В этой главе:

- Веб-потоки и где они используются
- Реализация простых последовательностей действий в стиле мастера
- Реализация ветвления сложных условий
- Подпотоки и упрощение логики работы
- Использование служб с сохранением информации о состоянии в области видимости потока
- Тестирование веб-потоков

9

Реализация мастеров и последовательностей с помощью веб-потоков

К настоящему моменту мы рассмотрели много вопросов. Мы подробно изучили основы контроллеров, представлений и моделей – как они взаимодействуют и как их тестировать; существенно улучшили пользовательский интерфейс нашего приложения и при этом многое узнали о разработке приложений на платформе Grails. Однако мы еще не исследовали основы реализации последовательностей действий.

Как быть, если у вас имеется процедура регистрации, занимающая несколько страниц? Или если при расчете за покупку необходимо предоставить пользователю выбор способа доставки, исходя из его места проживания? Вы сможете решить эти проблемы при текущем уровне знаний контроллеров в связке с механизмом перенаправления и скрытых переменных форм, но при этом вы создали бы свой механизм определения текущего состояния, а в зависимости от задачи сложность ее реализации может возрастать очень быстро.

Механизм веб-потоков в платформе Grails спроектирован именно для таких случаев и делает создание многостраничных последовательностей операций с необязательными шагами более понятным и управляемым. Веб-потоки предлагают простой в использовании и настраиваемый до тонкостей механизм сохранения состояний, прекрасно подходящий для реализации последовательностей действий в веб-приложениях без применения тяжеловесных механизмов управления.

Посмотрим, как с помощью веб-потоков можно за полдня создать интернет-магазин Hubbub.

9.1. Что такое веб–поток?

В терминологии платформы Grails веб-поток – это последовательность логически связанных страниц, через которые должен пройти пользователь, чтобы достичь конечной точки назначения. Пользователи могут пропускать отдельные шаги в зависимости от данных, которые ими вводятся (например, когда пользователю не требуется указывать какой-то нестандартный способ доставки). В результате выбора различных параметров они могут оказываться на совершенно разных заключительных страницах (например, на странице, сообщающей об окончании заполнения заказа, или на странице, извещающей об отсутствии товара на складе).

Рассмотрим идеальный поток, состоящий из нескольких шагов, в виде блок-схемы. В нашем примере мы будем использовать приложение Hubbub и дадим пользователям возможность выбирать некоторые товары, имеющиеся в магазине, и вводить информацию о способе доставки. Затем мы проверим номер кредитной карты и переадресуем пользователя на страницу подтверждения заказа. На рис. 9.1 изображена первая версия алгоритма работы потока для интернет-магазина Hubbub. Сначала мы реализуем основной веб-поток, а затем на протяжении главы будем добавлять некоторые дополнительные более сложные шаги и подпотоки.

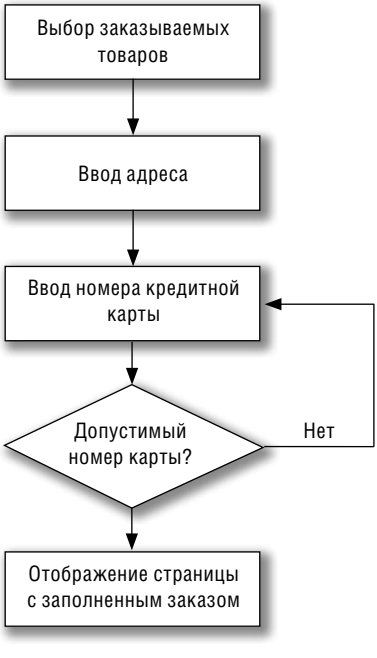

Рис. 9.1. *Блок-схема алгоритма работы основного веб-потока*

Наш первый поток достаточно прост, но даже на этом этапе видны несколько основных преимуществ использования механизма веб-потоков. Что такого дает нам веб-поток, что не может дать решение, основанное на стандартных контроллерах платформы Grails? Веб-поток обеспечивает следующие возможности:

- Запоминает, на какой стадии процедуры оформления заказа находится пользователь (вследствие чего действия кнопок `Next` (Далее) и `Previous` (Назад) зависят от контекста).
- Предоставляет простой предметно-ориентированный язык (Domain-Specific Language, DSL) для определения происходящего на каждом этапе процесса. Это позволяет легко увидеть, какой этап выполняется.
- Упрощает добавление новых шагов в процесс, освобождая от необходимости проводить реорганизацию большого количества методов действий контроллера.
- Обеспечивает широкие возможности в реализации принятия решений (например, на этапе проверки номера кредитной карты) и в выборе маршрута, на основе принятого решения.
- Вводит две новые области видимости: *потока* (*flow*) и *диалога* (*conversation*) – что позволяет создавать переменные, которые будут доступны только внутри потока и будут автоматически удаляться по завершении работы потока.

Веб-потоки в состоянии обеспечить весьма широкие возможности для любой части приложения, где необходимо реализовать взаимодействие, состоящее из нескольких шагов.

Но прежде чем приступать к использованию веб-потоков, вы должны понять, что они подходят не для всех типов приложений; основное их предназначение – реализация многоэтапных операций в стиле мастеров. Они абсолютно применимы для разбиения формы, содержащей 100 полей, на последовательность из нескольких страниц, или для реализации замысловатых алгоритмов, когда, например, требуется проверить номер кредитной карты, получить от пользователя адрес и способ доставки, а затем предложить ему страницу подтверждения заказа. Но не пытайтесь ограничить свой новый сайт в стиле Flickr взаимодействиями в стиле веб-потоков – вы потратите больше времени на борьбу с платформой, чем на решение насущных проблем.

9.1.1. Наш первый поток: мастер приема заказа

Теперь, когда у вас есть некоторое представление о ситуациях, в которых имеет и не имеет смысла применять веб-потоки, можно переходить к созданию нашего простого потока оформления заказа и изучению того, как взаимодействуют его составляющие.

Вы будете приятно удивлены, узнав, что все определения веб-потоков производятся в контексте стандартного контроллера. Начнем с созда-

ния в приложении Hubbub контроллера интернет-магазина, где пользователи смогут приобретать товары, имеющие отношение к сообществу Hubbub:

```
grails create-controller com.grailsinaction.shop
```

Создав контроллер `shop`, можно приступать к определению нашего первого потока. Для определения всех действий, изображенных на рис. 9.1, мы будем использовать язык Webflow DSL. Назовем наш поток `orderFlow`; в соответствии с соглашениями имена действий должны оканчиваться на «Flow» (поток). Это позволит платформе Grails автоматически определять действия, содержащие определения на языке Webflow DSL. Кроме того, потоки получают свой собственный адрес, то есть наш поток `orderFlow` будет иметь адрес URL /hubbub/shop/order.

Теперь можно определить поток. В листинге 9.1 определяется родительский элемент потока.

Листинг 9.1. Определение пустого потока, куда будут добавляться шаги процедуры

```
package com.grailsinaction

class ShopController {

    def index = {
        redirect(action: "order")
    }

    def orderFlow = {
    }

}
```

Это всего лишь заготовка. Обратите внимание, что мы включили в контроллер действие `index`. Как вы помните, если в контроллере имеется только одно действие, оно становится действием по умолчанию, но из-за специальной природы адресов URL данное правило к веб-потокам не применяется. Если вы хотите получить очевидный адрес URL (перенаправить адрес /shop на адрес /shop/order), вы должны взять дело в свои руки.

Обратите также внимание на то, что хотя мы присвоили потоку имя `orderFlow`, внутри контроллера мы должны обращаться к действию по имени без окончания «Flow»; именно поэтому перенаправление в действии `index()` выполняется по имени `"order"`, а не `"orderFlow"`.

В листинге 9.1 приводится определение потока, но он пока не выполняет никаких операций. Начинать определение потоков проще всего с создания скелета всего потока; затем можно наполнять его логикой. Без лишней суеты мы набросаем скелет определения потока, после чего на каждом шаге будем понемногу наращивать на нем мясо. В листин-

ге 9.2 приводится скелет определения потока, готовый к наполнению его некоторой логикой.

Листинг 9.2. Скелет определения потока

```
def orderFlow = {

    displayProducts {
        on("next") { ← Обработка кнопки Next (Далее) в форме
            // сохранить информацию о выбранных товарах
        }.to("enterAddress") ← Переход в состояние enterAddress
        on("cancel").to("finish")
    }

    enterAddress {
        on("next") {
            // сохранить адрес
        }.to("enterPayment")
        on("previous").to("displayProducts")
    }

    enterPayment {
        on("next") {
            // сохранить информацию о способе оплаты
        }.to("validateCard")
        on("previous").to("enterAddress")
    }

    validateCard {
        action { ← Действует, как точка ветвления
            // выполнить некоторые проверки
            if (params.validCard) {
                log.debug "Valid Card!!"
                valid()
            } else {
                log.debug "Invalid Card!!"
                invalid()
            }
        }
        on("valid").to("orderComplete")   ← Ветви алгоритма в зависимости от
        on("invalid").to("enterPayment")    значения, возвращаемого action
    }

    orderComplete {
        // display order
        on("finished").to("finish")
    }
    finish { ← Конечная точка потока
        redirect(controller:"homePage", action: "index")
    }

}
```

Хотя это всего лишь скелет процесса, здесь присутствует логика выполнения в стиле мастеров. Вы можете видеть, как каждое состояние веб-потока соответствует определенному этапу на блок-схеме (рис. 9.1). После каждого шага `on()` веб-поток выполняет переход к новому этапу `to()`. Чуть ниже мы займемся подробностями реализации, а пока посмотрите, можете ли вы уловить суть работы потока в целом, и насколько он соответствует нашей блок-схеме.

Теперь, когда вы имеете некоторое представление о том, как выполняется реализация потоков, можно поближе познакомиться с тем, как осуществляется переход от одного этапа к другому. Начнем с исследования определений `on().to()`.

Параметр execution: кнопка «Назад» и проблемы с постоянными ссылками

Начав работать с веб-потоками, вы заметите, как постоянно изменяется добавляемый в конец адреса URL параметр, определяющий состояние веб-потока (`execution=e1s1` со значением, которое изменяется при каждом изменении состояния).

Предположим, что пользователь создал закладку на страницу где-то в середине потока, или создал ссылку на нее где-то на внешнем ресурсе, или щелкнул на кнопке Назад броузера после того, как поток завершился, и попытался повторно отправить форму из середины потока. В таких случаях платформа Grails перехватит запрос к завершившемуся потоку или к потоку, время действия которого истекло, и перенаправит пользователя на первый этап потока. Если вы заглянете в журналируемые сообщения веб-потока (врезка «Отладка необычных ошибок веб-потоков»), вы увидите, как платформа перехватывает их:

```
[6431198] servlet.FlowHandlerAdapter Restarting a new execution of
          previously expired/ended flow 'order'
```

Как правило, возврат в начало потока – это самый лучший способ обработки таких ошибок. Если вам необходимо реализовать собственную логику обработки ошибок обращения к завершившимся потокам, проще всего будет это сделать с помощью фильтра, перехватывающего ошибку.

9.1.2. Анатомия состояния потока

Каждый шаг потока называется *состоянием*. Первое состояние, в котором оказывается веб-поток, называется *начальным состоянием* (*start state*), каковым в нашем случае является `displayProducts`.

При открытии страницы /shop/order платформа вызовет `displayProducts` для продолжения работы. Рассмотрим определение этого состояния:

```
displayProducts {
    on("next") {
        // сохранить информацию о выбранных товарах
    }.to("enterAddress")
    on("cancel").to("finish")
}
```

Как состояние `displayProducts` вызывает правила `"next"` и `"cancel"`? Ответ на этот вопрос лежит в плоскости соглашений о работе представления. В случае с состоянием `displayProducts` платформа Grails сначала отобразит страницу представления в файле /views/shop/order/displayProduct.gsp, а затем будет ждать, пока пользователь отправит форму. Поток продолжит свою работу в зависимости от того, на какой кнопке будет выполнен щелчок.

В листинге 9.3 приводится определение простой страницы `displayProduct`, с которой мы начнем. (Не забывайте, что она должна храниться в подкаталоге «order» в каталоге /views/shop, то есть мы находимся на один уровень ниже, чем обычно.)

Листинг 9.3. Выполнение шагов потока из представления

```
<html>
    <head>
        <title>Shop for Official Merchandise</title>
        <meta content="main" name="layout"/>
    </head>
    <body>

        Display Products Details Here
        <g:form action="order"> ← ❶ Имя формы соответствует имени потока
            <g:submitButton name="next" value="Next"/> ← ❷ Имя кнопки
                                                          соответствует имени перехода
            <g:submitButton name="cancel" value="Finished Shopping"/>
        </g:form>

    </body>
</html>
```

Выглядит, как обычная форма GSP платформы Grails, в которой присутствует всего несколько элементов, имеющих отношение к веб-потоку. Во-первых, обратите внимание, что в качестве имени формы выбрано имя `"order"`, а не `orderFlow` ❶. Представления в потоке всегда указывают на одно и то же действие, а механизм веб-потока определяет, в какое состояние следует перейти, опираясь на действия пользователя (с помощью параметра `execution`, который добавляется веб-потоком к каждому адресу URL, о чем говорилось во врезке выше, где обсуждался параметр `execution`).

Во-вторых, обратите внимание на особую важность имен кнопок ❷. Имя кнопки определяет соответствующее действие в реализации, поэтому будьте внимательны при их написании.

Заключительная часть действия – предложение .to(), которое сообщает веб-потоку, к какому состоянию следует перейти далее. Когда пользователь щелкает на кнопке Cancel (Отменить), весь механизм приходит в движение. Форма отправляется на сервер, веб-поток следует за предложением on() и оказывается в конечной точке, где производится переход на домашнюю страницу. В листинге 9.4 демонстрируется реализация перехода в потоке.

Листинг 9.4. Предложение on() отображает кнопку Cancel (Отменить) на следующее логическое действие

```
displayProducts {
    on("next") {
        // сохранить информацию о выбранных товарах
    }.to("enterAddress")
    on("cancel").to("finish")
}

finish { ← Завершает работу потока, так как здесь опущено предложение to()
    redirect(controller:"homePage", action: "index")
}
```

Обратите внимание на отсутствие предложений .to() у состояния finish. Любое состояние, в котором отсутствует предложение .to(), называется *конечным состоянием* (*end state*). Такое состояние является специальным случаем состояний, поскольку после перехода в конечное состояние уничтожаются все данные о состоянии потока (например, информация об этапе, на котором находился пользователь, и все данные в области видимости потока, о которой мы поговорим ниже).

В нарушение соглашений: отображение собственных представлений

Имена представлений веб-потоков следуют стандартным соглашениям (имя файла представления соответствует названию состояния), но вы легко можете преодолеть это ограничение в случае необходимости, воспользовавшись методом render(). Например, состояние displayProducts в листинге 9.4 могло бы вызывать представление с другим именем, как показано ниже:

```
displayProducts {
    render(view: "standardProducts")
    on("next") {
        // capture products
    }.to("enterAddress")
    on("cancel").to("finish")
}
```

9.2. Работа с веб-потоками

Теперь, когда вы немного представляете, как веб-потоки обрабатывают состояния и выбирают маршруты, можно попробовать применить эти знания для реализации нашего потока оформления заказа. В данном разделе вы изучите основы реализации потоков:

- Как сохранять объекты в области видимости потока
- Как выполнять привязку данных и обработку ошибок в потоках
- Как программно реализовывать состояния с принятием решения и как программно выбирать маршрут дальнейшего движения к различным состояниям

Начнем с исследования области видимости потока – новой области видимости, предоставляемой платформой Grails для хранения объектов, которые должны быть доступны только на протяжении времени существования веб-потока.

9.2.1. Область видимости потока: лучше, чем область видимости кадра, и дешевле, чем область видимости сеанса

Веб-потоки платформы Grails привносят с собой новую область видимости: *область видимости потока* (*flow scope*). Данные, находящиеся в этой области видимости, продолжают существовать, пока существует сам поток (то есть пока пользователь не попадет в конечное состояние, подобное состоянию `finish` в предыдущем примере). Область видимости потока обеспечивает эффективный способ хранения и использования данных на разных этапах потока, и она отлично подходит для хранения объектов, создаваемых внутри потока. В примере с приложением Hubbub мы могли бы сохранять товары, отобранные пользователем на первом этапе потока, и использовать их на этапе подтверждения заказа и выставления счета.

Находясь в одном из состояний потока, вы можете обращаться к области видимости потока явно, как и к другим областям видимости. Например:

```
displayProducts {
    on("next") {
        flow.order = new Order(params)
    }.to("enterAddress")
    on("cancel").to("finish")
}
```

← Явно сохраняет объект в области видимости потока

Это явный способ обращения к области видимости потока, но существует еще более удобная возможность. Когда объекты возвращаются состоянием потока (в виде отображения, как это обычно принято в контрол-

лерах), они автоматически сохраняются в области видимости потока. Например, можно написать такую реализацию:

```
displayProducts {
    on("next") {
        def orderDetails = new Order(params)
        [ order: orderDetails, orderStartDate: new Date() ]
    }.to("enterAddress")
    on("cancel").to("finish")
}
```

Сохраняет объект в области видимости потока

Объекты `orderDetails` и `orderStartDate` автоматически будут сохранены в области видимости потока. Но эта область видимости имеет собственные ограничения, и они весьма существенны. Самое серьезное ограничение состоит в том, что все объекты, сохраняемые в области видимости потока, должны поддерживать интерфейс `Serializable`.

Как быть с интерфейсом Serializable?

Требование поддержки интерфейса `Serializable` всеми объектами, сохраняемыми в области видимости потока, означает, что объекты простых типов, таких как `Strings` и `Date`, без исключений могут сохраняться в области видимости потока, но если потребуется сохранять объекты предметной области, они должны реализовать интерфейс `Serializable`. Хуже того: чтобы удовлетворить требования, предъявляемые интерфейсом `Serializable`, все классы предметной области, связанные отношениями с данным классом, также должны реализовать интерфейс `Serializable`. Это может повлечь необходимость внесения массы изменений!

Если прежде вам не приходилось сталкиваться с интерфейсом `Serializable` в языке Java, заметим, что вам следует начать с изучения документации Javadoc для `java.io.Serializable`. В большинстве случаев использования веб-потоков вам достаточно будет просто добавить `implements Serializable` к определениям своих классов предметной области. Однако с интерфейсом `Serializable` связаны некоторые сложные, трудно обнаруживаемые ситуации, которые могут приводить в замешательство начинающих разработчиков приложений на платформе Grails, – особенно когда появляется необходимость сохранять объекты, поддерживающие интерфейс `Serializable`, в коллекциях.

Многие элементы, которые вам придется сохранять в области видимости потока, являются объектами, созданными ранее, и вам едва ли захочется возвращаться обратно и отмечать все классы предметной области как поддерживающие интерфейс `Serializable` только для того,

чтобы получить возможность хранить их в области видимости потока. К счастью, управляющие объекты, которые можно объявить с поддержкой интерфейса Serializable, предлагают неплохое решение этой проблемы.

Продолжим наше исследование веб-потоков, чуть больше углубившись в практические приемы проверки данных в веб-потоках.

9.2.2. Стратегии привязки и проверки данных

Мы обсудили некоторые сложности сохранения объектов данных в области видимости потока, но веб-потоки предоставляют нам ряд других возможностей обработки данных форм. Одной из самых замечательных является поддержка потоками управляющих объектов. Мы рассматривали управляющие объекты в главе 5, когда изучали способы обработки форм, – они представляют собой отличное решение текущей проблемы, позволяющее избежать необходимости реализовать интерфейс Serializable в классах предметной области.

Допустим, что наше представление displayProducts дает возможность указать количество приобретаемых футболок и бейсболок. Реализуется это просто, как показано ниже:

```
<g:form action="order">
    Shirts: <g:textField name="numShirts" value="0"/>
    Hats: <g:textField name="numHats" value="0"/>
    <g:submitButton name="next" value="Next"/>
    <g:submitButton name="cancel" value="Finished Shopping"/>
</g:form>
```

Нам требуется гарантировать, что пользователь не сможет заказать более 10 футболок или 10 бейсболок. Напоминает классическую проблему проверки ограничений, правда? В листинге 9.5 приводится пример поддерживающего интерфейс Serializable объекта, который определяет диапазон количества заказываемых предметов от 0 до 10; сюда же мы можем добавить любые другие проверки, соответствующие решаемой задаче.

Листинг 9.5. Управляющие объекты, используемые в потоке, должны поддерживать интерфейс Serializable

```
class OrderDetailsCommand implements Serializable {

    int numShirts
    int numHats

    boolean isOrderBlank() {
        numShirts == 0 && numHats == 0
    }

    static constraints = {
```

```
            numShirts(range: 0..10)
            numHats(range: 0..10)
        }
    }
```

В листинге 9.6 показано, как можно привязать управляющий объект к состоянию `displayProducts`.

Листинг 9.6. Отображение управляющего объекта в этап потока

```
displayProducts {
    on("next") { OrderDetailsCommand odc ->  ❶ Привязка данных формы к объекту
        if (odc.hasErrors() || odc.isOrderBlank()) {  ❷ Проверка правильности
            flow.orderDetails = odc    ❸ Сохранение недопустимого
                                          заказа в области видимости потока
            return error()  ❹ Останавливает продвижение пользователя по потоку
        }
        [ orderDetails: odc, orderStartDate: new Date() ]
    }.to("enterAddress")
    on("cancel").to("finish")
}
```

Как и обычные действия (вне веб-потока), состояния веб-потока, выполняющие привязку к управляющему объекту, должны определить этот объект в первом аргументе замыкания ❶. Механизм привязки данных в платформе Grails передает параметры запроса в управляющий объект и устанавливает свойства объекта `errors`, если были обнаружены несоответствия установленным ограничениям ❷.

Единственное, с чем вы еще ранее не встречались, – это вызов метода `error()` ❹. Возврат значения методом `error()` предписывает веб-потоку вернуться к предыдущему представлению, чтобы пользователь мог исправить обнаруженные ошибки, но тогда пользователь должен иметь доступ к информации об ошибках, чтобы исправить их. Для этого мы сохраняем управляющий объект в области видимости потока ❸, в итоге мы получаем возможность использовать коллекцию ошибок для отображения на странице представления.

Все это приобретет для вас больше смысла, когда вы увидите эту реализацию в действии. В листинге 9.7 приводится измененная версия представления `displayProducts`, где на этот раз предусмотрена обработка ошибок.

Листинг 9.7. Измененная версия представления displayProducts с обработкой ошибок

```
<g:hasErrors bean="${orderDetails}">
    <div class="errors">
        <g:renderErrors bean="${orderDetails}"/>
    </div>
</g:hasErrors>
```

```
<g:form action="order">
    Shirts: <g:textField name="numShirts"
                  value="${orderDetails?.numShirts}"/>
    Hats: <g:textField name="numHats" value="${orderDetails?.numHats}"/>
    <g:submitButton name="next" value="Next"/>
    <g:submitButton name="cancel" value="Finished Shopping"/>
</g:form>
```

Обратите внимание, что теперь мы отображаем все ошибки, которые присутствуют в объекте `orderDetails`, сохраненном в области видимости потока. Кроме того, мы предусмотрели заполнение полей значениями свойств объекта, чтобы показать пользователю ошибочные значения. На рис. 9.2 показаны сообщения об ошибках в действии.

Это именно то, что нам требовалось. Вы могли заметить, что кроме всего прочего была выполнена настройка сообщений об ошибках. Для этого была использована поддержка механизмом интернационализации (i18n) пакетов сообщений (представленная в разделе 4.1.2), которая доступна и в веб-потоках. Для создания сообщений, представленных на рис. 9.2, мы добавили в файл /grails-app/i18n/messages.properties следующие строки:

```
OrderDetailsCommand.numShirts.range.toosmall=
➥ You may not order less than {3} shirts.
OrderDetailsCommand.numShirts.range.toobig=
➥ You may not order more than {4} shirts.
OrderDetailsCommand.numHats.range.toosmall=
➥ You may not order less than {3} hats.
OrderDetailsCommand.numHats.range.toobig=
➥ You may not order more than {4} hats.
```

Примечание

Чтобы освежить в памяти знания о поддержке механизмом интернационализации (i18n) пакетов сообщений, вернитесь к главе 4.

Теперь, когда мы реализовали в нашем потоке проверку накладываемых ограничений и сохранение управляющих объектов в области видимости потока, можно поближе познакомиться с состояниями потока, которые вообще не имеют пользовательского интерфейса: с *состояниями действий*.

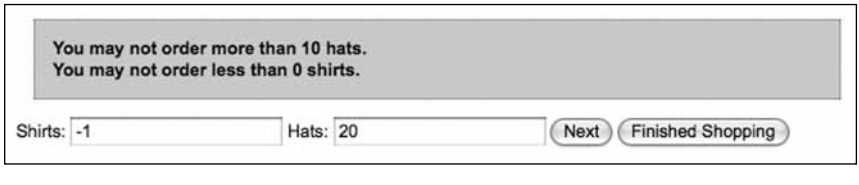

***Рис. 9.2.** Наша форма отображает сообщения об ошибках и заполняет поля ошибочными данными*

9.2.3. Принятие решений программным способом в состояниях действий

До сих пор все наши решения управлялись пользовательским интерфейсом: пользователь щелкал на кнопке Next (Далее) или Previous (Назад), или форма содержала недопустимые данные. Но мы пока не рассматривали возможность программного управления направлением движения потока. Нам может потребоваться проверить номер кредитной карты или предложить специальные способы доставки товаров для пользователей, проживающих на большом удалении.

У подобных этапов потока нет собственного пользовательского интерфейса, зато есть возможность изменять направление движения потока на основе результатов внутренней обработки данных. В контексте веб-потоков такие этапы называют *состояниями действий* (*action states*), чтобы отличать их от более типичных *состояний с переходами* (*transition states*), с которыми мы имели дело до сих пор.

Возьмем в качестве примера реализацию проверки кредитной карты. После того как пользователь укажет информацию о своей кредитной карте, нам необходимо проверить ее с помощью одной из служб электронных платежей. Если пользователь указал правильный номер кредитной карты и баланс позволяет выполнить покупку, поток может продолжать работу, но если кредитная карта не проходит проверку, нам следует вернуть пользователю форму ввода номера кредитной карты. Реализовать подобную логику работы потока можно с помощью состояния действия.

Реализуем этап простой проверки кредитной карты, который будет терпеть неудачу, если пользователь вводит информацию утром. Нам требуется получить результат, подобный показанному на рис. 9.3.

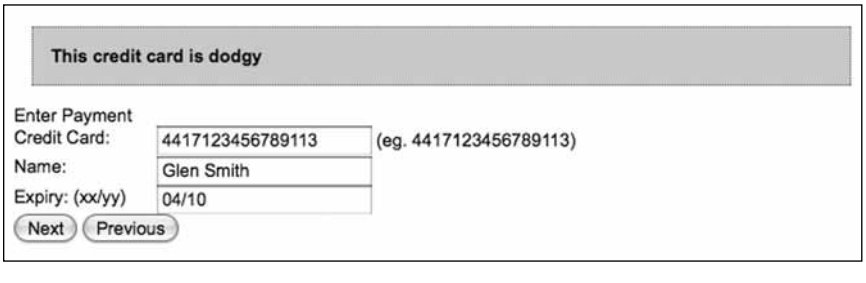

Рис. 9.3. *Потоки поддерживают возможность переопределения сообщений об ошибках с помощью пакетов сообщений*

Форма, как приведенная ниже, проходит обычную последовательность представление-состояние в процессе проверки значений полей, что может быть реализовано так:

```
enterPayment {
    on("next") { PaymentCommand pc ->
        flow.pc = pc
        if (pc.hasErrors()) {
            return error()
        }
    }.to("validateCard")
    on("previous").to("enterAddress")
}
```

Здесь выполняются основные проверки, и если все в порядке, совершается переход к новому состоянию действия: `validateCard`.

Определение состояния действия начинается с ключевого слова `action`, благодаря чему платформа Grails может определить, что для данного состояния не требуется отображать представление. В остальном тело состояния потока похоже на любое другое состояние. Обычно состояния действий определяют и возвращают собственные имена переходов, что делает тело потока более удобочитаемым. В листинге 9.8 мы возвращаем значение `invalid()` или `valid()`, в зависимости от результатов проверки кредитной карты.

Листинг 9.8. Реализация состояния действия, выполняющего проверку кредитной карты

```
validateCard {
    action {
        def validCard = new Date().hours > 11 // Требуется время после полудня
        if (validCard) {
            valid()
        } else {
            flow.pc.errors.rejectValue("cardNumber",
                "card.failed.validation",
                "This credit card is dodgy")
            invalid()
        }
    }
    on("valid").to("orderComplete")
    on("invalid").to("enterPayment")

}
```

← Отвергает кредитную карту с собственным кодом ошибки

В листинге 9.8, когда проверка терпит неудачу, мы добавляем собственный объект ошибки в управляющий объект `PaymentCommand` (`pc`), находящийся в области видимости потока. Это позволит представлениям использовать тег `hasErrors`, предлагаемый платформой Grails, для отображения текста нашего сообщения.

Обсуждение выше достаточно полно описывает, что делают состояния действий и как их использовать. Состояния действий прекрасно подходят для представления сложной логики принятия решения без вмеше-

ния в работу действия с переходом, а также для организации ветвления и запуска подпотоков, с которыми мы познакомимся вскоре.

Теперь вы знакомы со всеми основными особенностями веб-потоков, которые вы, вероятно, захотите применить при создании своего следующего приложения на платформе Grails. В следующем разделе мы познакомим вас с некоторыми дополнительными концепциями веб-потоков – на практике они используются не так часто, но их удобно использовать в некоторых ситуациях.

9.3. Дополнительные особенности веб-потоков

Теперь у вас должно иметься четкое представление об основах реализации прикладных потоков, однако веб-потоки обладают еще целым рядом дополнительных особенностей, которые могут пригодиться в определенных ситуациях. В этом разделе мы исследуем некоторые из них, включая подпотоки и диалоги. Начнем со знакомства со службами в области видимости потока.

9.3.1. Службы области видимости потока

К настоящему моменту мы изучили, как сохранять в области видимости потока объекты, обладающие поддержкой интерфейса `Serializable`, для того чтобы их можно было передавать из одного состояния потока в другое. Но мы пока не знакомы с возможностью использования служб области видимости потока, которые можно рассматривать как легковесный эквивалент компонента сеанса JavaBean с сохранением состояния.

Службы области видимости потока удобно использовать, когда необходимо получить объект службы, который будет использоваться только в пределах потока, и управлять данными, имеющими отношение только к текущему потоку. Хорошим примером такой службы может служить функция проверки кредитной карты. В начале потока мы выполняем проверку кредитной карты, а затем мы могли бы получить доступ к коду ошибки или к проверенному номеру карты, сохраненному в пределах службы.

Приведем в порядок наш предыдущий пример и поэкспериментируем с созданием службы области видимости потока, выполняющей проверку кредитной карты. Службы области видимости потока похожи на обычные службы и создаются стандартным способом:

```
grails create-service com.grailsinaction.CreditCard
```

Все точно так же, как и при создании любой другой службы. Но в отличие от случая со стандартной службой, необходимо явно указать область видимости службы с помощью атрибута `scope`:

```
class CreditCardService implements Serializable {
```

```
    static scope = "flow"
    static transactional = false
}
```

Как и все объекты, сохраняемые в области видимости потока, данные службы должны поддерживать интерфейс `Serializable`. Когда служба определяется таким способом, к контроллерам, в которых используется служба, применяются стандартные правила внедрения служб (вернитесь к главе 4, если вам требуется освежить в памяти информацию о службах).

Следует отметить, что хотя объявление службы находится в контроллере (а не в определении потока на языке DSL), каждый новый экземпляр потока будет получать новый экземпляр внедренной службы – новый объект службы создается для каждого потока, а не для контроллера.

Ниже приводится измененная версия контроллера `ShopController` с внедренной службой:

```
class ShopController {

    def creditCardService

    // ... остальные действия контроллера
}
```

После того как служба будет объявлена в контроллере, ее использование внутри любого состояния потока будет гарантировать, что обращения будут выполняться к уникальной службе, созданной специально для этого потока. Следовательно, вы можете применять службы в состояниях действий. В листинге 9.9 приводится переделанная реализация действия `validateCard`, в которой теперь используется служба области видимости потока.

Листинг 9.9. Вызов службы с сохранением состояния для проверки кредитной карты

```
validateCard {
    action {
        def validCard = creditCardService.checkCard(
            flow.pc.cardNumber,                          ← Вызов службы области
            flow.pc.name,                                  видимости потока
            flow.pc.expiry)
        if (validCard) {
            valid()
        } else {
            flow.pc.errors.rejectValue("cardNumber",
                "card.failed.validation",
                "This credit card is dodgy")
            invalid()
        }
    }
```

```
        on("valid").to("orderComplete")
        on("invalid").to("enterPayment")
    }
```

Обратите внимание, что мы просто вызываем службу `creditCardService`, а о сохранении состояния службы в области видимости потока заботится сама платформа Grails. Не требуется никакого волшебства с `flow.creditCardService`.

Самое большое преимущество служб области видимости потока состоит в том, что в последующих состояниях потока вы можете благополучно обращаться к данным, сохраненным службой. Такая свобода означает, что мы можем реализовать службу `CreditCardService` как компонент с сохранением состояния и пребывать в уверенности, что у нас никогда не возникнут конфликты между состояниями веб-потока, использующими эту службу. В листинге 9.10 приводится базовая реализация службы с сохранением состояния.

Листинг 9.10. Абстрактная проверка кредитной карты в службе с сохранением состояния

```
package com.grailsinaction

class CreditCardService implements Serializable {  <-- Гарантирует реализацию службой интерфейса Serializable

    static scope = "flow"  <-- Отмечает как службу области видимости потока
    static transactional = false

    boolean cardChecked = false
    String cardNumber
    String name
    String expiry
    Date checkedAt

    boolean checkCard(String aNumber, String aName, String anExpiry) {
        (cardNumber, name, expiry) = [ aNumber, aName, anExpiry ]
        log.debug "Validating ${cardNumber} for ${name} with expiry ${expiry}"

        // вызвать удаленную службу здесь
        checkedAt = new Date()
        cardChecked = true
        println "Card is ${cardChecked}"
        return cardChecked

    }
}
```

Как объяснялось в главе 4, службы без сохранения информации о состоянии создаются обычно потому, что по умолчанию они используются единственными экземплярами объектов. Но в листинге 9.10 выполняется сохранение информации о кредитной карте в самой службе. Позд-

нее в потоке мы сможем безопасно обращаться к этим данным или вызывать другие методы службы, использующие эти же данные, будучи уверенными, что никакой другой объект не изменит их к тому времени.

Поскольку службы области видимости потока сохраняются в потоке, они обязаны поддерживать интерфейс `Serializable`, как и любые другие объекты, сохраняемые в потоке. Этим они отличаются от стандартных служб; вы знаете, что делать при появлении ошибки, гласящей: «object in flow state could not be serialized» («объект не может быть сохранен в потоке»).

> **Как быть, если мне потребуется служба, которая могла бы использоваться и в области видимости потока, и в области видимости контроллера?**
>
> Мы знаем, о чем вы размышляете, – вы хотите использовать службу в области видимости потока, но при этом иметь возможность вызывать некоторые ее методы из стандартного контроллера. Очень жаль, но это практически всегда говорит о том, что вы что-то недостаточно продумали.
>
> Основное назначение служб области видимости потока заключается в сохранении информации о состоянии. Если у вас возникает желание смешать методы с сохранением и без сохранения состояния, то вы, скорее всего, подразумеваете методы, которые должны принадлежать двум разным классам служб.
>
> Одно из решений данной проблемы состоит в том, чтобы выделить в отдельную службу программный код, предполагающий сохранение информации о состоянии в области видимости потока, а потом внедрить в нее службу без сохранения состояния. Это означает, что вы сможете многократно использовать логику службы без сохранения состояния, использовать компонент с сохранением состояния и одновременно остаться верным принципу DRY (Don't Repeat Yourself – избегай повторений).

9.3.2. Подпотоки и диалоги

До сих пор мы рассматривали возможности перехода от состояния к состоянию и ветвления в пределах единственного потока. Но по мере разработки потока все усложняется. Вам может потребоваться выполнять переходы между различными частями потока, ваши процессы могут принимать весьма запутанный вид, и тогда область видимости потока может начать напоминать свалку. Для таких ситуаций платформа Grails предоставляет возможность создавать подпотоки, которые позволяют реализовать некоторую логику работы в виде отдельного, самостоятельного потока.

Подпотоки обладают следующими преимуществами:

- Подпотоки можно многократно использовать в нескольких родительских потоках.
- Область видимости потока становится более упорядоченной, потому что она доступна только подпотоку и очищается по завершении его работы.
- Группировка логики работы в подпотоки делает основные потоки более короткими и более самодостаточными, а реализацию – более удобной в сопровождении.
- Вы можете обеспечить совместное использование данных родительскими и дочерними потоками, поэтому вам не придется дублировать данные потока.

Наша процедура вычисления стоимости посылки и ее доставки – отличный кандидат на выделение ее в отдельный подпоток. Мы дадим пользователю возможность выбирать наиболее подходящий для него способ доставки, и если такой выбор будет сделан, мы будем отправлять пользователя в подпоток, обрабатывающий различные параметры доставки. На рис. 9.4 изображен измененный фрагмент нашей блок-схемы (из рис. 9.1), где после этапа ввода адреса появился новый подпоток.

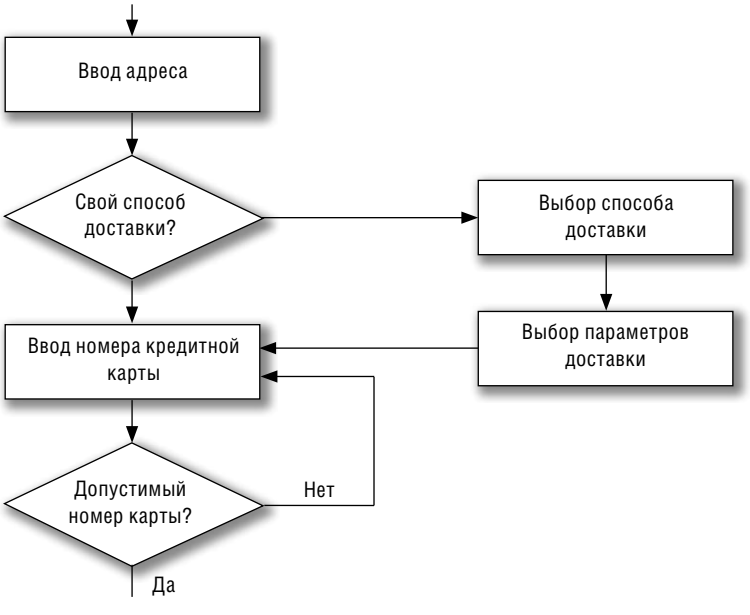

Рис. 9.4*. Новый подпоток, обрабатывающий параметры доставки, после этапа ввода адреса*

На странице ввода адреса пользователю предоставляется возможность установить флажок Custom Shipping (Свой способ доставки); мы будем ис-

пользовать его, чтобы запустить подпоток. Для сохранения информации о способе доставки нам потребуется управляющий объект: определение этого класса `ShippingCommand` приводится в листинге 9.11. Ограничения для данных мы добавим позднее, а пока будем использовать его для простой передачи данных.

Листинг 9.11. Объект ShippingCommand, используемый для сохранения информации о способе доставки

```
class ShippingCommand implements Serializable {

    String address
    String state
    String postcode
    String country
    boolean customShipping
    String shippingType
    String shippingOptions

}
```

Теперь реализуем запуск подпотока из основного потока `orderFlow`. Нам потребуется обеспечить совместное использование информации о способе доставки родительским и дочерним потоками. В настоящий момент данные в виде объекта `ShippingCommand` сохраняются в области видимости потока, но область видимости основного потока недоступна подпотокам. Поэтому нам необходимо создать *область видимости диалога* (*conversation scope*), где будут храниться данные, совместно используемые текущим потоком и всеми подпотоками. Внесем изменения в реализацию и сохраним информацию о способе доставки в области видимости диалога, как показано в листинге 9.12.

Листинг 9.12. Сохранение данных в области видимости диалога обеспечивает возможность передачи данных подпотокам

```
enterAddress {
    on("next") { ShippingCommand sc ->
        conversation.sc = sc ← Помещает управляющий объект в область
        if (sc.hasErrors()) {   видимости диалога
            return error()
        }
    }.to("checkShipping")
    on("previous").to("displayProducts")
}
```

Сохранив информацию о способе доставки в области видимости диалога, можно приступать к реализации ветвления на тот случай, если у пользователя появятся свои требования к способу доставки. Это отличный шанс применить на практике наши знания о действиях потока. В листинге 9.13 приводится реализация действия `checkShipping`, в котором сосредоточена логика выбора способа доставки.