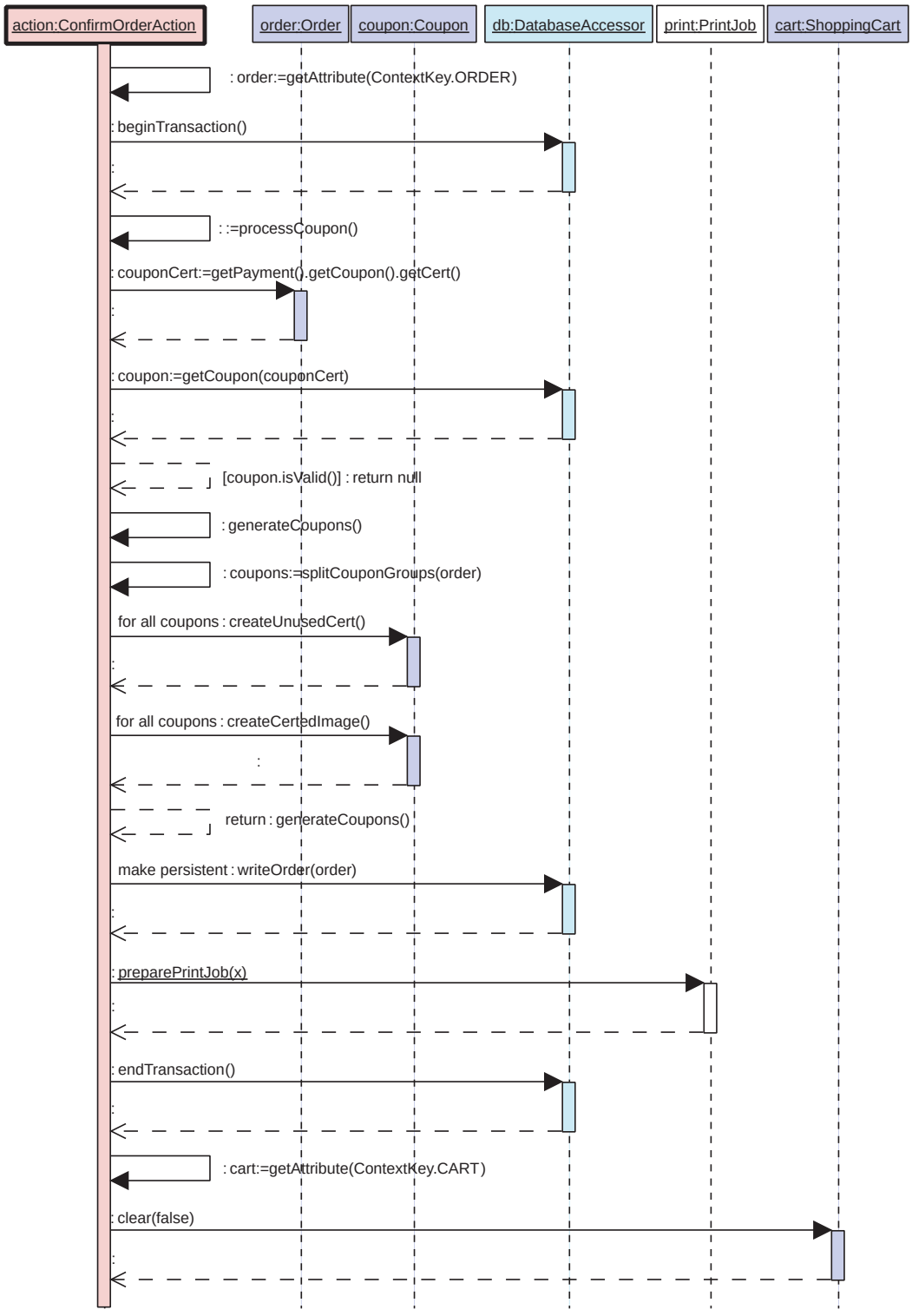

Abbildung 5.21 ConfirmOrderAction Beispiel mit bestellten Gutscheinen

5.3.1 Beteiligte Actions

Siehe Beteiligte Actions unter Schnupperbestellung.

5.4 Informieren /F30/

(a) Ablaufdiagramm

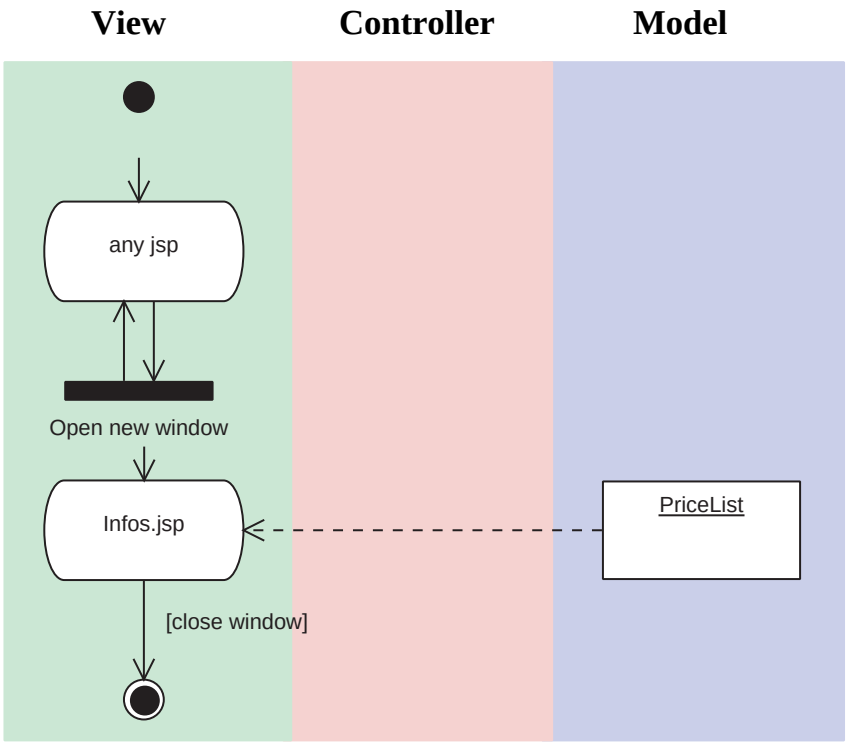

Abbildung 5.22 Informieren /F30/

(b) Beschreibung

- Der Kunde befindet sich auf einer beliebigen Seite der Applikation und möchte die aktuellen Preise oder AGBs einsehen.
- Beim Klick auf den entsprechenden Link öffnet sich ein neues Fenster mit den gewünschten Informationen, das der Kunde bei Bedarf wieder schließen kann. (4.1. / F30/ 1 im Pflichtenheft)

5.5 Profil ändern /F40/

(a) Ablaufdiagramm

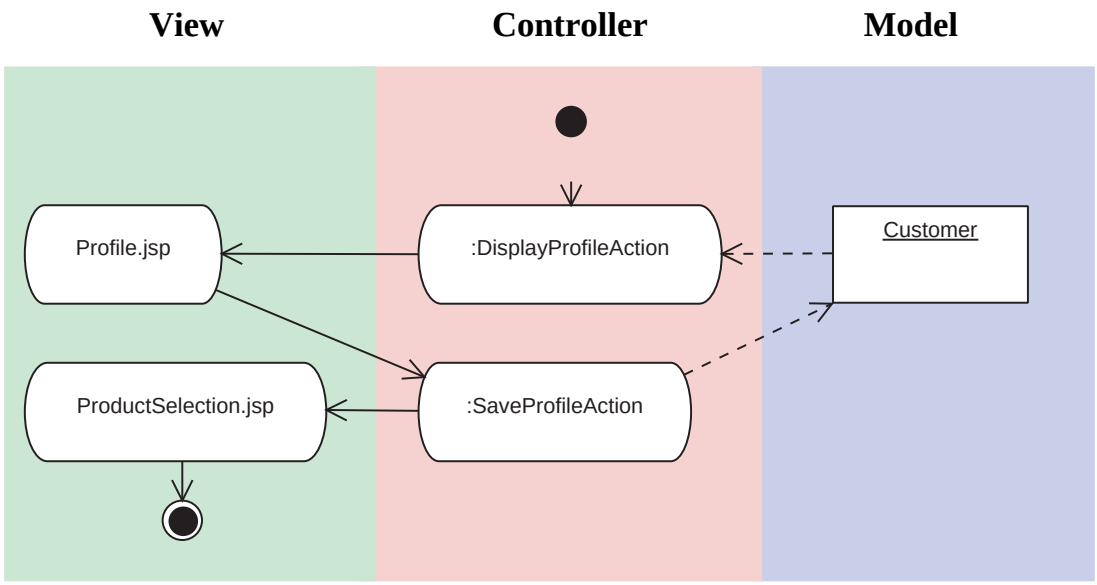

Abbildung 5.23 Profil ändern /F40/

(b) Beschreibung

- Wenn der eingeloggte Kunde sein Benutzerprofil ändern möchte, bekommt er zunächst durch `Profile.jsp` seine derzeitigen Einstellungen angezeigt. Diese werden durch die `DisplayProfileAction` bereitgestellt. (4.1 /F40/ 1 im Pflichtenheft)
- Bestätigt der Kunde die Änderungen, die er in `Profile.jsp` vorgenommen hat, so werden sie anschließend von der `SaveProfileAction` in sein Benutzerobjekt zurückgeschrieben und gespeichert. (4.1 /F40/ 2 und 3 im Pflichtenheft)
- Danach kann der Kunde mit der Produktauswahl weitermachen.

(c) Beispiel: SaveProfileAction

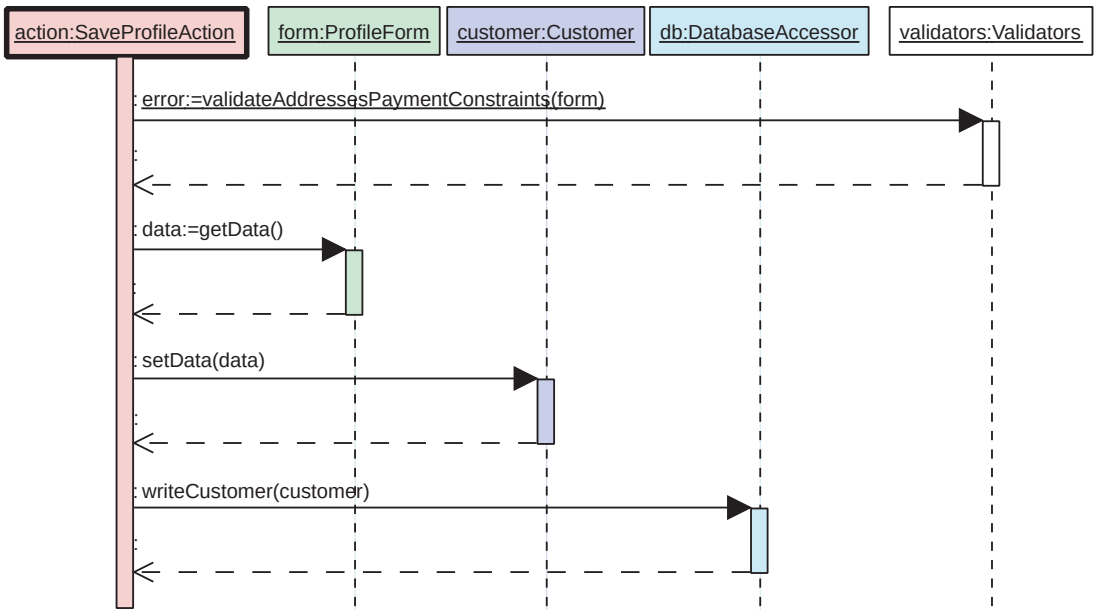

Abbildung 5.24 SaveProfileAction Beispiel: Änderung der Benutzereinstellungen

(d) Beteiligte Actions

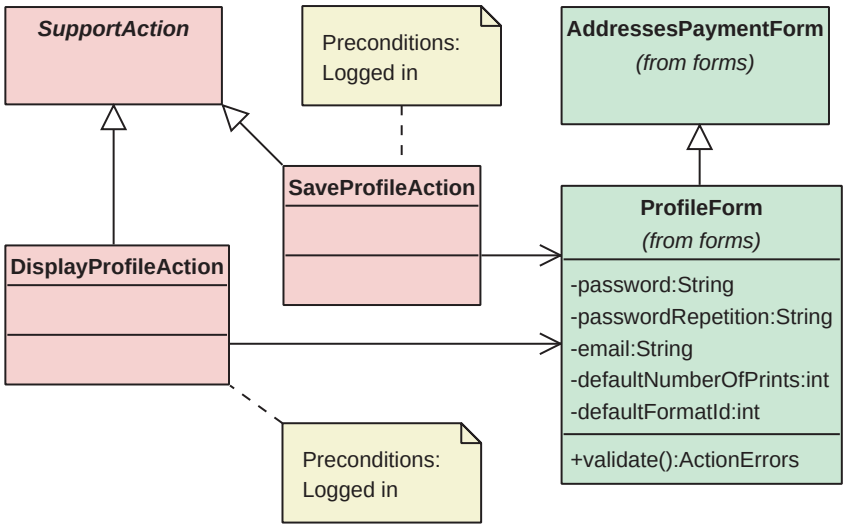

Abbildung 5.25 Beteiligte Actions

5.6 Registrieren /F50/

(a) Ablaufdiagramm

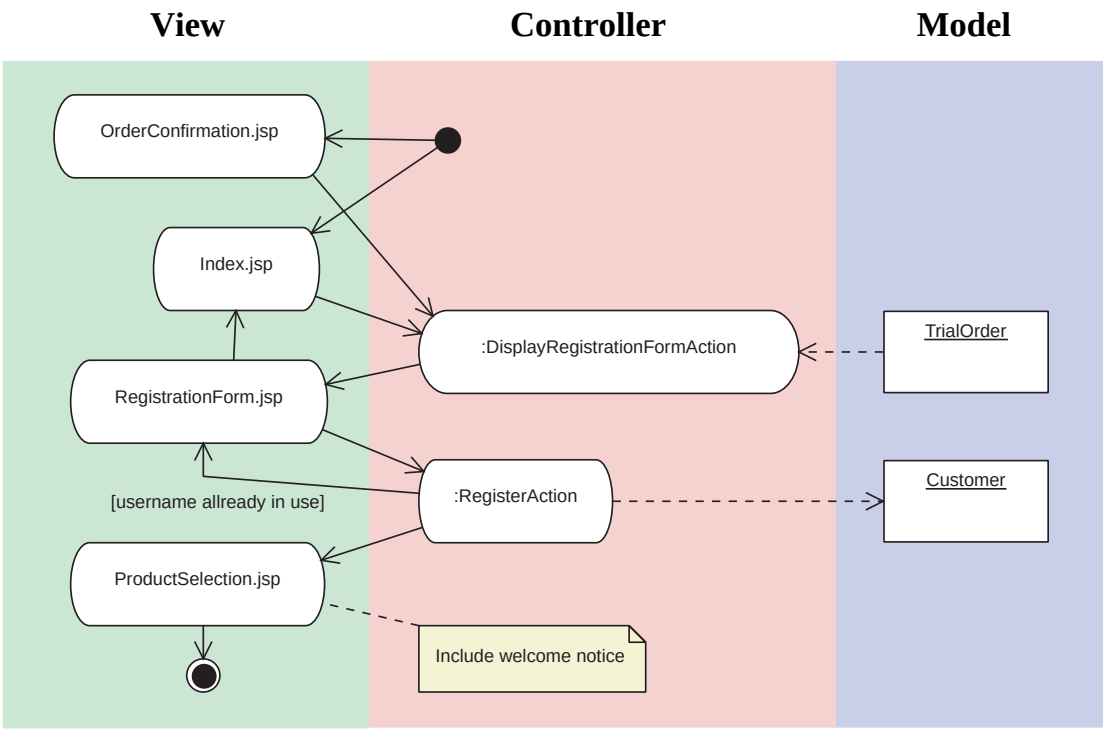

Abbildung 5.26 Registrieren /F50/

(b) Beschreibung

- Dieser Prozess wir durch zwei mögliche Auslöser angestoßen: Entweder ein Kunde entschließt sich einfach so zur Registrierung und wählt diese Option auf der Startseite der Applikation oder der Kunde hat gerade eine Schnupperbestellung getätigt und möchte sich nun registrieren.
- Im ersten Fall bekommt er ein leeres Registrierungsformular, das er ausfüllt, im Zweiten werden die Daten aus der Bestellung von der `DisplayRegistrationFormAction` aufbereitet. Für die Anzeige der Daten und des Formulars ist die `RegistrationForm.jsp` zuständig. (4.1. /F50/ 1,2,3 im Pflichtenheft)
- Beim Abschicken der Registrierung wird die `RegisterAction` aktiv. Sie prüft, ob der gewünschte Benutzername bereits vergeben ist oder nicht und führt dann die Registrierung durch oder leitet zur Eingabeseite zurück. (4.1. /F50/ 4,5 im Pflichtenheft)
- Im Erfolgsfall ist der Kunde danach eingeloggt bekommt einen Willkommensgruß und kann mit der Produktauswahl beginnen. (4.1 /F50/ 6 im Pflichtenheft)

(c) Beispiel: RegisterAction

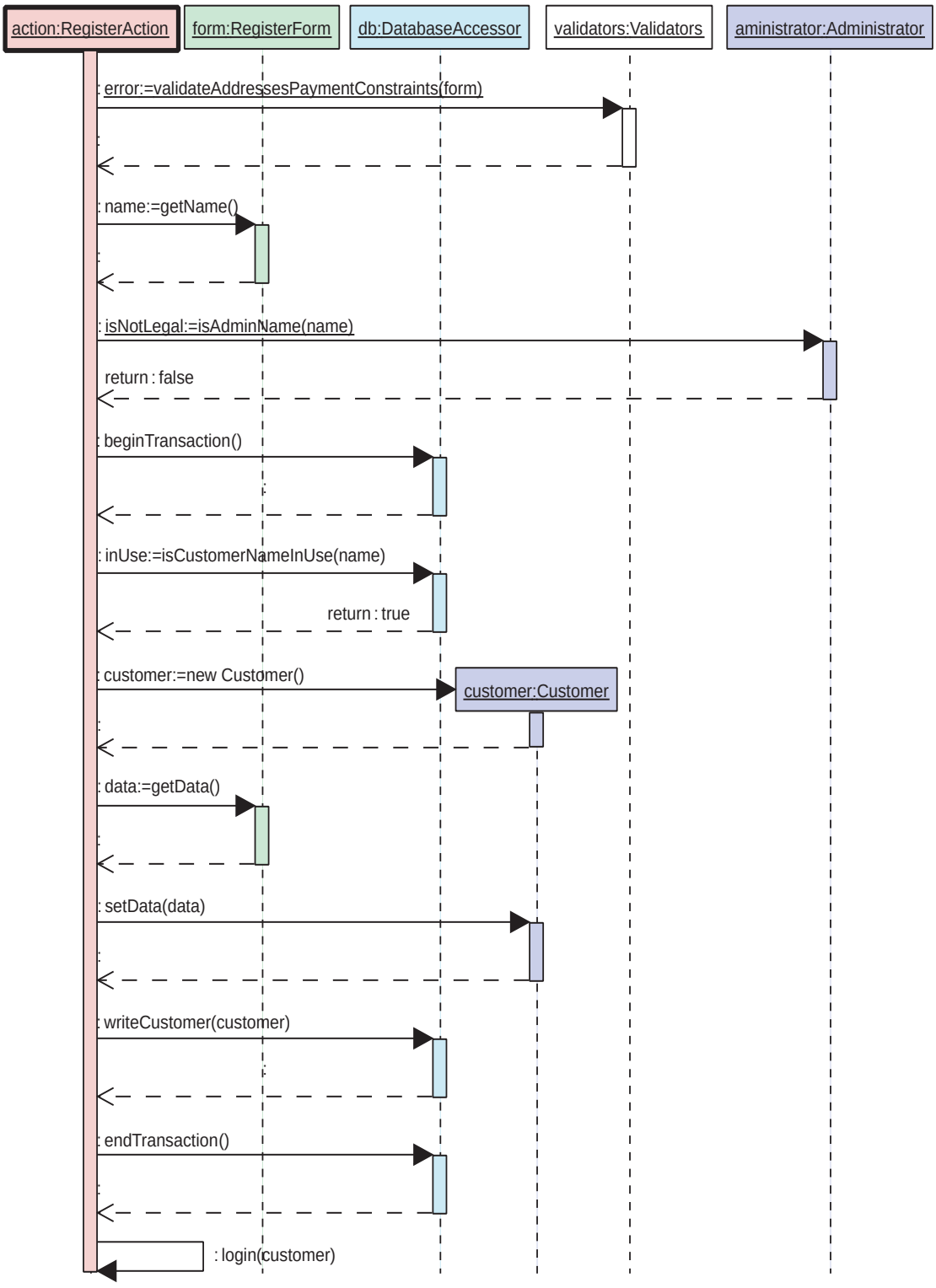

Abbildung 5.27 RegisterAction Beispiel: Erfolgreiche Registrierung eines Kunden

(d) Beteiligte Actions

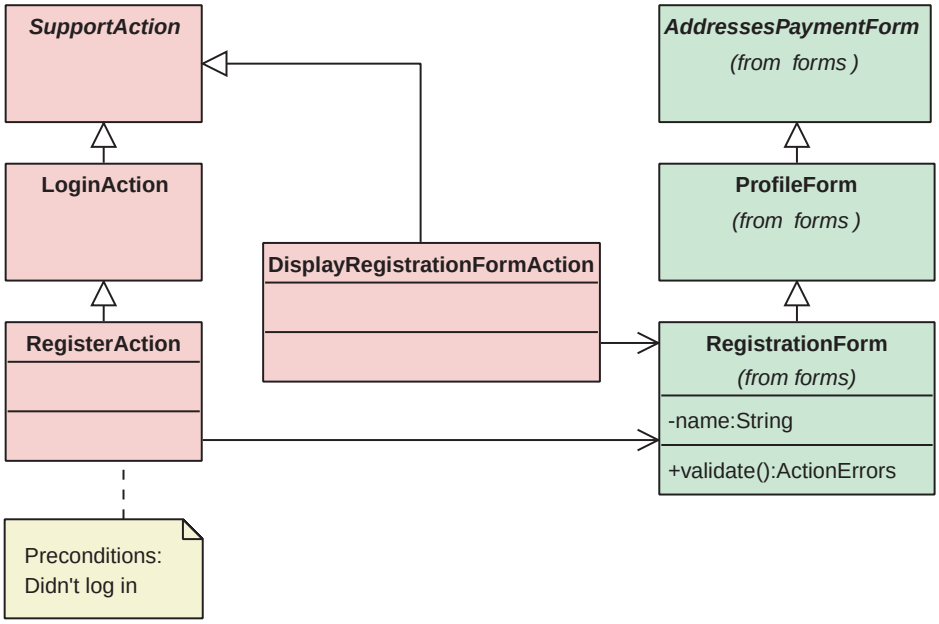

Abbildung 5.28 Beteiligte Actions

5.7 Login /F55/

(a) Ablaufdiagramm

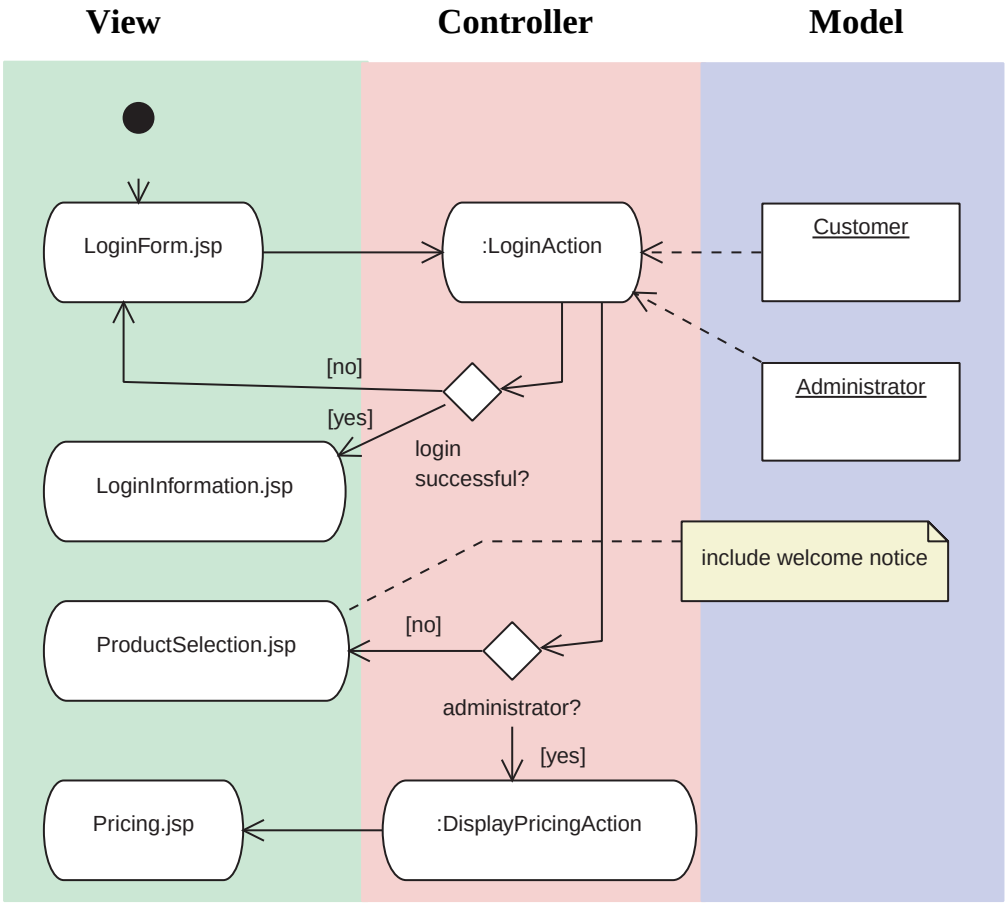

Abbildung 5.29 Login /F55/

(b) Beschreibung

- Wenn sich eine Person einloggen möchte wird zunächst versucht das zum Benutzernamen passende Person-Objekt zu ermitteln. Dies kann sowohl ein Kunde als auch der Administrator sein. Beim Fehlschlag geht es gleich zurück zur Eingabeseite, die darauf hinweist, dass Benutzername oder Passwort falsch war. (4.1. /F55/ 1,2,3 im Pflichtenheft)
- Dann muss das Passwort überprüft werden. Auch hier geht's im Fehlerfall zurück zur Eingabeseite. (4.1. /F55/ 2 im Pflichtenheft)
- Steht dem nichts im Wege wird die Person eingeloggt. Dies geschieht, indem ihr Objekt im Sessionkontext der Applikation abgelegt wird. Ist die Person ein Kunde, so ist seine Bonität darin implizit ebenfalls enthalten. (4.1. /F55/ 4 im Pflichtenheft)

- Je nach eingeloggter Person geht es dann auf die entsprechende Anzeigeseite weiter, auf der eine Begrüßung angezeigt wird. (4.1. /F55/ 4a,5 im Pflichtenheft)
- Die Vorgabe der Barzahlung und Filiallieferung entfällt an dieser Stelle; sie ist im Bestellprozess verwirklicht.

(c) Beispiel: LoginAction

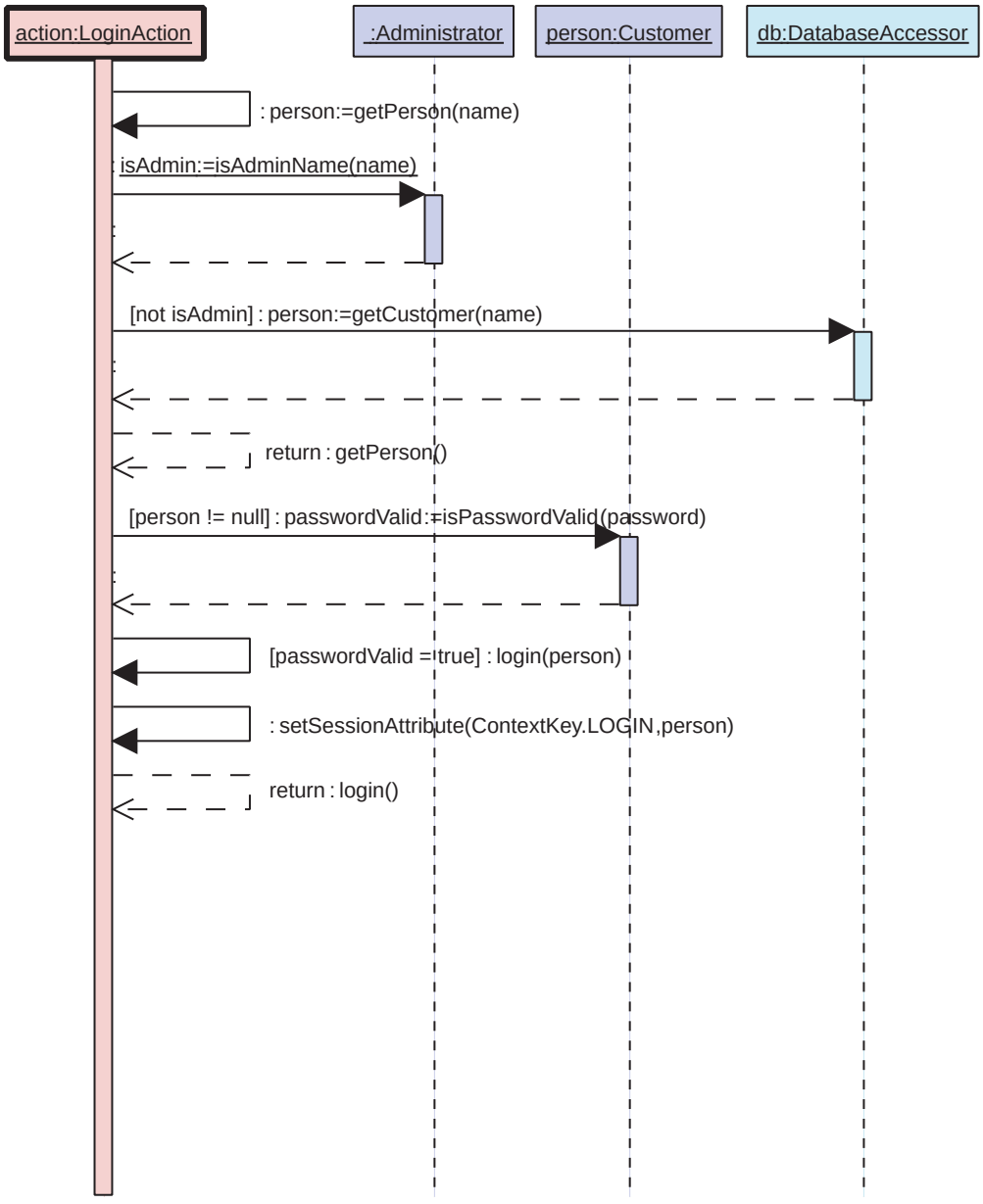

Abbildung 5.30 LoginAction Beispiel: Erfolgreiche Einloggen eines Kunden

(d) Beteiligte Actions

- **Methoden der LoginAction**

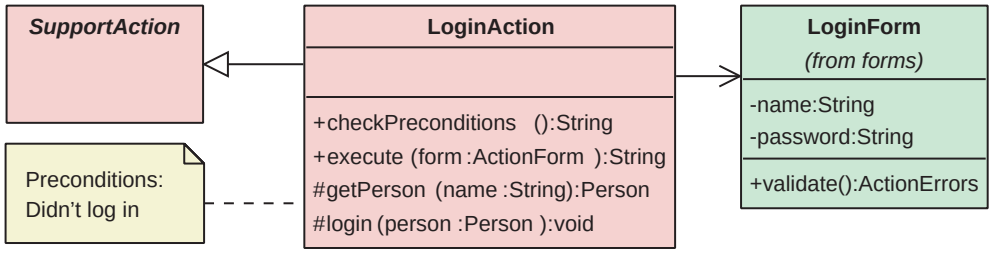

Abbildung 5.31 Methoden der LoginAction

Methode	*Beschreibung*
#getPerson(name:String): Person	Versucht die Person mit dem gegebenen Namen aus der Datenbank zu bekommen.
#login(person:Person)	Loggt die Person ein indem ihr Objekt als Session-variable gespeichert wird. Auch der Bonitätsstatus wird so verfügbar gemacht.

5.8 Verwalten /F60/

(a) Ablaufdiagramm

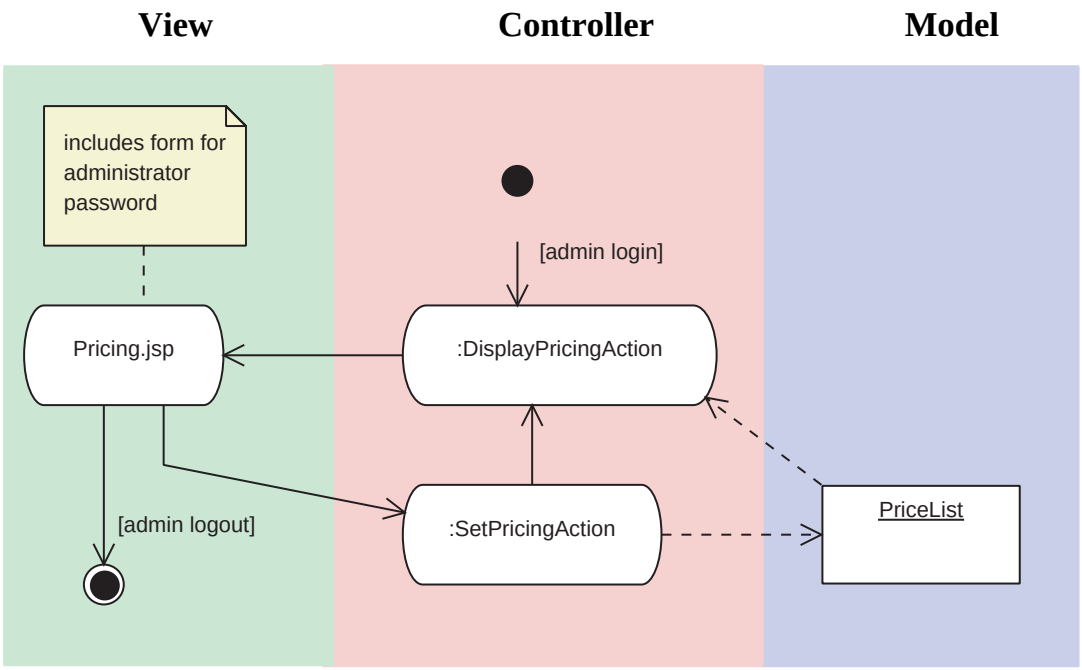

Abbildung 5.32 Verwalten /F60/

(b) Beschreibung

- Wenn sich der Administrator einloggt wird durch die `DisplayPricingAction` die aktuelle Tabelle mit den Preisen und Sonderaktionen aus der Datenbank geladen und von der `Pricing.jsp` angezeigt.
- Der Administrator kann auf dieser Seite dann an diesen Daten Änderungen vornehmen und auch sein Passwort ändern. (4.1. /F60/ 1 im Pflichtenheft)
- Bei einer Bestätigung der Änderungen werden diese durch die `SetPricingAction` in die Datenbank übertragen und zur Ansicht zurückgekehrt. (4.1. /F60/ 2 im Pflichtenheft)

(c) Beispiel: SetPricingAction

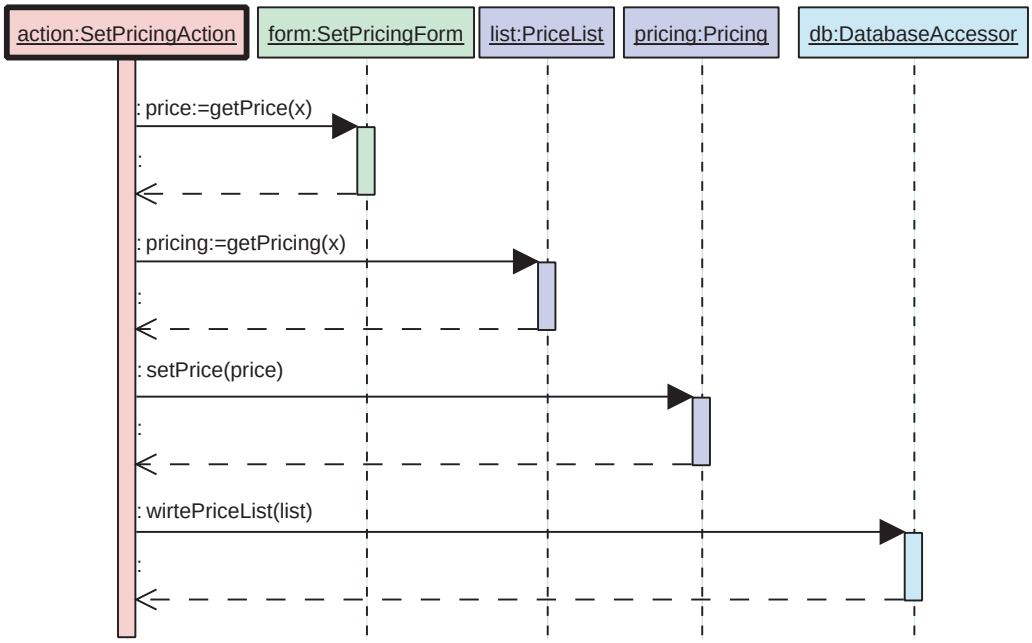

Abbildung 5.33 SetPricingAction Beispiel: Änderung eines Preises

(d) Beteiligte Actions

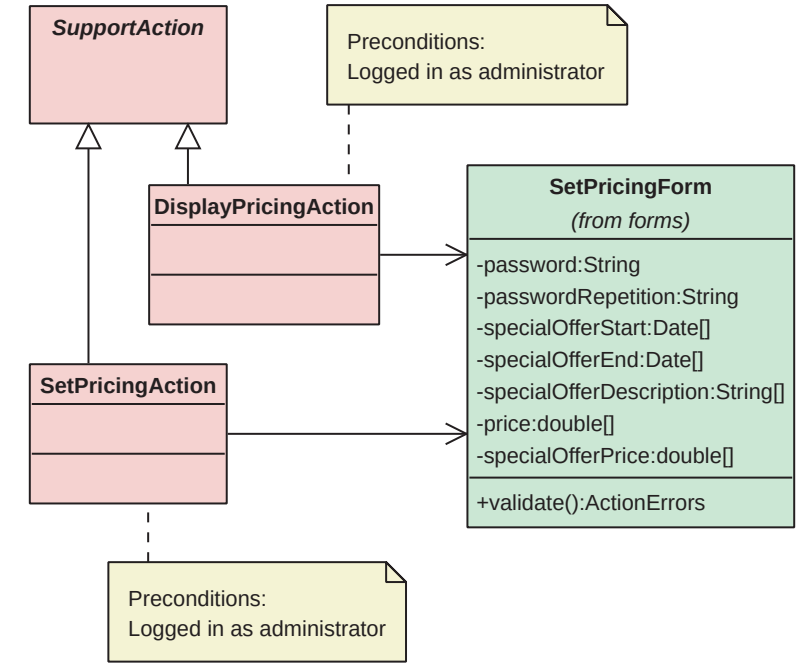

Abbildung 5.34 Beteiligte Actions

5.9 Automatische Systempflege /F70/

(a) Ablaufdiagramm

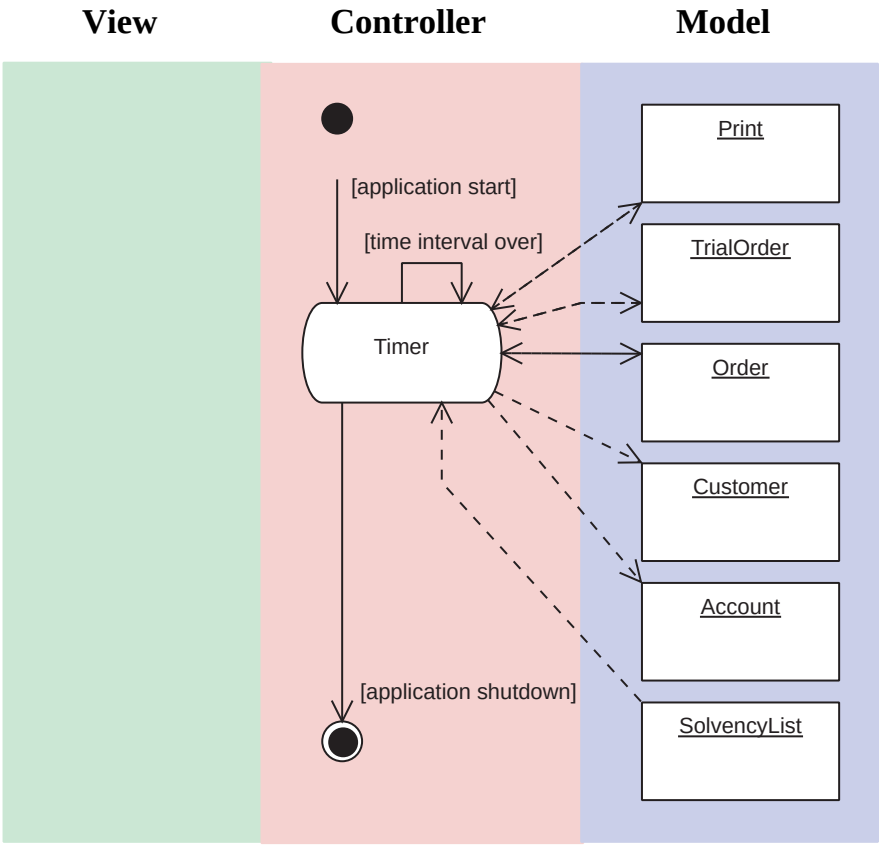

Abbildung 5.35 Automatische Systempflege /F70/

(b) Beschreibung

- Beim Start der Applikation wird ein Zeitgeber aktiviert, der jeden Tag einmal die Batch-Jobs ausführt. Diese sind Bonitätsliste einlesen, Abrechnungsliste erstellen und das Löschen der alten Bilder.
- Beim Einlesen der Bonitätsliste werden alle `Customer` die durch die Ids in der Liste identifiziert werden auf fehlende Bonität gesetzt alle anderen behalten ihre Bonität bzw. erhalten sie zurück. (4.1. /F70/ 1 im Pflichtenheft)
- Das Erstellen der Abrechnungsliste geschieht durch Iterieren aller noch nicht abgerechneter `Order`- und `TrialOrder`-Objekte und Schreiben der dort gespeicherten Daten in die XML-Datei (4.1. /F70/ 2 im Pflichtenheft)

- Zum Schluss werden noch alle `Print`-Objekte, die noch auf eine Datei verweisen und älter als der Schwellwert von 7 Tagen sind ermittelt und die verknüpften Dateien gelöscht. (4.1. /F70/ 3 im Pflichtenheft)

(c) Beteiligte Klassen

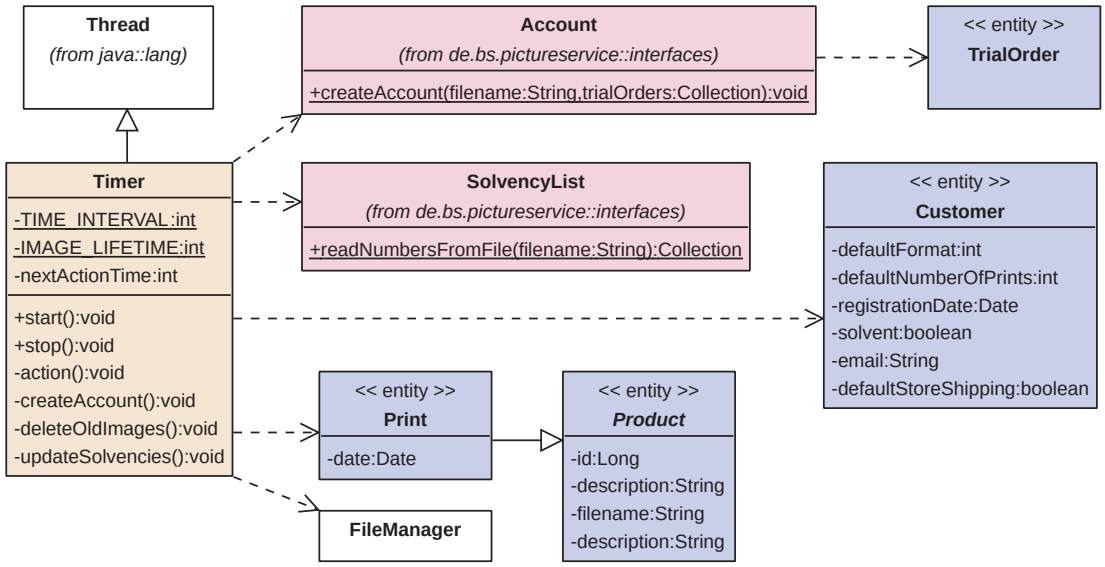

Abbildung 5.36 Beteiligte Klassen

- **Methoden von Timer**

Name	***Funktion***
+start()	Startet den Timer
+stop()	Beendet den Timer
-action()	Wird automatisch alle TIME_INTERVAL msec aktiviert und sorgt dafür, dass die Timeraktionen ausgeführt werden.
-createAccount()	Erzeugt eine neue Abrechnungsliste
-deleteOldImages()	Löscht die Bildateien, die älter als IMAGE_LIFETIME sind.
-updateSolvencies()	Aktualisiert die Bonitäten der Kunden nach der aktuell vorliegenden Bonitätsliste.

5.10 Sonstiges

5.10.1 Bild abrufen

Im Browser des Kunden müssen auf manchen JSPs Vorschaubilder der Bestellung oder der Poster und Gutscheine angezeigt werden.

(a) Ablaufdiagramm

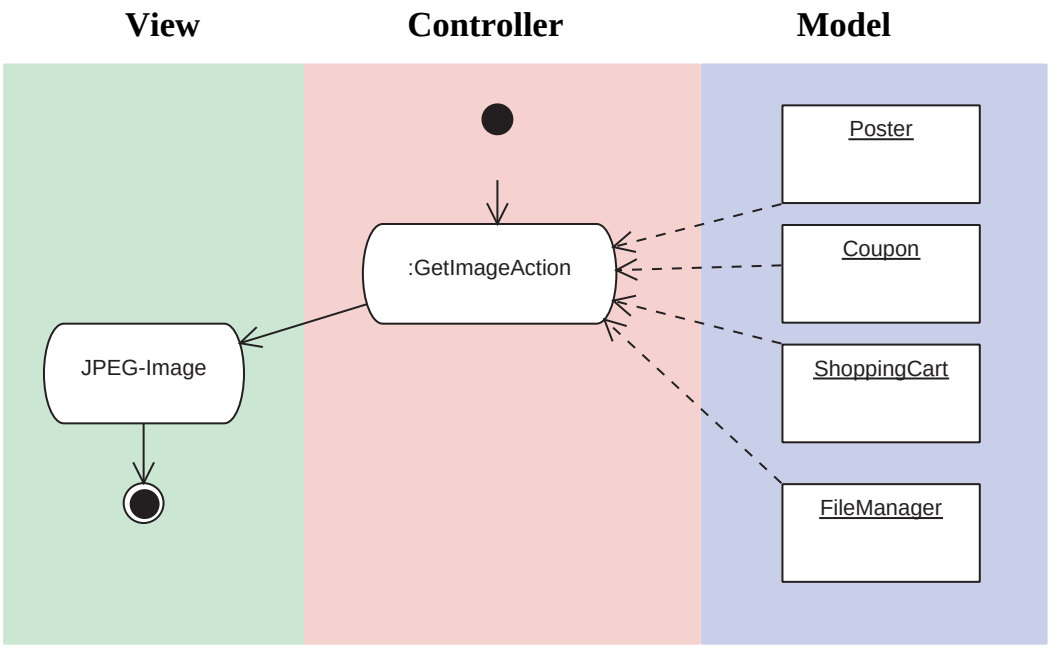

Abbildung 5.37 Bild abrufen

(b) Beschreibung

- Am Anfang steht eine Anfrage des Browsers an eine Instanz der `GetImageAction`, die ein Bild zurückliefern soll.
- `GetImageAction` prüft, ob der Zugriff zulässig ist. Dies ist der Fall wenn das gewünschte Bild ein Poster, eine Gutscheinvorlage oder ein Produkt aus dem Warenkorb des Kunden ist.
- Danach wird die gewünschte Bilddatei falls nötig erzeugt und an den Browser übertragen.

(c) Beispiel: GetPreviewAction

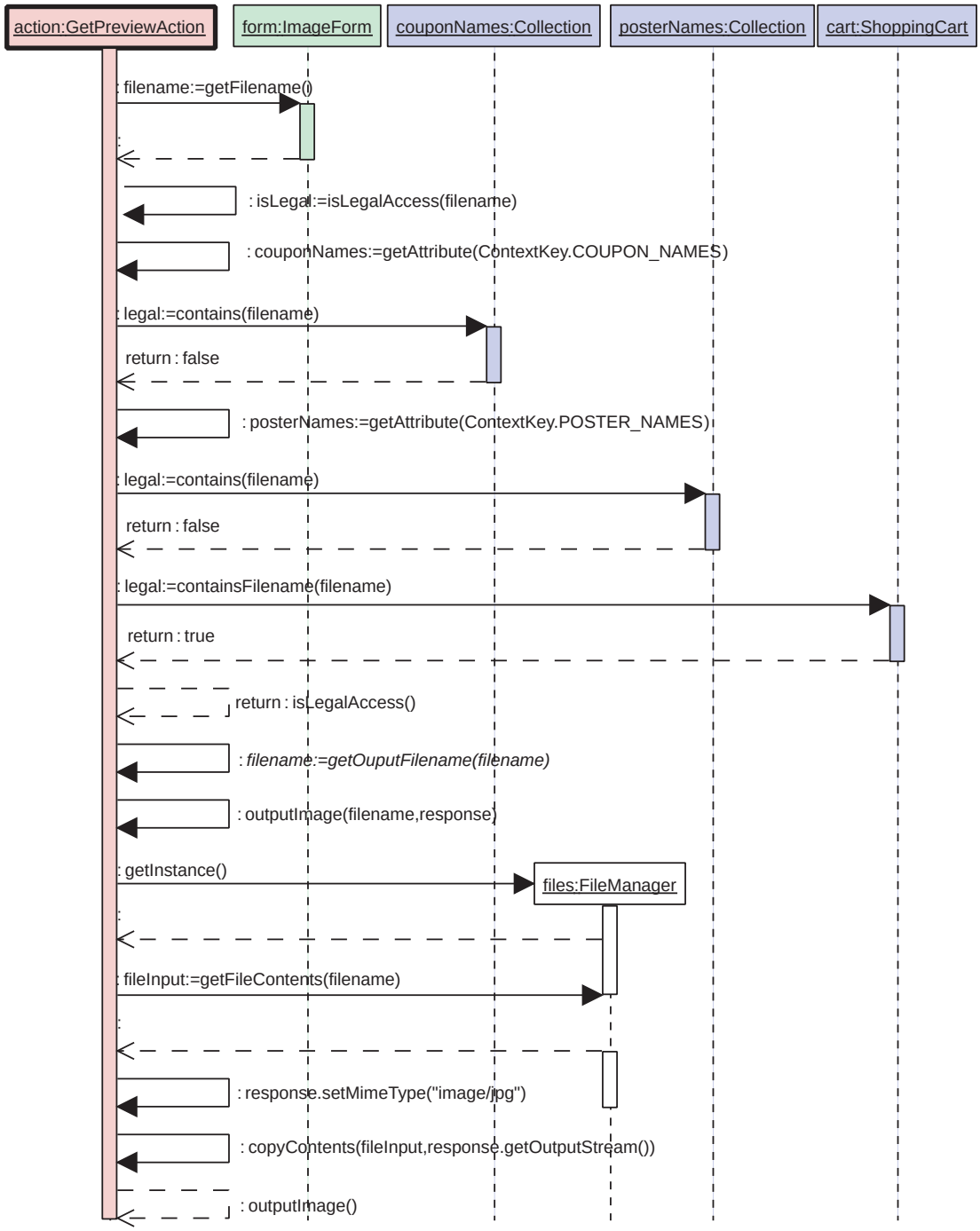

Abbildung 5.38 GetPreviewAction Beispiel: Abfrage eines hochgeladenen Bildes

(d) Beteiligte Actions

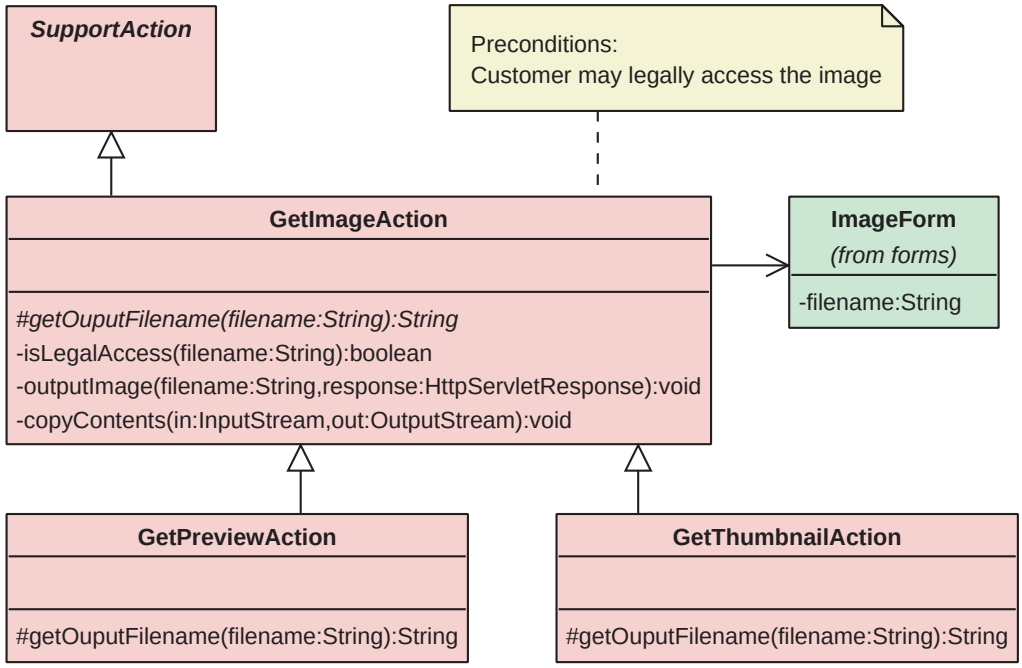

Abbildung 5.39 Beteiligte Actions

- **Methoden der GetImageAction**

Name	*Funktion*
#getOutputFilename(filename:String):String	Ermittelt den Dateinamen der tatsächlich auszugebenden Datei.
-isLegalAccess(filename:String):boolean	Prüft, ob der Kunde überhaupt auf das angegebene Bild zugreifen darf.
-outputImage(filename,response)	Liefert das Bild an den Aufrufenden zurück.
-copyContents(in,out)	Kopiert den Inhalt des Eingabestromes in den Ausgabestrom.

5.10.2 Logout

(a) Ablaufdiagramm

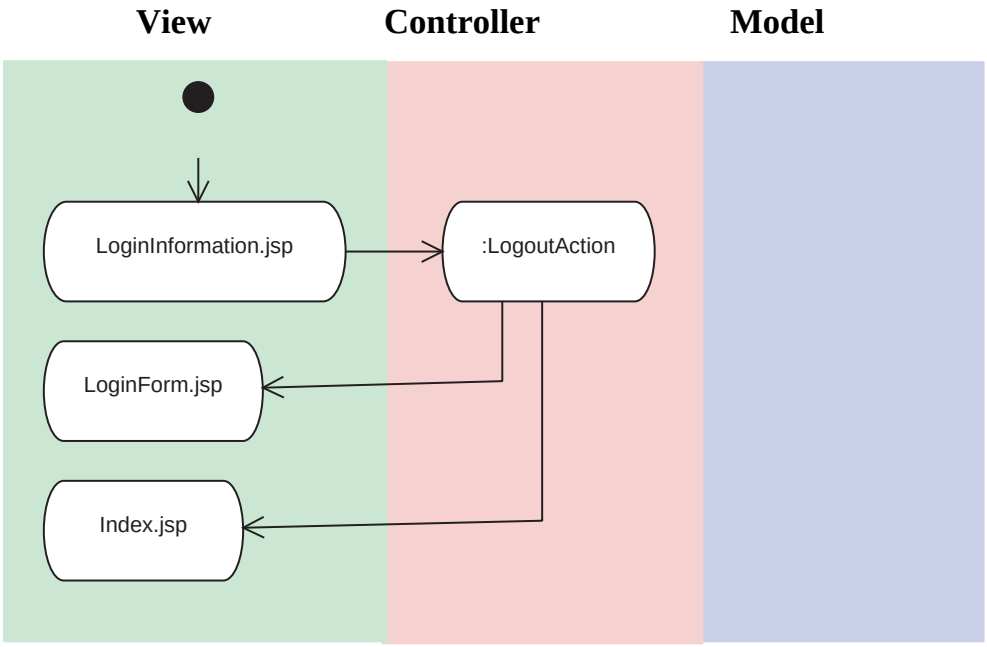

Abbildung 5.40 Logout

(b) Beschreibung

- Der Kunde oder Administrator zeigt seinen Logoutwunsch durch Anklicken des entsprechenden Links auf der `LoginInformation.jsp` an.
- Dann werden die Login-Informationen durch die `LogoutAction` aus den entsprechenden Kontexten gelöscht.
- Und im Browser wird wieder die Hauptseite (`Index.jsp`) und das Login-Formular (`LoginForm.jsp`) angezeigt.

(c) Beteiligte Actions

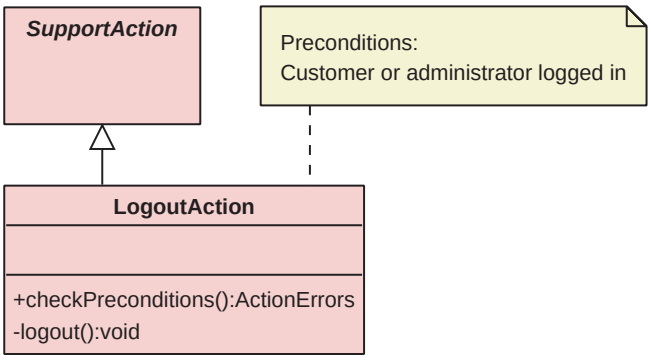

Abbildung 5.41 Beteiligte Actions

6 Dokumenttyp-Definitionen

In diesem Kapitel wollen wir die (XML-)Dokumente definieren, die unser System mit anderen Systemen verbindet. Dabei möchten wir im Designheft das Augenmerk auf Beispieldokumente richten. Sie veranschaulichen, wie ein solches Dokument aussehen kann. Die formale Definition (als XML-Schema) entnehmen Sie bitte dem Anhang.

6.1 Lieferschein (/F80/)

```
<?xml version="1.0" encoding="UTF-8"?>
<DeliveryNote ID="23" CreatedAt="2004-12-03T16:08:19+01:00"
xmlns="http://www.bs.de/pictureservice/deliveryNote"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.bs.de/pictureservice/deliveryNote
deliveryNote.xsd">
	<Total>18.31</Total>
	<OrderTotal>26.31</OrderTotal>
	<Customer ID="1234567">
		<Address>
			<Name>Emma Musterfrau</Name>
			<Street>Musterstraße 19a</Street>
			<City>Musterstadt</City>
			<PostCode>98765</PostCode>
		</Address>
	</Customer>
	<Paid>true</Paid>
	<Image Filename="portrait.jpg">
		<ImageGroup>
			<Format>9x13</Format>
			<Number>2</Number>
			</ImageGroup>
	</Image>
	<Image Filename="tiger.jpg">
		<ImageGroup>
			<Format>30x45</Format>
			<Number>1</Number>
			</ImageGroup>
	</Image>
	<Image Filename="castle.jpg">
		<ImageGroup>
			<Format>10x15</Format>
			<Number>1</Number>
		</ImageGroup>
	</Image>
	<Image Filename="coupon01.jpg">
		<Description>Coupon delivered as a standard print.</Description>
		<ImageGroup>
			<Format>9x13</Format>
			<Number>1</Number>
		</ImageGroup>
	</Image>
</DeliveryNote>
```

6.2 Abrechnungsliste (/F90/)

```
<?xml version="1.0" encoding="UTF-8"?>
<Account CreatedAt="2004-12-17T04:00:00+01:00"
xmlns="http://www.bs.de/pictureservice/account"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.bs.de/pictureservice/account
account.xsd">
	<Order ID="8934174902" Total="213.98">
		<Customer ID="1234567">
			<Address>
				<Name>Emma Musterfrau</Name>
				<Street>Musterstraße 19a</Street>
				<City>Musterstadt</City>
				<PostCode>98765</PostCode>
			</Address>
		</Customer>
		<Payment>
			<CreditCard Type="VISA">
				<Number>4321432143214321</Number>
				<Holder>Emma Musterfrau</Holder>
				<expirationDate>2005-10</expirationDate>
			</CreditCard>
		</Payment>
	</Order>
	<Order ID="3425632235" Total="0.00">
		<Customer ID="0052564">
			<Address>
				<Name>Reinhold Messherr</Name>
				<Street>Am Gipfel 1</Street>
				<City>Höhendorf</City>
				<PostCode>01244</PostCode>
				<Store>false</Store>
			</Address>
		</Customer>
		<Payment>
			<Cash/>
		</Payment>
		<Coupons>
			<Cert>453249263492629352522352</Cert>
			<Value>100.00</Value>
		</Coupons>
	</Order>
</Account>
```

6.3 Bonitätsliste[20]

```
<?xml version="1.0" encoding="UTF-8"?>
<SolvencyList CreatedAt="2004-05-31T13:20:00+01:00"
xmlns="http://www.bs.de/pictureservice/solvencyList"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.bs.de/pictureservice/solvencyList
solvencyList.xsd">
<Customer ID="0000000">1</Customer>
<Customer ID="1234567">0</Customer>
</SolvencyList>
```

20 Siehe Pflichtenheft Kapitel 10.4.2 Schnittstelle zum Abrechnungssystem (Seite 31)

7 GUI Prototyp

In diesem Kapitel haben wir Bildschirmfotos von dem GUI Prototypen aufgeführt. Auf diese Bilder werden in den anderen Kapiteln an geeigneten Stellen verwiesen.

7.1 Homepage

Abbildung 7.1 Homepage

7.2 Schnupperbestellung

Abbildung 7.2 Produktauswahl-Dialog bei der Schnupperbestellung

Abbildung 7.3 Hochladen-Dialog bei der Schnupperbestellung

Abbildung 7.4 Der Warenkorb bei der Schnupperbestellung

Abbildung 7.5 Adresseingabe-Dialog bei der Schnupperbestellung

Abbildung 7.6 Anzahl und Formate eines Bildes ändern

Abbildung 7.7 Zahlungsart-Dialog bei der Schnupperbestellung

Abbildung 7.8 Bestellübersicht bei der Schnupperbestellung

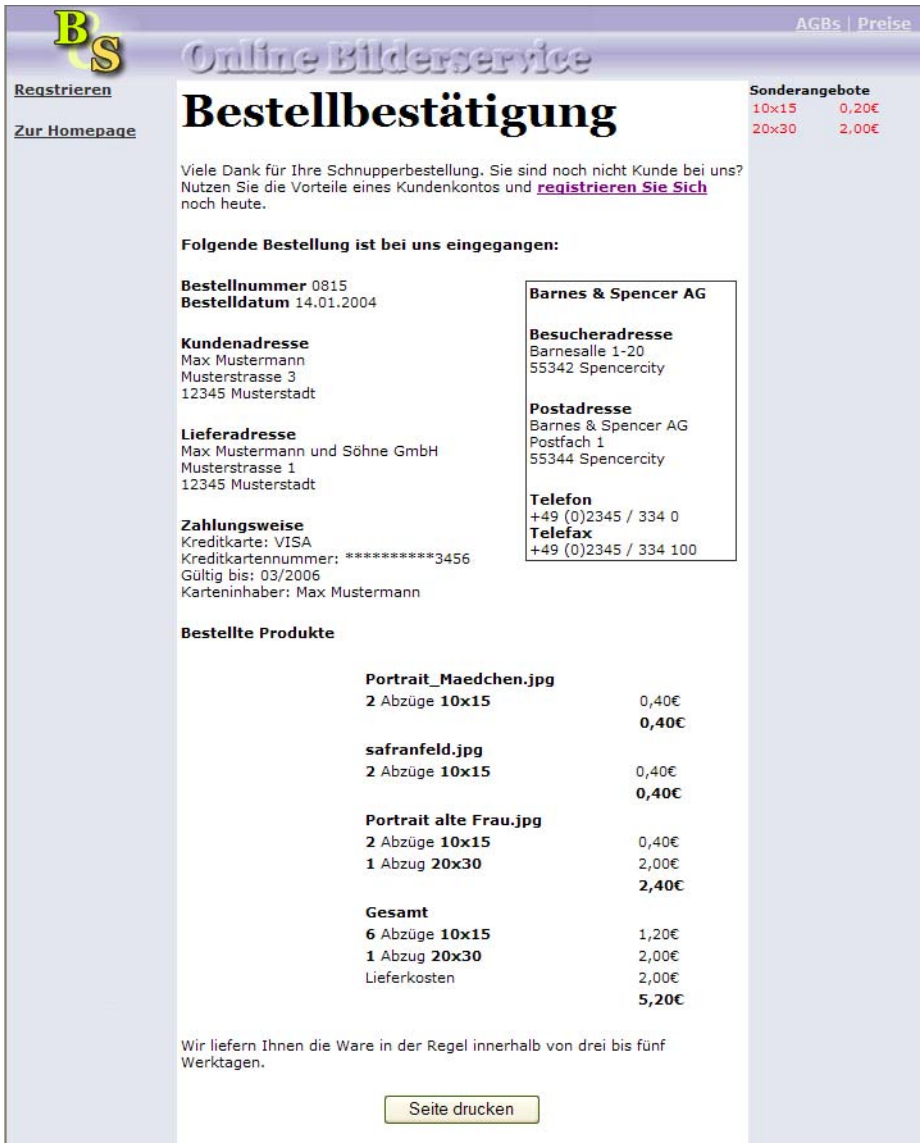

Abbildung 7.9 Bestellbestätigung bei der Schnupperbestellung

Bestellbestätigung

Viele Dank für Ihre Schnupperbestellung. Sie sind noch nicht Kunde bei uns? Nutzen Sie die Vorteile eines Kundenkontos und **registrieren Sie Sich** noch heute.

Folgende Bestellung ist bei uns eingegangen:

Bestellnummer 0815
Bestelldatum 14.01.2004

Kundenadresse
Max Mustermann
Musterstrasse 3
12345 Musterstadt

Lieferadresse
Max Mustermann und Söhne GmbH
Musterstrasse 1
12345 Musterstadt

Zahlungsweise
Kreditkarte: VISA
Kreditkartennummer: **********3456
Gültig bis: 03/2006
Karteninhaber: Max Mustermann

Barnes & Spencer AG

Besucheradresse
Barnesalle 1-20
55342 Spencercity

Postadresse
Barnes & Spencer AG
Postfach 1
55344 Spencercity

Telefon
+49 (0)2345 / 334 0
Telefax
+49 (0)2345 / 334 100

Bestellte Produkte

Portrait_Maedchen.jpg	
2 Abzüge **10x15**	0,40€
	0,40€
safranfeld.jpg	
2 Abzüge **10x15**	0,40€
	0,40€
Portrait alte Frau.jpg	
2 Abzüge **10x15**	0,40€
1 Abzug **20x30**	2,00€
	2,40€
Gesamt	
6 Abzüge **10x15**	1,20€
1 Abzug **20x30**	2,00€
Lieferkosten	2,00€
	5,20€

Wir liefern Ihnen die Ware in der Regel innerhalb von drei bis fünf Werktagen.

Abbildung 7.10 Druckversion der Bestellbestätigung bei der Schnupperbestellung

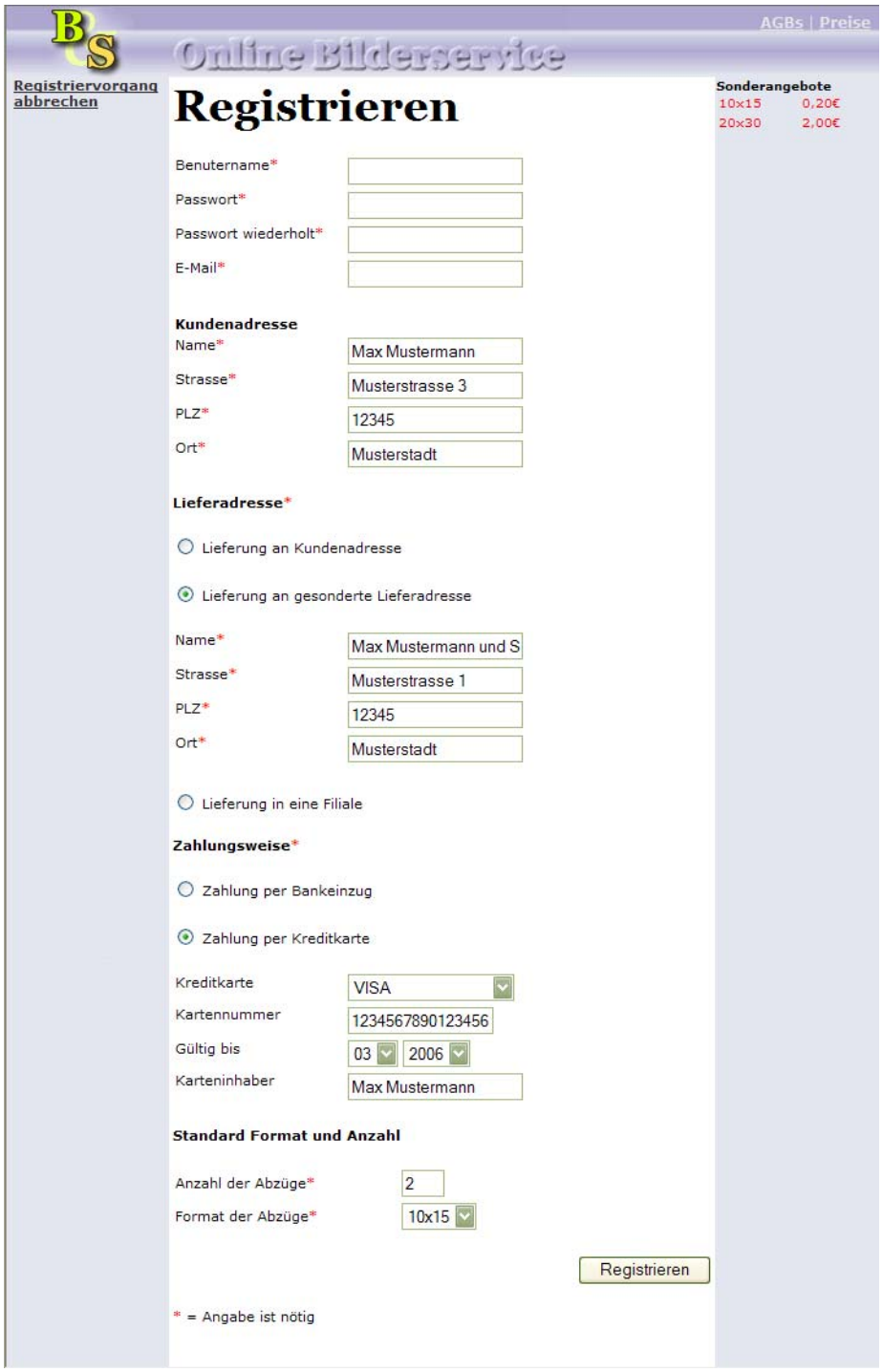

Abbildung 7.11 Registrieren-Dialog mit vorausgefüllten Werten nach einer Schnupperbestellung

7.3 Bestellung

Abbildung 7.12 Produktauswahl

Abbildung 7.13 Bilder Hochladen-Dialog

Abbildung 7.14 Der Warenkorb

Abbildung 7.15 Anzahl und Formate eines Bildes ändern

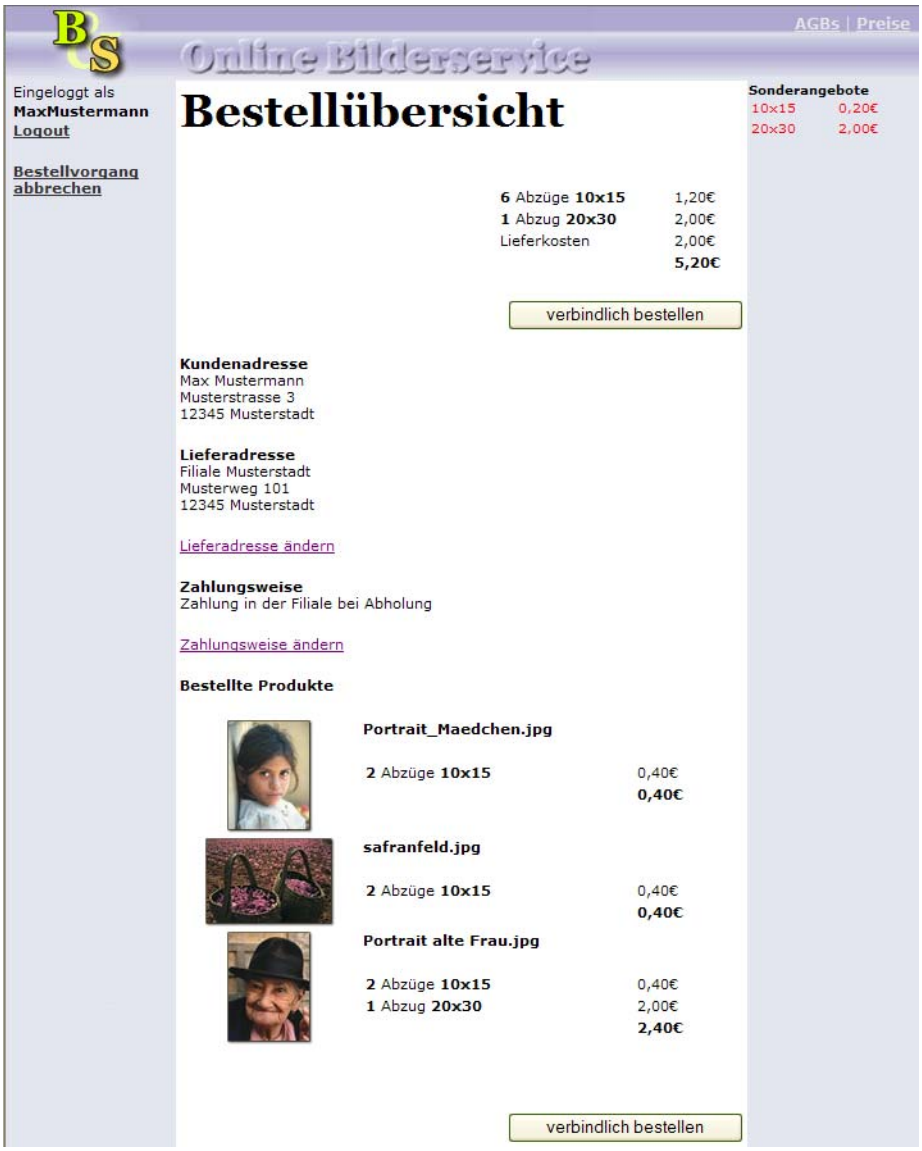

Abbildung 7.16 Bestellübersicht

Abbildung 7.17 Lieferadresse ändern

Abbildung 7.18 Zahlungsweise ändern

Abbildung 7.19 Bestellbestätigung