Проверка ввода данных на Java с использованием схем XML. Часть 1

Значение проверки данных: почему Java не обеспечивает решения этой задачи самостоятельно...

Data Validation

Бретт МакЛафлин

Если Вам приходилось разрабатывать множество многофункциональных приложений на Java, особенно тех, входными данными которых являются формы Web или Swing GUI, Вам, возможно, приходилось затратить значительное время на создание кода проверки ввода, то есть, обработку всех мыслимых пользовательских ошибок. Однако, при завершении проекта этот код скорее всего оказывался настолько привязанным к конкретному приложению, что его повторное использование было невозможно. Java заботится о строгой типизации, поэтому ей недостает средств установки ограничений на ввод данных. В этой серии статей, Брет МакЛафлин предлагает решение поставленной задачи; он отвечает на вопрос, почему недостаточно одного только кода на Java, и рассматривает схему XML в качестве решения задачи установления ограничений на данные, используемые программами на Java. В заключении представлен набор универсальных и расширяемых компонент, которые позволяют освободиться от повторного сочинения логики проверки. *(2500 слов)*

Совершенствование технологии и API для Java и других языков программирования снизили сложность кодирования (лишь немногими наиболее свежими примерами тому являются JMS, EJB и XML), в результате чего усилия программистов направлены сегодня на деловую логику. С развитием стандартов деловой логики возрастают требования к определению обрабатываемых ей данных.

Например, приложение уже не просто принимает заказ на ботинки; оно должно убедиться, ботинки имеют правильный размер, имеются в наличии на складе и в заказе

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

точно указана цена. Даже в простом примере магазина ботинок правила бизнеса, которым надо следовать, могут оказаться весьма сложными. Необходимо проверять не только ввод пользователя, но и сочетание введенных данных; проверка таких сочетаний часто сопряжена с вычислениями, которые следует выполнить и проверить результат прежде, чем передавать другой компоненте приложения. При возрастающей сложности приходится тратить больше времени на написание методов проверки. Надо проверить, что значение является числом, десятичным, в долларовом эквиваленте, неотрицательным и так далее, и тому подобное.

Сервлеты и страницы JSP высылают параметры формы в виде текстовых значений (точнее, массивов строк Java), поэтому приложение должно преобразовывать их в различные типы на каждом шаге проверки пользовательского ввода. Эти преобразованные данные чаще всего направляются в компоненты сессии. Эти компоненты (beans) гарантируют безопасность преобразования типов (например, типа `int`), но не принадлежность диапазону значения. Поэтому проверка должна быть проведена вновь. В конечном итоге, можно передавать значения в логику приложения. (А сделал ли Док Мартин эти ботинки по 10 размеру?) Только после этого можно рассчитывать на надежное выполнение вычислений и выдачу правильных результатов пользователю. Если Вы уже утомлены, это хорошо! Ибо Вы начинаете чувствовать важность задачи проверки, именно этой задаче и посвящена данная серия статей.

1. Грубая и точная проверки

В первую очередь процесс проверки данных следует разбить на две части: *грубую проверку* и *точную проверку*. Рассмотрим их в отдельности.

Грубая проверка заключается в проверке данных по критериям ввода. Критериями ввода в данном случае называются основные ограничения, накладываемые на данные — тип, диапазон, список возможных значений. Эти ограничения не зависят от других данных и не требуют обращения к целевой логике. Примером грубой проверка может служить проверка на соответствие размеров ботинок положительным числам, меньше 20, целым или половинам размера.

Точная проверка представляет собой процесс выполнения целевой логики над значениями. Обычно она производится после грубой проверки и является последним

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

этапом подготовки данных, после которого результат возвращается пользователю или сообщается другим модулям для дальнейшей обработки. Примером точной проверки является проверка наличия запрошенного размера (который имеет правильный формат после грубой проверки) у заданной марки обуви. Так, размер V-образных коньков может быть только целым числом, поэтому запрос размера 10 1/2 приведет к ошибке. В данном случае требуется доступ к хранилищу данных и работа целевой программы, поэтому такой тип обработки относится к точной проверке.

Процесс точной проверки всегда зависит от приложения и не является универсальным, поэтому выходит за границы нашего повествования. Однако, грубая проверка может быть реализована во всех приложениях в соответствии с определенным набором простых правил (типизация данных, проверка диапазона и прочее). В этой серии мы рассмотрим методы грубой проверки с реализацией на базе Java/XML.

2. Данные: всегда есть, всегда озадачивают

Чтобы окончательно развеять сомнения в необходимости средств проверки, вспомним, что именно **данные** сегодня стали предметом потребления на глобальном рынке. Не приложения, не технологии, даже не люди, осуществляющие бизнес — чистые данные. Задача выбора языка программирования, сервера приложений, построения приложения решаются на пути к основной задаче поддержки данных. Таким образом, все эти решения могут быть пересмотрены и изменены. (Приходилось ли Вам переходить с SAP или dBase на Oracle? Или от NetDynamics на Lutris Enhydra?)

Фундаментальная материя всегда остаются данные. Платформы меняются, устанавливается другое программное обеспечение, но никто никогда не говорил: "Ладно, давайте сотрем все старые данные заказчика и начнем заново." Поэтому проблема ограничения данных является фундаментальной. Она всегда будет входить в состав любого приложения, на *любом* языке программирования. Работа с данными всегда проблематична из-за того, что из вводит пользователь. Люди печатают слишком быстро или слишком медленно, делают глупые ошибки или проливают кофе на клавиатуры — таковы предпосылки основной задачи сохранения точных данных и написания хорошего приложения. Имея это в виду, покажем, как сегодня можно решить эти общие проблемы.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

3. Существующие решения (и проблемы)

В силу важности проверки данных логично ожидать существование многочисленных решений данной проблемы. На деле большинство решений проверки данных выполнены топорно и отнюдь не универсально, в результате чего проверка выполняется в громоздком коде, который подходит только в специфической ситуации. Более того, код зачастую переплетается с целевой логикой и представительной логикой, что приводит к сложностям при отладке и исправлении ошибок. И, кончено, наиболее распространенным решением задачи проверки является ее игнорирование, что приводит к исключительным ситуациям для пользователя. Очевидно, что ни один из этих подходов не является хорошим решением, а понимание проблем, которых они *не* решают может помочь для определения требований к разрабатываемым ниже решениям.

3.1. Большой молот

Наиболее распространенный способ проверки данных (помимо ее игнорирования) является также наиболее тяжеловесным. Он заключается в простом кодировании проверок прямо в сервлете, классе, или EJB, в которых требуется проверка данных. В следующем примере, проверка осуществляется сразу по получении параметра от сервлета:

Проверка в коде сервлета

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ShoeServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        // принят размер ботинок
        int shoeSize;
        try {
            shoeSize = Integer.parseInt(req.getParameter("shoeSize"));
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
        } catch (NumberFormatException e) {
            throw new IOException(
"Размер ботинок должен быть числом.");
        }

        // проверить диапазон размера
        if ((shoeSize <= 0) || (shoeSize > 20)) {
            throw new IOException(
"Неправильный размер.");
        }

        // принята марка ботинок
        String brand = req.getParameter("brand");

        // проверить марку
        if (!validBrand(brand)) {
            throw new IOException(
"Неправильная марка.");
        }

        // проверить существование размера такой марки ботинок
        if (!validSizeForBrand(shoeSize, brand)) {
            throw new IOException(
"Ботинки указанного размер данной марки не выпускаются.");
        }

         // дальнейшая обработка

    }

}
```

Этот код нельзя использовать по частям, а тем более во всех случаях. Специальный параметр `shoeSize` мог быть получен, возможно, из отправленной формы HTML. Этот параметр преобразуется в численное значение (надеемся!), затем сравнивается с приемлемыми максимальной и минимальной величинами. В данном примере даже не выполняется проверка половинчатых размеров. В общем случае при получении четырех или более параметров код их проверки сервлетом составит более 100 строк. Представим теперь увеличение числа сервлетов до 10 или 15. Такой подход ведет к

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

большому объему кода, который часто плохо документирован и в котором сложно разобраться.

Помимо недостатка ясности кода проверки, часто с проверкой смешивается также и целевая логика, поддержка модульности кода становится затруднительной. В следующем примере компонент сессии не только выполняет его непосредственную задачу, но также гарантирует правильность формата данных:

Проверка в коде компонента сессии

```
import java.rmi.RemoteException;

public class ShoeBean implements javax.ejb.SessionBean {

    public Shoe getShoe(int shoeSize, String brand) {
        // проверить диапазон размера
        if ((shoeSize <= 0) || (shoeSize > 20)) {
            throw new RemoteException("Неправильный размер.");
        }

        // проверить марку
        if (!validBrand(brand)) {
            throw new RemoteException("Неправильная марка.");
        }

        // проверить существование размера такой марки ботинок
        if (!validSizeForBrand(shoeSize, brand)) {
            throw new RemoteException(
 "Ботинки указанного размер данной марки не выпускаются.");
        }

        // следует целевая логика
    }
```

Очевидная проблема здесь состоит в том, что единственным способом сообщить вызывающему компоненту о существовании проблем является генерирование `Exception`, обычно `java.rmi.RemoteException` в EJB. Это в лучшем случае делает затруднительными локализацию исключения и достойный ответ пользователю. Естественно, каждая целевая компонента, которая использует переменную `shoeSize`, должна выполнить тот же набор проверок, который стоит на пути следования данных

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

между различными блоками целевой логики.

Решение с помощью "большого молота" такого типа не способствует повторному использованию, росту ясности кода, или даже информированию пользователя. Это решение является наиболее распространенным при проверке данных, его следует использовать только как пример того, к чему *не* стоит стремиться в последующих проектах.

3.2. Малый молот

Со временем некоторые разработчики осознают проблемы подхода с "большим молотом". С ростом популярности сервлетов и обработки текстовых параметров рассматриваемая проблема стала признаваться как достойная решения. В результате появились вспомогательные классы для разбора параметров и преобразования из в специфические типы данных. Самым популярным решением стал класс `com.oreilly.servlet.ParameterParser` Джейсона Хантера, описываемый в его книге *Программирование сервлетов на Java.* (См. Ресурсы.) Класс Хантера позволяет вводить текстовые значения, отформатированные в соответствии с типом данных в них и возвращать типизированные значения. Фрагмент класса приводится ниже:

Класс com.oreilly.servlet.ParameterParser

```
package com.oreilly.servlet;

import java.io.*;
import javax.servlet.*;

public class ParameterParser {

    private ServletRequest req;

    public ParameterParser(ServletRequest req) {
        this.req = req;
    }

    public String getStringParameter(String name)
        throws ParameterNotFoundException {
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
        // Используется getParameterValues()
        // во избежание запрещенного getParameter()
        String[] values = req.getParameterValues(name);
        if (values == null)
            throw new ParameterNotFoundException(name + " not found");
        else if (values[0].length() == 0)
            throw new ParameterNotFoundException(name + " was empty");
        else
            return values[0]; // игнорировать значения со многими полями
    }

    public String getStringParameter(String name, String def) {
        try { return getStringParameter(name); }
        catch (Exception e { return def; }
    }

    public int getIntParameter(String name)
        throws ParameterNotFoundException, NumberFormatException {
        return Integer.parseInt(getStringParameter(name));
    }

    public int getIntParameter(String name, int def) {
        try { return getIntParameter(name); }
        catch (Exception e) { return def; }
    }

    // Методы для других примитивов Java
}
```

Для каждого примитивного типа данных Java предоставляются две версии вспомогательного метода. Одна возвращает преобразованное значение или генерирует исключение при неудачном преобразовании, другой возвращает преобразованное значение или значение по умолчанию, если преобразование не выполняется. Использование класса `ParameterParser` сервлетом значительно снижает количество описанных выше проблем:

Использование класса com.oreilly.servlet.ParameterParser в сервлете

```
import java.io.*;
import javax.servlet.*;
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
import javax.servlet.http.*;
import com.oreilly.servlet.ParameterParser;

public class ShoeServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        ParameterParser parser = new ParameterParser(req);

        // принят размер ботинок
        int shoeSize = parser.getIntParameter("shoeSize", 0);

        // проверить диапазон размера
        if ((shoeSize <= 0) || (shoeSize > 20)) {
            throw new IOException("Неправильный размер.");
        }

        // принята марка ботинок
        String brand = parser.getStringParameter("brand");

        // проверить марку
        if (!validBrand(brand)) {
            throw new IOException("Неправильная марка.");
        }

        // проверить существование размера такой марки ботинок
        if (!validSizeForBrand(shoeSize, brand)) {
            throw new
          IOException("указанного размер данной марки не выпускаются.");
        }

         // дальнейшая обработка

    }

}
```

Это решение лучше, но все равно оно выгладит громоздко; можно получить подходящий тип данных, но проверка диапазона все еще выполняется вручную. Оно

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

не позволяет, например, принимать набор значений (такой как "истина" и "ложь", помимо простых текстовых значений). Попытка реализовать такой тип логики с помощью класса `ParameterParser` связана с большими программными трудностями, а именно реализацией перебора как минимум четырех различных вариантов типов данных для каждого параметра.

Этот подход также требует жесткой привязки анализируемых значений параметров в код сервлета или класса Java. Максимальный размер ботинок, равный в нашем примере 20, задан в компилируемом коде, а не в простом файле, которые можно было бы без проблем изменить (как, например, файл свойств или документ XML). Изменение этого значения в коде не вызывает затруднений, но приводит к перекомпиляции исходного модуля. Такой подход ведет нас в правильном направлении (хвала Хантеру за разработку вспомогательного класса), но не дает решения задачи проверки данных.

4. Где же инструмент?

Общая проблема с проверкой правильности состоит в том, что, в настоящее время не существует ее универсального или выделенного решения. Хотя класс `ParameterParser` и является универсальным, он все же требует жестко закодированных значений и проверки диапазонов. Не существует также и решение, которое позволяло ба сеансовым компонентам просто исполнять деловую логику, принимая соответствующие значения для сравнения. Также, не существует простого способа добавить функциональные возможности к рассмотренным решениям без изменения кода — как вспомогательного класса, так и вызывающего кода.

Кроме того, эти решения несовместимы с другими приложениями и языками программирования. Данные, не удовлетворяющие специальному формату (в рассмотренных примерах классы `String` языка Java), не могут быть обработаны кодом проверки. Другими словами, эти решения попросту не действенны в современных сложных приложениях.

4.1. Чистый Java: не действует

Попытка найти решение поставленной задачи на чистом использовании языка Java

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

сама по себе является проблемой. Изменение правил проверки всегда будет приводить к перекомпиляции, если только не пытаться использовать некоторого рода некомпилируемый формат представления диапазонов, типы данных и дозволенных значений. Известны лучшие способы хранения такой информации; как было упомянуто ранее, два формата, которые могли бы помочь справиться с задачей — это файлы свойств Java и XML.

4.2. Файлы свойств

Файлы свойств Java используются в качестве метода решения задачи проверки правильности. Однако, основанный на них подход имеет несколько существенных недостатков. Во-первых, стандартные файлы свойств Java не обеспечивают вложенности свойств (`key1.key2.key3 = value`). Желаемое подчинение уровней невозможно реализовать без создания собственного кода обработки. Простой файлы свойств должен выглядеть следующим образом:

Не стандартный файл свойств

```
field.shoeSize.minSize = 0
field.shoeSize.maxSize = 20
field.brand.allowedValue = Nike
field.brand.allowedValue = Adidas
field.brand.allowedValue = Dr. Marten
field.brand.allowedValue = V-Form
field.brand.allowedValue = Mission
```

в терминах чистого Java в конце концов будет выглядеть так:

Файлы свойств Java с указанием ограничений

```
shoeSizeMin = 0
shoeSizeMax = 20
```

При этом путь решения становится менее ясным, а способа представить набор разрешенных значений размеров для марки попросту не существует — файлы свойства Java не могут иметь повторяющихся ключей с различными значениями.

Некоторые библиотеки позволяют расширенное представление данных в файлах свойств. (Смотрите для примера проект Java Apache.) Однако, использование файлов свойств для записи ограничений приводит к еще более сложной проблеме:

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

смешиванию основных функциональных возможностей. Файлы свойств обычно используются для хранения параметров запуска, информации о конфигурации, и связывающих имен в пространстве имен JNDI. Смешивание логики проверки правильности с этими данными вызывает беспорядок, и для пользователей и для программистов, которые обслуживают код.

Как искать, например, минимальный доступный размер обуви среди свойств, определяющих порт запуска сервера Web, рекомендуемый размер динамической памяти Java, и имя каталога хоста LDAP. Вместо этого следует использовать отдельный компонент, предназначенный специально для обработки информации проверки правильности.

4.3. XML приходит на помощь?

Я исследовал несколько возможных решений по обработке проверки правильности, ни одна из которых не кажется совершенной. Я предлагаю другой подход, который использует XML (и схему XML) совместно с Java. Решение будет подробно описано в следующих двух статьях серии, здесь же представлено введение.

Во-первых, отметим используя документ XML можно представить ограничения на данные. При этом ограничения, являясь значениями в документе XML, становятся изменяемыми без перетрансляции кода. Одновременно становится возможным отделение ограничений от других данных прикладной программы. Наконец, использование документов XML и схем XML позволяет использовать простой синтаксический анализатор и API (который является самостоятельным стандартом) управления данными. Для обработки данных XML не требуется дополнительных расширений или API, поэтому результирующий код является портируемым.

Схема XML определяет введенные ранее ограничения:

Ограничения проверки, представленные с помощью схемы XML

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

<schema targetNamespace="http://www.buyShoes.com"
        xmlns="http://www.w3.org/199/XMLSchema"
        xmlns:buyShoes="http://www.buyShoes.com"
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
>

  <attribute name="shoeSize">
    <simpleType baseType="integer">
      <minExclusive value="0" />
      <maxInclusive value="20" />
    </simpleType>
  </attribute>

  <attribute name="brand">
    <simpleType baseType="string">
      <enumeration value="Nike" />
      <enumeration value="Adidas" />
      <enumeration value="Dr. Marten" />
      <enumeration value="V-Form" />
      <enumeration value="Mission" />
    </simpleType>
  </attribute>

</schema>
```

Все ограничения могут быть определены как атрибуты в схеме XML, причем полностью и довольно просто. Если на базе схем XML можно выполнять грубую проверку, то можно не включать в приложение код, описанный в этой статье и сосредоточиться на целевой логике. Решение будет рассмотрено далее в этой серии.

5. Заключение

Теперь, когда мы коснулись всех возможных проблем, рассмотрели их возможные решения в соответствии с подходом чистого Java к проверке правильности, можно усомниться в возможностях языка. Однако не стоит испытывать опасений. В следующих статьях, я постараюсь реализовать решение с помощью Java. Прежде всего, я рассмотрю схемы XML более подробно, в частности богатый набор ограничений, которые она позволяет устанавливать на данные. Фактически, мы увидим, что по отношению к данным схема XML работает также, как интерфейсы Java работают по отношению к коду; она может обеспечить `интерфейс данных` для пользовательского ввода.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Для моей следующей статьи, я подготовлю набор документов XML, которые позволяют обрабатывать пользовательский ввод. Затем я использую JDOM, API Java для управления XML (и схемами XML), для программирования вспомогательных классы обработки ограничений, с которыми мы здесь уже познакомились. В результате, Вы получите запас знаний по компонентам многократного использования для проверки правильности, использующих совместную обработки на Java и XML. Я надеюсь, что до той поры Вы будете думать о проблемах, которые я обозначил, найдете приложение, на котором Вы сможете испытать код следующего месяца и познакомитесь с JDOM (см. Ресурсы), поскольку я собираюсь интенсивно ее использовать. Увидимся в следующем месяце!

6. Об авторе

Бретт МакЛафлин — стратег по Enhydra в Lutris Technologies, специализирующийся на архитектурах распределенных систем. Он является автором *Java and XML* и работает с такими технологиями как сервлеты Java, технология Enterprise JavaBeans, XML, приложения бизнес-ту-бизнес. Недавно он организовал проект JDOM совместно с Джейсоном Хантером, в котором разработал простой API для работы с XML из приложений Java. МакЛафлин активно участвует в разработке проекта Apache Cocoon и сервера EJBoss EJB, и является сооснователем проекта Apache Turbine.

7. Ресурсы

- Класс `ParameterParser`: http://www.servlets.com
- Познакомьтесь с JDOM перед чтением следующей статьи — "Easy Java/XML integration with JDOM, Part 1," Jason Hunter and Brett McLaughlin (*JavaWorld,* May 2000): http://www.javaworld.com/javaworld/jw-05-2000/jw-0518-jdom.html
- "Easy Java/XML integration with JDOM, Part 2," Jason Hunter and Brett McLaughlin (*JavaWorld,* July 2000): http://www.javaworld.com/javaworld/jw-07-2000/jw-0728-jdom2.html
- JDOM Website: http://www.jdom.org
- *Java Servlet Programming,* Jason Hunter (*O'Reilly & Associates,* November 1998): http://www.oreilly.com/catalog/jservlet
- Информация по проекту Java Apache: http://java.apache.org

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

- Дополнительные ссылки по XML — *Java и XML,* Brett McLaughlin (*O'Reilly & Associates,* June 2000): http://www.oreilly.com/catalog/javaxml

Reprinted with permission from the September 2000 edition of JavaWorld magazine. Copyright © ITworld.com, Inc., an IDG Communications company.
View the original article at: http://www.javaworld.com/javaworld/jw-09-2000/jw-0908-validation.html

Перевод на русский © Андрей Ковалев, к.т.н., доцент МИЭТ, 2000

Copyright © 1999 - 2007, IT-archiv. All rights reserved.