Основные паттерны J2EE
View Helper

Ситуация

Система создает содержимое презентации, которое требует обработки динамических бизнес-данных.

Задача

В тех случаях, когда логика доступа к бизнес-данным и логика форматирования презентации перемешаны, часто производятся изменения в ярусе презентации, которые сложно разрабатывать и поддерживать. Это делает систему менее гибкой, менее подходящей для повторного использования, то есть, в целом, менее гибкой для изменений.

Смешение бизнес- и системной логики с обработкой вида приводит к уменьшению модульности, а также к слабому разделению ролей между командами Web-производства и разработчиков программного обеспечения.

Требования

- Требования к ассимиляции бизнес данных не тривиальны.
    - Встраивание бизнес логики в вид способствует буферному типу повторного использования. Это приводит к проблемам в процессе поддержки и багам, так как часть логики используется повторно (путем простого копирования) в том же или другом виде.
    - Желательно обеспечить чистое разделение труда, при котором роли разработчика программного обеспечения должны отделяться от ролей члена команды Web-производства.
    - Обычно каждому виду соответствует отдельный бизнес-запрос.

Решение

Вид содержит код форматирования, передающий полномочия обработчика классам хелпера, реализованным как JavaBean-компоненты или заказные тэги. Хелперы также хранят промежуточную модель данных вида и функционируют как адаптеры бизнес-данных.

Для реализации компонента вида существует множество стратегий. Стратегия JSP View предлагает в качестве компонента вида использовать JSP-страницу. Данная стратегия наиболее предпочтительна и используется чаще всего. Другой принципиально отличающейся стратегией является Servlet View, в которой сервлет используется как вид (за дополнительной информацией обращайтесь к разделу «Стратегии»).

Инкапсуляция бизнес-логики в хелпере, а не в виде, делает приложение более модульным и упрощает повторное использование компонентов. Многочисленные клиенты, такие как контроллеры и виды, могут помогать тому же хелперу извлекать и адаптировать похожее состояния модели для представления ее различными способами. Единственным способом повторно использовать логику, встроенную в вид, является копирование и вставка из буфера. Более того, такое дублирование кода усложняет поддержку системы, так как один и тот же баг нужно исправлять сразу в нескольких местах.

Преобладание кода скриптлета над JSP видом может служить сигналом к применению данного паттерна к существующему коду. Главной целью применения данного паттерна является разбиение бизнес логики вне вида. Одну часть логики лучше всего инкапсулировать в объектах хелпера, а другую поместить в централизованный

компонент, который расположен перед видами и хелперами. Он может включать в себя общую для всех запросов логику (например, проверку аутентификации или службы входа в систему). Дополнительную информацию по данному вопросу можно найти в главах «Intercepting Filter» и «Front Controller».

Если отдельный контроллер не задействован в архитектуре или не используется для обработки всех запросов, то компонент вида становится начальной точкой контакта для обработки некоторых запросов. Данный сценарий хорошо подходит для запросов с минимальной обработкой. Чаще всего такая ситуация встречается на страницах, в основе которых лежит статическая информация. Такой страницей может быть первая страница в серии, служащая для сбора некоторой информации (смотрите главу «Dispatcher View»). Кроме того, данный сценарий применим в тех случаях, когда механизм используется для создания составных страниц (см. раздел «Composite View»).

Паттерн View Helper фокусируется на рекомендациях о том, как распределять обязанности в приложении. О направлении клиентских запросов напрямую к виду можно прочитать в главе «Dispatcher View».

Structure

На рисунке 7.11 представлена классовая диаграмма паттерна View Helper.

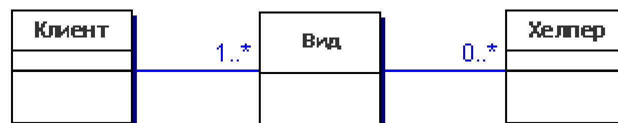

Рисунок 7.11 Классовая диаграмма View Helper

Участники и обязательства

На рисунке 7.12 представлена циклограмма паттерна View Helper. Контроллер обычно является промежуточным звеном между клиентом и видом. Тем не менее, в некоторых случаях контроллер не используется. Поэтому при обработке запроса виды становятся начальной точкой контакта. (Смотрите также главу «Dispatcher View»).

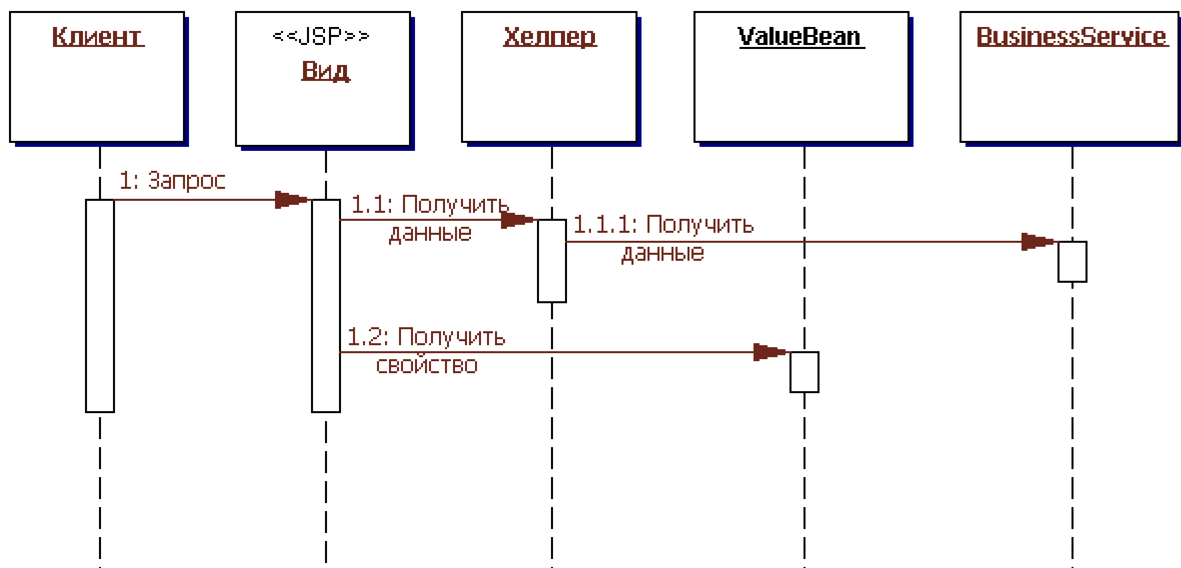

Рисунок 7.12 Циклограмма View Helper

Как видно из классовой диаграммы, ассоциированных с видом хелперов может и не быть. В этом простом случае страница может быть полностью статической или же

включать совсем небольшое количество линейного кода скриптлета. Данный сценарий описан в циклограмме на рисунке 7.13.

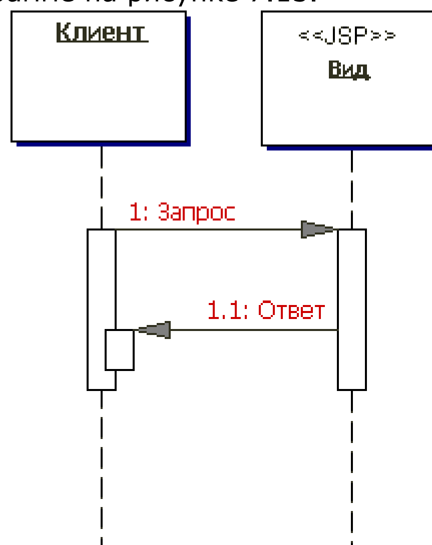

Рисунок 7.13 Простая циклограмма View Helper

Вид

Вид формулирует и отображает клиенту информацию, извлекаемую из модели. Хелперы поддерживают виды путем инкапсуляции и адаптации модели для использования ее в отображении.

Помощник

Хелпер отвечает за содействие в выполнении обработки вида или контроллера. Таким образом, хелперы выполняют несколько функций, куда входит сбор данных, необходимых для вида и сохранение промежуточной модели. В этом случае на хелпер иногда ссылаются, как на компонент-значение. Кроме этого, хелперы могут адаптировать модель данных для использования ее в виде. Хелперы могут обслуживать запросы данных из вида путем обычного предоставления доступа к необработанным данным или путем форматирования данных как Web-содержимого.

Вид может взаимодействовать с любым количеством хелперов, которые обычно реализованы как JavaBean-компоненты (JSP 1.0+) и заказные тэги (JSP 1.1+). Кроме того, хелпер может отображать объект Command, delegate (см. главу «Business Delegate») или XSL преобразователь. Последний для приведения модели к подходящему виду используются в комбинации с таблицей стилей.

ValueBean

ValueBean – это другое название хелпера, отвечающего за удерживание состояния промежуточной модели для использования ее видом. Чаще всего, как показано в циклограмме на рисунке 7.12, бизнес-служба возвращает компонент-значение (value bean) в ответ на запрос. В этом случае ValueBean выполняет роль Transfer Object (см. главу «Transfer Object»).

BusinessService

BusinessService является ролью, выполняемой службой, к которой клиент стремится получить доступ. Обычно доступ к бизнес-службе (business service) организован через Business Delegate. Ролью business delegate является предоставление управления бизнес-службе и ее защита (см. главу «Business Delegate»).

Стратегии

Стратегия JSP View

Стратегия JSP View предлагает использовать в качестве компонента вида JSP-страницу. Данная стратегия более предпочтительна, чем Servlet View. Не смотря на семантическую эквивалентность стратегии Servlet View, стратегия JSP View является более изящным решением и гораздо чаще используется. Виды являются сферой деятельности Web-дизайнеров, которые предпочитают визуальную разметку Java-коду. В примере 7.17 показан пример кода, для которого применена данная стратегия. Он является выдержкой из файла `welcome.jsp`, которому контроллер сервлета отправляет данные после помещения JavaBean-компонента WelcomeHelper в область запроса.

Пример 7.17 Пример кода стратегии JSP View

```
<jsp:useBean id="welcomeHelper" scope="request"
  class="corepatterns.util.WelcomeHelper" />

<HTML>
<BODY bgcolor="FFFFFF">
<% if (welcomeHelper.nameExists())
{
%>
<center><H3> Welcome <b>
<jsp:getProperty name="welcomeHelper" property="name" />
</b><br><br> </H3></center>
<%
}
%>

<H4><center>Glad you are visiting our
  site!</center></H4>

</BODY>
</HTML>
```

Альтернативная стратегия Servlet View обычно реализуется путем прямого внедрения HTML-разметки в код Java-сервлета. Совмещение Java-кода с тэгами разметки приводит к слабому разделению ролей пользователя в проекте и увеличивает зависимости от одних и тех же ресурсов среди многочисленных членов разных команд. При работе с шаблоном, содержащим неизвестные тэги или код, увеличивается вероятность случайных изменений, что приводит к ошибкам в системе. Кроме того, наблюдается снижение производительности в рабочей среде (так как слишком многие пользуются одним и тем же физическим ресурсом) и усложнение управления источниками.

Эти проблемы с наибольшей вероятностью могут возникать в больших коммерческих структурах, которые предъявляют более сложные системные требования, и в которых работают целые команды разработчиков. Вероятность того, что упомянутые проблемы возникнут при работе с малыми системами (с простыми бизнес-требованиями и небольшим коллективом разработчиков) невелика, так как несколько ролей (описанных ранее) может выполнять один человек. Тем не менее, нужно иметь в виду, что все серьезные фирмы начинаются с малых проектов.

Стратегия Servlet View

Стратегия Servlet View использует в качестве вида сервлет. Она семантически эквивалентна стратегии JSP View. Тем не менее, как видно из примера 7.18, стратегия Servlet View часто бывает более громоздкой для разработчиков программного обеспечения и команд Web-производства, так как при ее использовании тэги разметки включены прямо в Java-код. Шаблон вида при этом гораздо сложнее обновлять и изменять.

Пример 7.19 Пример кода стратегии Servlet View

```
public class Controller extends HttpServlet {
  public void init(ServletConfig config) throws
    ServletException {
```

```
    super.init(config);
  }

  public void destroy() { }

  /** Обработка запросов для HTTP методов
   * <code>GET</code> и <code>POST</code>.
   * @param запрос, запрос сервлета
   * @param ответ, ответ сервлета
   */
  protected void processRequest(HttpServletRequest
    request, HttpServletResponse response)
    throws ServletException, java.io.IOException {
    String title = "Servlet View Strategy";
    try {
      response.setContentType("text/html");
      java.io.PrintWriter out = response.getWriter();
      out.println("<html><title>"+title+"</title>");
      out.println("<body>");
      out.println("<h2><center>Employees List</h2>");
      EmployeeDelegate delegate =
          new EmployeeDelegate();

      /** ApplicationResources обеспечивает простой API
        * для извлечения констант и других
        * предустановленных значений**/
      Iterator employees = delegate.getEmployees(
            ApplicationResources.getInstance().
                getAllDepartments());
      out.println("<table border=2>");
      out.println("<tr><th>First Name</th>" +
        "<th>Last Name</th>" +
          "<th>Designation</th><th>Id</th></tr>");
      while (employees.hasNext())                       {
        out.println("<tr>");
        EmployeeTO emp = (EmployeeTO)employees.next();
        out.println("<td>"+emp.getFirstName()+
            "</td>");
        out.println("<td>"+emp.getLastName()+
            "</td>");
        out.println("<td>"+emp.getDesignation()+
            "</td>");
        out.println("<td>"+emp.getId()+"</td>");
        out.println("</tr>");
}
      out.println("</table>");
      out.println("<br><br>");
      out.println("</body>");
      out.println("</html>");
      out.close();
    }
    catch (Exception e) {
      LogManager.logMessage("Handle this exception",
        e.getMessage() );
    }
  }

  /** Обрабатывает HTTP метод <code>GET</code>.
    * @param запрос, запрос сервлета
    * @param ответ, ответ сервлета
    */
  protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, java.io.IOException {
        processRequest(request, response);
  }

  /** Обрабатывает HTTP метод <code>POST</code>.
    * @param запрос, запрос сервлета
    * @param ответ, ответ сервлета
    */
  protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, java.io.IOException {
        processRequest(request, response);
```

```
  }

  /** Возвращает короткое описание сервлета. */
  public String getServletInfo() {
    return "Example of Servlet View. " +
              "JSP View is preferable.";
  }

  /** метод dispatcher **/
  protected void dispatch(HttpServletRequest request,
      HttpServletResponse response, String page)
  throws javax.servlet.ServletException,
      java.io.IOException      {
    RequestDispatcher dispatcher =
      getServletContext().getRequestDispatcher(page);
    dispatcher.forward(request, response);
  }
}
```

Стратегия JavaBean Helper

Хелпер реализован как JavaBean-компонент. Использование хелперов приводит к чистому разделению вида и бизнес обработки в приложении, так как бизнес логика вынесена из вида в компонент хелпера. В этом случае бизнес-логика инкапсулирована в JavaBean-компонент, который помогает извлекать содержимое, а также адаптирует и хранит модель для последующего использования ее видом.

При использовании стратегии JavaBean Helper требуется выполнить меньшую предварительную работу, чем в случае со стратегией Custom Tag Helper, так как JavaBean-компоненты проще конструируются и интегрируются в JSP-среду. Кроме того, в JavaBean-компонентах разбираются даже новички. Данная стратегия проще и с точки зрения управляемости, так как единственными полученными артефактами являются готовые JavaBean-компоненты.

Пример 7.19 Пример кода стратегии JavaBean Helper

```
<jsp:useBean id="welcomeHelper" scope="request"
  class="corepatterns.util.WelcomeHelper" />

<HTML>
<BODY bgcolor="FFFFFF">
<% if (welcomeHelper.nameExists())
{
%>
<center><H3> Welcome <b>
<jsp:getProperty name="welcomeHelper" property="name" />
</b><br><br> </H3></center>
<%
}
%>

<H4><center>Glad you are visiting our
  site!</center></H4>

</BODY>
</HTML>
```

Стратегия Custom Tag Helper

Хелпер реализован как заказной тэг (только в версии JSP 1.1+). Использование хелперов приводит к чистому разделению вида и бизнес обработки в приложении, так как бизнес логика вынесена из вида в компонент хелпера. В этом случае бизнес логика инкапсулируется в компоненте заказного тэга, который может помочь в извлечении контента и адаптировать модель для использования ее видом.

При использовании стратегии Custom Tag Helper требуется выполнить большую предварительную работу, чем в случае со стратегией JavaBean Helper, так как в среднем разработка заказного тэга сложнее разработки JavaBean-компонента. Однако дело не только в этом. Готовыми тэгами гораздо сложнее управлять и интегрировать их в среду. Использование данной стратегии требует наличия в среде большого количества

сгенерированных артефактов, включая сам тэг, дескриптор библиотеки тэгов и файлы настроек. В примере 7.20 представлена выдержка из кода, при написании которого была использована стратегия JSP View.

Пример 7.20 Пример кода для стратегии Custom Tag Helper

```
<%@ taglib uri="/web-INF/corepatternstaglibrary.tld"
  prefix="corepatterns" %>
<html>
<head><title>Employee List</title></head>
<body>

<div align="center">
<h3> List of employees in <corepatterns:department
  attribute="id"/> department - Using Custom Tag
  Helper Strategy. </h3>
<table border="1" >
    <tr>
        <th> First Name </th>
        <th> Last Name </th>
        <th> Designation </th>
        <th> Employee Id </th>
        <th> Tax Deductibles </th>
        <th> Performance Remarks </th>
        <th> Yearly Salary</th>
    </tr>
    <corepatterns:employeelist id="employeelist_key">
    <tr>
        <td><corepatterns:employee
            attribute="FirstName"/> </td>
        <td><corepatterns:employee
            attribute= "LastName"/></td>
        <td><corepatterns:employee
            attribute= "Designation"/> </td>
        <td><corepatterns:employee
            attribute= "Id"/></td>
        <td><corepatterns:employee
            attribute="NoOfDeductibles"/></td>
        <td><corepatterns:employee
            attribute="PerformanceRemarks"/></td>
        <td><corepatterns:employee
            attribute="YearlySalary"/></td>
        <td>
     </tr>
    </corepatterns:employeelist>
</table>
</div>
</body>
</html>
```

Стратегия Business Delegate as Helper

Компоненты helper часто осуществляют распределенные вызовы к бизнес ярусу. Мы предлагаем вам использовать business delegate для того, чтобы спрятать основные детали реализации данного запроса. При этом хелпер может просто вызывать бизнес-службу, не зная каких-либо деталей о ее физической реализации и распределении (см. главу «Business Delegate»).

И хелпер и business delegate могут быть реализованы, как JavaBean-компонент. Таким образом, для реализации business delegate, как специального типа хелпера, необходимо сочетать понятия компонента helper и business delegate. Важным отличием хелпера от business delegate является следующее: компонент helper пишет разработчик, отвечающий за ярус презентации, а delegate – разработчик служб в бизнес ярусе. (Примечание: delegate может также обеспечиваться как часть каркаса.) Таким образом, данная стратегия скорее связана с типом разработчиков delegate, а не с тем, как delegate реализован. Стратегия Business Delegate as Helper наилучшим образом подходит для случаев, когда роли разработчиков пересекаются.

Пример 7.21 Пример кода для стратегии Business Delegate as Helper

```
/**Сервлет передает полномочия хелперу командного объекта, как
```

```
  показано в следующей выдержке:**/
String resultPage = command.execute(request,
  response);

/**Хелпер командного объекта применяет business
  delegate, который реализован просто как другой
  JavaBean хелпер, как показано в следующей выдержке:**/

  AccountDelegate accountDelegate = new
  AccountDelegate();
```

Примечание по хелперам:

JavaBean-хелперы лучше всего подходят для помощи в извлечении контента, а также хранении и адаптации модели для вида. Кроме того, JavaBean-хелперы часто используются как командные объекты.

Хелперы заказных тэгов могут выполнять те же роли, что и JavaBean-хелперы, за исключением действия в качестве командного объекта. В отличие от JavaBean-хелперов, хелперы заказных тэгов хорошо подходят для управления потоком и итерации в виде. Хелперы заказных тэгов, используемые подобным образом, инкапсулируют логику, которая в противном случае будет напрямую встроена в JSP-страницу, как код скриптлета. Другой областью действия хелперов заказных тэгов является форматирование необработанных данных с целью их отображения. Заказной тэг производит итерацию в коллекции результатов, приводит эти результаты к виду HTML-таблицы, после чего внедряет полученную таблицу в вид JSP. При этом код Java-скриптлета не требуется.

Рассмотрим пример, в котором Web-клиент запрашивает у системы информацию об учетной записи (смотрите рисунок 7.14). На диаграмме представлены пять хелперов: четыре JavaBean-хелпера (объекты AccountCommand, Account, AccountDAO и AccountDetails) и один хелпер заказного тега (TableFormatter).

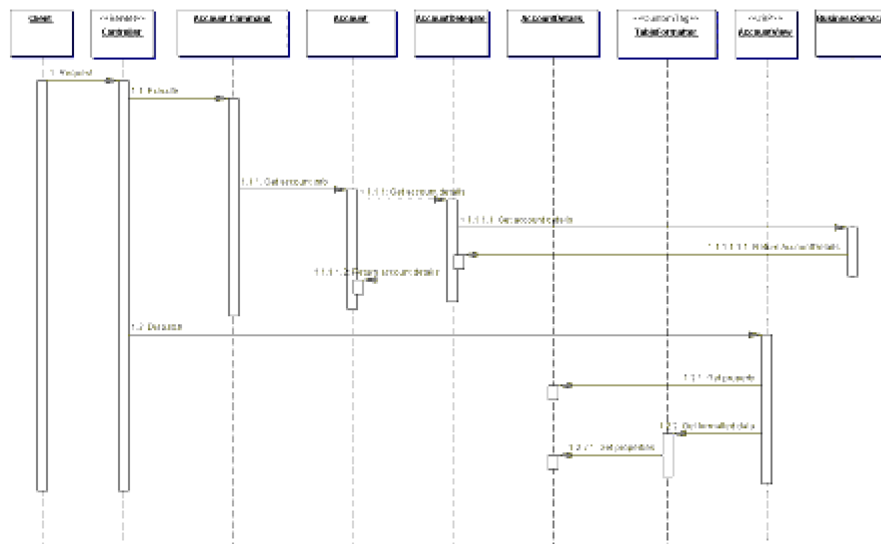

Рисунок 7.14 Использование хелперов

Запрос обрабатывает контроллер. Он создает или производит поиск подходящего командного объекта, который реализован как JavaBean-хелпер. В этом случае запрос на информацию об учетной записи обрабатывает именно командный объект. Контроллер вызывает объект Command, который запрашивает у объекта JavaBean Account информацию об учетной записи. Объект Account вызывает бизнес-службу, запрашивая именно те детали, которые возвращаются в форме Transfer Object (см. главу «Transfer Object»), реализованного как JavaBean-компонент.

Каким образом объект Account получает доступ к бизнес-службам? Давайте рассмотрим два случая, один простой, другой – посложнее. В простом случае представим, что в проекте применяется фазированный подход. При этом со временем корпоративные JavaBean-компоненты фазируются в бизнес ярус. Предположим, что доступ к базе данных осуществляется посредством JDBC-вызовов из яруса презентации. В этом случае объект Account будет использовать объект Data Access (см. главу «Data Access Object»), скрывая основные детали реализации доступа к базе данных. Объект Data Access знает, какие необходимы SQL-запросы для извлечения информации. Эти детали спрятаны от остальной части приложения, что

приводит к уменьшению связности, увеличению модульности и возможности повторного использования каждого компонента. Данный случай описан в предыдущей циклограмме.

При усложнении архитектуры и введении в бизнес ярус EJB-компонентов объект Data Access заменяется business delegate (см. главу «Business Delegate»), который обычно пишут разработчики бизнес-служб. Delegate скрывает от клиента детали реализации EJB-поиска, вызов и обработку исключительных ситуаций. Обеспечение служб кэширования также может улучшить производительность. С другой стороны, объект уменьшает связность ярусов, совершенствуя при этом возможности повторного использования и модульность для различных компонентов. Не смотря на особую реализацию данного объекта, его интерфейс в процессе этого перехода может оставаться без изменений. На рисунке 7.15 описан этот сценарий после перехода к business delegate.

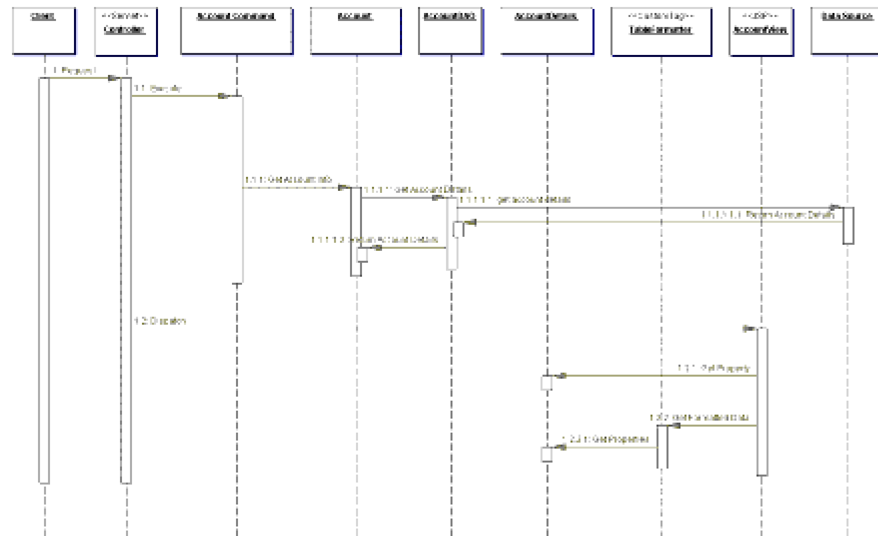

Рисунок 7.15 Организация доступа к бизнес-службам

Теперь командному объекту необходимо обработать объект AccountDetails, который он хранит до того момента, как передаст управление контроллеру. Контроллер направляет к соответствующему виду, называемому `AccountView.jsp`. Затем вид присваивает комбинацию необработанных и отформатированных данных из хелперов AccountDetails и TableFormatter соответственно. Хелпер TableFormatter реализован как заказной тэг, который проходит по необработанным данным и приводит их к виду HTML-таблицы. Как уже упоминалось, данное преобразование не требует использования в виде скриптлет-кода, который был бы обязателен для выполнения той же операции при помощи JavaBean-хелпера.

Кроме того, объект Account или хелпер AccountDetails могли бы обеспечить другие удобные способы адаптации необработанных данных. Подобные методы не вводили бы в данные HTML-разметку, поэтому они могли бы обеспечить различные комбинации данных. Например, можно было бы выводить полное имя в различных вариантах: «Фамилия Имя» или «Имя Фамилия» и т.д.

И, наконец, пользователю отображается завершенный вид.

Стратегия Transformer Helper

Хелпер реализован как eXtensible Stylesheet Language Transformer (преобразователь языка таблицы стилей). Это иногда подходит для моделей, которые существуют в виде структурной разметки (например XML) или происходят из наследственных систем, или же возникают в результате некого преобразования. Использование данной стратегии может помочь в принудительном отделении модели от вида, так как большая часть разметки вида должна быть вынесена в отдельную таблицу стилей.

На рисунке 7.16 описана потенциальная реализация данной стратегии.

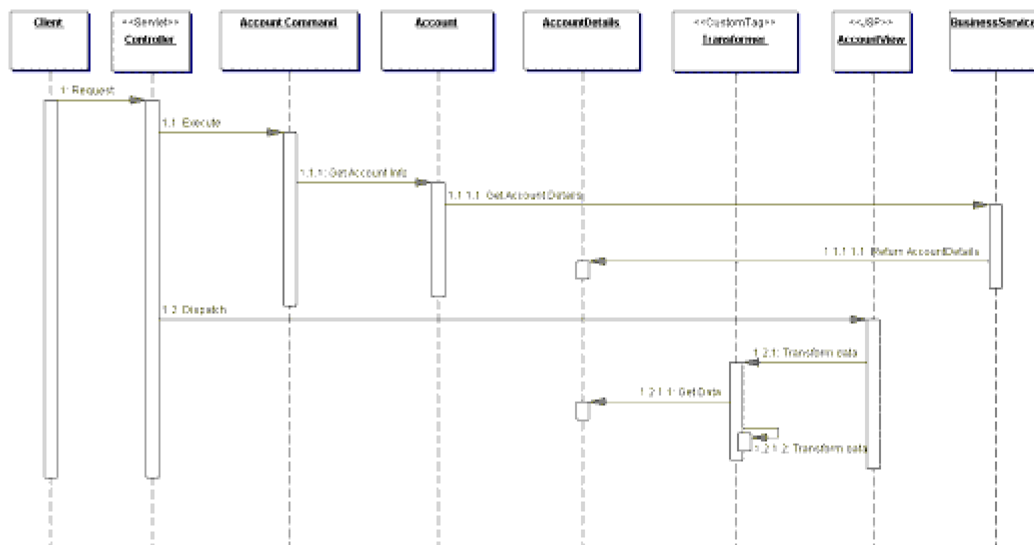

Рисунок 7.16 Циклограмма стратегии Transformer Helper

Контроллер обрабатывает запрос и вызывает объект Command, реализованный как JavaBean-хелпер. Объект Command инициирует извлечение данных Account. Объект Account вызывает бизнес-службу, которая возвращает данные в форме объекта Transfer Object (смотрите главу «Transfer Object»), реализованного как JavaBean-компонент.

После того, как будет выполнено извлечение контента, управление передастся AccountView, который для манипулирования состоянием модели использует свой преобразователь заказных тэгов. В основе преобразователя лежит таблица стилей, описывающая правила преобразования модели. Чаще всего ими являются правила соответствия разметки, позволяющей отображать информацию клиенту. Таблица стилей обычно является отдельным статичным файлом, хотя и может генерироваться динамически.

Далее представлен пример того, как может выглядеть хелпер заказного тега в AccountView:

```
<xsl:transform model="accounthelper"
 stylesheet="/transform/styles/basicaccount.xsl"/>
```

Интеграция eXtensible Stylesheets и XML с JSP развивается, так как библиотеки тэгов в этой области продолжают пополняться. На данный момент, такая стратегия является наименее предпочтительной, так как поддержка библиотек пока находится не на должном уровне, а для создания и поддержки таблиц стилей требуются дополнительные навыки.

Результаты

- **Улучшение разделения программ, возможности повторного использования и управляемости**
  Использование хелперов приводит к более чистому отделению видов от бизнес-обработки. Хелперы в форме JavaBean-компонентов (JSP 1.0+) и заказных тэгов (JSP 1.1+) обеспечивают для инкапсуляции бизнес-логики пространство вне вида. С другой стороны, коды скриптлетов загромождают JSP-страницы, что является особенно обременительным при создании больших проектов.

  Кроме того, бизнес-логика, которая вынесена из JSP-страниц в JavaBean-компоненты и заказные тэги, используется повторно. При этом количество дублирований уменьшается, а поддержка упрощается.

- **Улучшает разделение ролей**
  Разделение логики форматирования и бизнес-логики приложения уменьшает количество зависимостей. При этом разработчики, выполняющие разные роли могут работать над одними и теми же ресурсами. Например, разработчик программного обеспечения работает с кодом, внедренным в HTML-разметку, а члену команды Web-производства нужно изменять разметку страницу и компоненты дизайна, которые перемешаны с бизнес-логикой. Разработчики, выполняющие эти роли, могут не быть знакомы со спецификой работы друг друга, что повышает вероятность случайных изменений, приводя к багам в системе.

Родственные паттерны

- **Business Delegate**
  Компонентам хелперов необходим доступ к методам в интерфейсе API бизнес-службы. Важно также уменьшить связность хелперов в ярусе презентации и бизнес-служб в бизнес ярусе. Здесь рекомендуется использовать delegate, так как данные ярусы могут быть физически распределены в сети. Delegate скрывает от клиента основные детали поиска и доступа к бизнес-службам. Кроме этого, он может обеспечить промежуточное кэширование с целью уменьшения сетевого трафика.
- **Dispatcher View and Service to Worker**
  Паттерны Dispatcher View и Service to Worker можно использовать в тех случаях, когда безопасностью, технологическим потоком, извлечением контента и навигацией лучше всего управлять централизованно.
- **Front Controller**
  Данный паттерн спарен с View Helper с целью создания паттернов Dispatcher View pattern или Service to Worker.