

**Power EnJoy**  
*Code Inspection*  
Software Engineering 2  
A.A. 2016/2017  
Version 1.0

Emanuele Ghelfi (mat. 875550)  
Emiliano Gagliardi (mat. 878539)

January 29, 2017

## Contents

<b>1</b>	<b>Class assigned to the group</b>	<b>3</b>
<b>2</b>	<b>Functional role of the assigned class</b>	<b>3</b>
<b>3</b>	<b>List of issues found by applying the checklist</b>	<b>5</b>
3.0.1	Naming Conventions . . . . .	5
3.0.2	Indention . . . . .	5
3.0.3	Braces . . . . .	5
3.0.4	File Organization . . . . .	6
3.0.5	Wrapping Lines . . . . .	8
3.0.6	Comments . . . . .	9
3.0.7	Java Source Files . . . . .	9
3.0.8	Package and Import Statements . . . . .	9
3.0.9	Class and Interface Declarations . . . . .	9
3.0.10	Initialization and Declarations . . . . .	10
3.0.11	Method Calls . . . . .	11
3.0.12	Arrays . . . . .	11
3.0.13	Object Comparison . . . . .	11
3.0.14	Output Format . . . . .	11
3.0.15	Computation, Comparisons and Assignments . . . . .	12
3.0.16	Exceptions . . . . .	12
3.0.17	Flow of Control . . . . .	13
3.0.18	Files . . . . .	13
<b>4</b>	<b>Other problem</b>	<b>14</b>
4.1	Return statements . . . . .	14
4.2	Error Handling . . . . .	14
4.3	Performance Issues . . . . .	20
4.4	Method Metrics . . . . .	21
4.5	Assignments Issues . . . . .	21

## 1 Class assigned to the group

The class assigned to the group is the class `SOAPEventHandler.java`.  
The namespace is: `apache-ofbiz-16.11.01/framework/webapp/src/main/-java/org/apache/ofbiz/webapp/event/SOAPEventHandler.java`

## 2 Functional role of the assigned class

`SOAPEventHandler` class is a class responsible for managing soap events. It's the implementation of the Event Handler interface.  
Simple Object Access Protocol (SOAP) is a protocol for exchanging structured information in a decentralized, distributed environment. It is XML-based and consists of three parts:

- An envelope that defines a framework for describing what is in a message and how to process the message
- A set of encoding rules for expressing instances of application defined data types
- A convention for representing remote procedure calls and responses

`SOAPEventHandler` takes a series of service to call, a delegator and a dispatcher.

Parses the request and calls the dispatcher with the services to call.

The functional role of the class has been understood by looking at the JavaDoc and at the code and trying to understand the class logic.

### Init Method

Initializes the handler. Since handlers use the singleton pattern this method should only be called the first time the handler is used.

`ServletContext` parameter may be needed by the handler in order to lookup properties or XML definition files for rendering pages or handler options.

In this class the context parameter is totally ignored but it's imposed by the `init` method of `EventHandler` interface.

### Invoke Method

This method invokes the web event.

Parameter `event` contains information about what to execute.

Parameter `requestMap` contains information about the request-map the event was called from.

Parameter `request` is the servlet request object.

Parameter response is the servlet response object.  
This method returns the result code of the event (i.e. if the event has been done in a proper way or if some error happened).

## 3 List of issues found by applying the checklist

### 3.0.1 Naming Conventions

1. All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.
2. If one-character variables are used, they are used only for temporary “throwaway” variables, such as those used in for loops.
3. Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: class Raster; class ImageSprite;
4. Interface names should be capitalized like classes.
5. Method names should be verbs, with the first letter of each addition word capitalized. Examples: getBackground(); computeTemperature().
6. Class variables, also called attributes, are mixed case, but might begin with an underscore ( ‘ ’ ) followed by a lowercase first letter. All the remaining words in the variable name have their first letter capitalized. Examples: windowHeight, timeSeriesData.
7. Constants are declared using all uppercase with words separated by an underscore. Examples: MIN\_WIDTH; MAX\_HEIGHT.

No issues found in naming conventions.

### 3.0.2 Indention

1. Three or four spaces are used for indentation and done so consistently.
2. No tabs are used to indent.

No issues found in indentions.

### 3.0.3 Braces

1. Consistent bracing style is used, either the preferred “Allman” style (first brace goes underneath the opening block) or the “Kernighan and Ritchie” style (first brace is on the same line of the instruction that opens the new block).
2. All if, while, do-while, try-catch, and for statements that have only one statement to execute are surrounded by curly braces. Example: Avoid this: if ( condition ) doThis(); Instead do this: if ( condition ) { doThis(); }

No issues found

### 3.0.4 File Organization

1. Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods).

```
/* *****  
 * Licensed to the Apache Software Foundation (ASF) under one  
 * or more contributor license agreements. See the NOTICE file  
 * distributed with this work for additional information  
 * regarding copyright ownership. The ASF licenses this file  
 * to you under the Apache License, Version 2.0 (the  
 * "License"); you may not use this file except in compliance  
 * with the License. You may obtain a copy of the License at  
 *  
 * http://www.apache.org/licenses/LICENSE-2.0  
 *  
 * Unless required by applicable law or agreed to in writing,  
 * software distributed under the License is distributed on an  
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY  
 * KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations  
 * under the License.  
 * *****  
package org.apache.ofbiz.webapp.event;
```

There is no blank line between beginning comments and package statement.

2. Where practical, line length does not exceed 80 characters.  
According to the general principles of “breaking after a comma” and “breaking before an operator” when a statement doesn’t fit in a single line, the following lines have been found for this point (considering only modification that doesn’t compromise the readability):

- `ModelService model = dispatcher.getDispatchContext().  
getModelService(serviceName);`

At line 182, with 99 characters. The line should break before the last dot operator.

- `Debug.logError("Trying to call Service[" + serviceName + "]  
that is not exported.", module);`

At line 192, with 114 characters. The breaking of the line on one of the two “+” operators compromises the readability, at least the line can be broken after the comma.

- `sendError(response, "Problem processing the service, check  
your parameters.", serviceName);`

At line 203, with 112 characters. The line could be broken after each comma, separating the parameters.

- `XMLStreamReader reader = XMLInputFactory.newInstance().  
createXMLStreamReader(new StringReader(xmlResults));`

At line 242, with 120 characters. The line should be broken before the second dot operator.

- `OMElement resService = factory.createOMElement(new QName(serviceName + "Response"));`

At line 250, with 97 characters. The line should be broken before the first dot operator

- `OMAttribute defaultNS = factory.createOMAttribute("xmlns", null, ModelService.TNS);`

At line 259, with 96 character. The line should be broken separating the parameters of `factory.createOMAttribute`.

- `OMAttribute defaultNS = factory.createOMAttribute("xmlns", null, ModelService.TNS);`

At line 308, with 96 characters. The same as the previous considered line.

3. When line length must exceed 80 characters, it does NOT exceed 120 characters.

- ```
/**
 * @see org.apache.ofbiz.webapp.event.EventHandler#invoke(
 *      ConfigXMLReader.Event, ConfigXMLReader.RequestMap, javax.
 *      servlet.http.HttpServletRequest, javax.servlet.http.
 *      HttpServletResponse)
 */
```

At line 78, the characters are 192. The comment is on a single line.

- `public String invoke(Event event, RequestMap requestMap, HttpServletRequest request, HttpServletResponse response) throws EventHandlerException {`

At line 80, the characters are 150. The line should be broken on the second parameter comma and on the throws clause.

- `sb.append("<li><a_href=\"\"\".append(locationUri).append(\"/\").append(model.name).append(\"?wsdl\">\");`

At line 131, the characters are 127. The line should be broken before the third dot operator.

- `Map<String, Object> parameters = UtilGenerics.cast(SoapSerializer.deserialize(serviceElement.toString(), delegator));`

At line 179, the characters are 130. It is better to reduce the call nesting, defining temporal variables only to improve the readability.

- `throw new EventHandlerException("One service call expected, but received: " + numServiceCallRequests.toString());`

At line 232, the characters are 126. The line should be broken before the “+” operator.

- `private void createAndSendSOAPResponse(Map<String, Object> serviceResults, String serviceName, HttpServletResponse response) throws EventHandlerException {`

At line 236, the characters are 160. The line should be broken on the parameters comma and on the throw clause.

- `StAXOMBuilder resultsBuilder = (StAXOMBuilder) OMXMLBuilderFactory.createStAXOMBuilder(OMAbstractFactory.getOMFactory(), reader);`

At line 243, the characters are 142. The line should be broken before the dot operator.

- `private void sendError(HttpServletResponse res, String errorMessage, String serviceName) throws EventHandlerException {`

At line 278, the characters are 124. The line should be broken on the parameters comma and on the throw clause.

- `private void sendError(HttpServletResponse res, List<String> errorMessages, String serviceName) throws EventHandlerException {`

At line 283, the characters are 131. The line should be broken on the parameters comma and on the throw clause.

- `XMLStreamReader xmlReader = XMLInputFactory.newInstance().createXMLStreamReader(new StringReader(xmlResults));`

At line 291, the characters are 123. The line should be broken before the second dot operator.

- `StAXOMBuilder resultsBuilder = (StAXOMBuilder) OMXMLBuilderFactory.createStAXOMBuilder(OMAbstractFactory.getOMFactory(), xmlReader);`

At line 292, the characters are 145. The line should be broken before the second dot operator.

- `OMELEMENT errMsg = factory.createOMELEMENT(new QName((serviceName != null ? serviceName : "") + "Response"));`

At line 299, the characters are 122. A temporal variable should be used to reduce the call nesting and to improve the readability.

### 3.0.5 Wrapping Lines

1. Line break occurs after a comma or an operator.
2. Higher-level breaks are used.
3. A new statement is aligned with the beginning of the expression at the same level as the previous line.

In the class there are no line breaks.



### 3.0.6 Comments

1. Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.  
//TODO
2. Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.

### 3.0.7 Java Source Files

1. Each Java source file contains a single public class or interface.
2. The public class is the first class or interface in the file.
3. Check that the external program interfaces are implemented consistently with what is described in the javadoc.
4. Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).

### 3.0.8 Package and Import Statements

1. If any package statements are needed, they should be the first non-comment statements. Import statements follow.

### 3.0.9 Class and Interface Declarations

1. Order into the class declaration and body  
No issues found.
2. Methods are grouped by functionality rather than by scope or accessibility.  
In the inspected class there is only one useful public method that is invoke. The init method is only imposed by the interface implementation. The functionality grouping is respected because all the other methods are private methods that decompose some invoke method operations (they are all called by the invoke method).
3. Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.  
The code is free of duplicates.  
The invoke method starts at line 80 and ends at line 221, too much lines for a method. A part consist of error handling, that can be separated in a private method. Another big code block has to be executed only if a condition is satisfied (`wsdlReq != null`), and has the objective to obtain some information about the request (service, dispatcher context) so it can be separated from the method and decomposed.

### 3.0.10 Initialization and Declarations

1. Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected).
2. Check that variables are declared in the proper scope.
3. Check that constructors are called when a new object is desired.
4. Check that all object references are initialized before use.
5. Variables are initialized where they are declared, unless dependent upon a computation.
6. Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces '{' and '}'). The exception is a variable can be declared in a for loop.

```
try {
    SOAPBody reqBody = reqEnv.getBody();
    validateSOAPBody(reqBody);
    OMElement serviceElement = reqBody.getFirstElement();
    serviceName = serviceElement.getLocalName();
    Map<String, Object> parameters = UtilGenerics.cast(SoapSerializer.de
try {
    // verify the service is exported for remote execution and invoke
    ModelService model = dispatcher.getDispatchContext().getModelSer

    if (model == null) {
        sendError(response, "Problem processing the service", service
        Debug.logError("Could not find Service [" + serviceName + "]
        return null;
    }

    if (!model.export) {
        sendError(response, "Problem processing the service", service
        Debug.logError("Trying to call Service [" + serviceName + "]
        return null;
    }

    Map<String, Object> serviceResults = dispatcher.runSync(serviceName
    Debug.logVerbose("[EventHandler] : Service invoked", module);

    createAndSendSOAPResponse(serviceResults, serviceName, response)
```

At line 175. In this fragment the variable **serviceResults** is not declared at the beginning of the block, but instead is declared and initialized in the middle of the block.

The variable should be declared at the beginning and since it depends upon a computation should be initialized only when the computation is available.

#### **3.0.11 Method Calls**

1. Check that parameters are presented in the correct order.
2. Check that the correct method is being called, or should it be a different method with a similar name.
3. Check that method returned values are used properly.

No issues found in method calls.

#### **3.0.12 Arrays**

1. Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index).
2. Check that all array (or other collection) indexes have been prevented from going out-of-bounds.
3. Check that constructors are called when a new array item is desired.

No issues found in Arrays since arrays are used rarely in this class.

#### **3.0.13 Object Comparison**

1. Check that all objects (including Strings) are compared with equals and not with ==.

No issues found in Object Comparison.

#### **3.0.14 Output Format**

1. Check that displayed output is free of spelling and grammatical errors.
2. Check that error messages are comprehensive and provide guidance as to how to correct the problem.
3. Check that the output is formatted correctly in terms of line stepping and spacing.

No issues found in Output Format.

### 3.0.15 Computation, Comparisons and Assignments

1. Check that the implementation avoids “brutish programming”: (see <http://users.csc.calpoly.edu/~j-dalbey/SWE/CodeSmells/bonehead.html>).
2. Check order of computation/evaluation, operator precedence and parenthesesizing.
3. Check the liberal use of parenthesis is used to avoid operator precedence problems.
4. Check that all denominators of a division are prevented from being zero.
5. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding.
6. Check that the comparison and Boolean operators are correct.
7. Check throw-catch expressions, and check that the error condition is actually legitimate.
8. Check that the code is free of any implicit type conversions.

No issues found in Computation, Comparisons and Assignments since in this class there are no computations.

### 3.0.16 Exceptions

1. Check that the relevant exceptions are caught.
2. Check that the appropriate action are taken for each catch block.

```
// log the request message
    if (Debug.verboseOn()) {
        try {
            Debug.logInfo("Request Message:\n" + reqEnv + "\n", module);
        } catch (Throwable t) {
        }
    }
```

In this case there are no reason for inserting a try-catch block since the Throwable is never thrown and there no actions in the block.

This error is present also in the following examples:

```
// log the response message
    if (Debug.verboseOn()) {
        try {
            Debug.logInfo("Response Message:\n" + resEnv + "\n", module);
        } catch (Throwable t) {
        }
    }
```

```
// log the response message
    if (Debug.verboseOn()) {
        try {
            Debug.logInfo("Response Message:\n" + resEnv + "\n", module)
        } catch (Throwable t) {
        }
    }
}
```

### 3.0.17 Flow of Control

1. In a switch statement, check that all cases are addressed by break or return.
2. Check that all switch statements have a default branch.
3. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions.

No issues found in Flow of Control since loops, switch are rarely used in this class.

### 3.0.18 Files

1. Check that all files are properly declared and opened.
2. Check that all files are closed properly, even in the case of an error.
3. Check that EOF conditions are detected and handled correctly.
4. Check that all file exceptions are caught and dealt with accordingly.

No issues found in Files since files are used rarely and in the proper way.

## 4 Other problem

### 4.1 Return statements

A possible problem may be that the invoke method should return a result code that specifies the result of the event. The invoke method of this class always returns a null string. This could be a problem if the caller method performs some operations on that string. This object should reflect the event result and such a string doesn't say nothing.

While occasionally useful, this construct may make the code more prone to failing with a `NullPointerException`, and often indicates that the developer doesn't really understand the class' intended semantics.

```
if (wsdl != null) {
    try {
        OutputStream os = response.
            getOutputStream();
        response.setContentType("text/xml");
        UtilXml.writeXmlDocument(os, wsdl);
        response.flushBuffer();
    } catch (IOException e) {
        throw new EventHandlerException(e);
    }
    return null;
}
```

This is only an example, at line 106.

### 4.2 Error Handling

#### Nested try statements

Nested try statements may result in confusing code, and should probably have their catch and finally sections merged.

```
try {
    SOAPBody reqBody = reqEnv.getBody();
    validateSOAPBody(reqBody);
    OMElement serviceElement = reqBody.
        getFirstElement();
    serviceName = serviceElement.getLocalName();
    Map<String, Object> parameters = UtilGenerics
        .cast(SoapSerializer.deserialize(
            serviceElement.toString(), delegator));
    try {
```

```

// verify the service is exported for
remote execution and invoke it
ModelService model = dispatcher.
    getDispatchContext().getModelService(
        serviceName);

if (model == null) {
    sendError(response, "Problem
        processing the service",
        serviceName);
    Debug.logError("Could not find
        Service [" + serviceName + "].",
        module);
    return null;
}

if (!model.export) {
    sendError(response, "Problem
        processing the service",
        serviceName);
    Debug.logError("Trying to call
        Service [" + serviceName + "] that
        is not exported.", module);
    return null;
}

Map<String, Object> serviceResults =
    dispatcher.runSync(serviceName,
        parameters);
Debug.logVerbose("[EventHandler] :
    Service invoked", module);

createAndSendSOAPResponse(serviceResults,
    serviceName, response);

} catch (GenericServiceException e) {
    if (UtilProperties.getPropertyAsBoolean("
        service", "secureSoapAnswer", true)) {
        sendError(response, "Problem
            processing the service, check your
            parameters.", serviceName);
    } else {
        if (e.getMessageList() == null) {
            sendError(response, e.getMessage
                (), serviceName);
        } else {

```

```

        sendError(response, e.
            getMessageList(), serviceName)
        ;
    }
    Debug.logError(e, module);
    return null;
}
}
} catch (Exception e) {
    sendError(response, e.getMessage(),
        serviceName);
    Debug.logError(e, module);
    return null;
}
}

```

At line 175.

```

try {
    // setup the response
    Debug.logVerbose("[EventHandler] : Setting up
        response message", module);
    String xmlResults = SoapSerializer.serialize(
        serviceResults);
    //Debug.logInfo("xmlResults
        =====" + xmlResults, module);
    XMLStreamReader reader = XMLInputFactory.
        newInstance().createXMLStreamReader(new
        StringReader(xmlResults));
    StAXOMBuilder resultsBuilder = (StAXOMBuilder
        ) OMXMLBuilderFactory.createStAXOMBuilder(
        OMAbstractFactory.getOMFactory(), reader);
    OMElement resultSer = resultsBuilder.
        getDocumentElement();

    // create the response soap
    SOAPFactory factory = OMAbstractFactory.
        getSOAP11Factory();
    SOAPEnvelope resEnv = factory.
        createSOAPEnvelope();
    SOAPBody resBody = factory.createSOAPBody();
    OMElement resService = factory.
        createOMEElement(new QName(serviceName + "
        Response"));
    resService.addChild(resultSer.getFirstElement
        ());
    resBody.addChild(resService);
    resEnv.addChild(resBody);
}

```



```

// The declareDefaultNamespace method doesn't
// work see (https://issues.apache.org/jira/browse/AXIS2-3156)
// so the following doesn't work:
// resService.declareDefaultNamespace(
//     ModelService.TNS);
// instead, create the xmlns attribute
// directly:
OMAttribute defaultNS = factory.
    createOMAttribute("xmlns", null,
        ModelService.TNS);
resService.addAttribute(defaultNS);

// log the response message
if (Debug.verboseOn()) {
    try {
        Debug.logInfo("Response Message:\n" +
            resEnv + "\n", module);
    } catch (Throwable t) {
    }
}

resEnv.serialize(response.getOutputStream());
response.getOutputStream().flush();
} catch (Exception e) {
    Debug.logError(e, module);
    throw new EventHandlerException(e.getMessage
        (), e);
}

```

At line 239.

### Throw inside catch block that ignores the caught exception

It is considered good practice when throwing an exception in response to an exception to wrap the initial exception, so that valuable context information such as stack frames and line numbers are not lost.

```

} catch (Exception e) {
    sendError(response, "Unable to obtain
        WSDL", null);
    throw new EventHandlerException("
        Unable to obtain WSDL");
}

```

At line 143.

### Overly broad catch block

Catch blocks which have parameters which are more generic than the exceptions thrown by the corresponding try block.

```
try {
    Writer writer = response.getWriter();
    StringBuilder sb = new StringBuilder
        ();
    sb.append("<html><head><title>OFBiz
        SOAP/1.1 Services</title></head>")
        ;
    sb.append("<body>No such service.").
        append("<p>Services:<ul>");

    for (String scvName: dctx.
        getAllServiceNames()) {
        ModelService model = dctx.
            getModelService(scvName);
        if (model.export) {
            sb.append("<li><a href=\"").
                append(locationUri).append
                    ("/").append(model.name).
                    append("?wsdl\">");
            sb.append(model.name).append(
                "</a></li>");
        }
    }
    sb.append("</ul></p></body></html>");

    writer.write(sb.toString());
    writer.flush();
    return null;
} catch (Exception e) {
    sendError(response, "Unable to obtain
        WSDL", null);
    throw new EventHandlerException("
        Unable to obtain WSDL");
}
```

At line 123. Catch of Exception is masking IOException and GenericServiceException.

```
1  try {
2      InputStream inputStream = (InputStream)
        request.getInputStream();
3      SOAPModelBuilder builder = (SOAPModelBuilder)
        OMXMLBuilderFactory.
```

```

        createSOAPModelBuilder(inputStream, "UTF-8
        ");
4      reqEnv = (SOAPEnvelope) builder.
        getDocumentElement();
5
6      // log the request message
7      if (Debug.verboseOn()) {
8          try {
9              Debug.logInfo("Request Message:\n" +
                reqEnv + "\n", module);
10             } catch (Throwable t) {
11             }
12         }
13     } catch (Exception e) {
14         sendError(response, "Problem processing the
            service", null);
15         throw new EventHandlerException("Cannot get
            the envelope", e);
16     }

```

At line 155. Catch of Exception is masking IOException.

```

    } catch (Exception e) {
        sendError(response, e.getMessage(),
            serviceName);
        Debug.logError(e, module);
        return null;
    }

```

At line 215. Catch of Exception is masking exceptions OMException, EventHandlerException, SerializeException, SAXException, ParserConfigurationException and IOException.

```

    } catch (Exception e) {
        Debug.logError(e, module);
        throw new EventHandlerException(e.getMessage()
            (), e);
    }

```

At line 274. Catch of Exception is too broad, masking exceptions SerializeException, FileNotFoundException, IOException, XMLStreamException' and SOAPProcessingException.

```

    } catch (Exception e) {
        throw new EventHandlerException(e.getMessage(), e);
    }

```

At line 323. Catch of Exception is too broad, masking exceptions SerializeException, FileNotFoundException, IOException, XMLStreamException and SOAPProcessingException.

## 4.3 Performance Issues

### String literals of length one

String literals of length one should not be used in concatenation. These literals may be replaced by equivalent character literals, gaining some performance enhancement.

```
sb.append("<li><a href=\"") .append(locationUri) .append("
/") .append(model.name) .append("?wsdl\">");
```

At line 132.

```
Debug.logInfo("Request Message:\n" + reqEnv + "\n",
    module);
```

At line 163.

```
uri.append(":");
```

At line 334.

```
reqInfo = "/" + reqInfo;
```

At line 342.

### String Buffer or String Builder without initial capacity

It's a bad practice to instantiate a new `StringBuffer` or `StringBuilder` object without specifying its initial capacity. If no initial capacity is specified, a default capacity is used, which will rarely be optimal. Failing to specify initial capacities for `StringBuffers` may result in performance issues, if space needs to be reallocated and memory copied when capacity is exceeded.

```
Writer writer = response.getWriter();
    StringBuilder sb = new StringBuilder
        ();
    sb.append("<html><head><title>OFBiz
        SOAP/1.1 Services</title></head>")
        ;
    sb.append("<body>No such service.")
        .append("<p>Services:<ul>");
```

At line 124.

```
StringBuilder uri = new StringBuilder();
    uri.append(request.getScheme());
    uri.append("://");
    uri.append(request.getServerName());
```

At line 329.

## 4.4 Method Metrics

### Overly Complex Method

Cyclomatic complexity is basically a measurement of the number of branching points in a method. Methods with too high a cyclomatic complexity may be confusing and difficult to test.

In this class method `invoke` has a too high cyclomatic complexity. There are multiple `if` statements and 21 branching point.

### Method with multiple return points

Methods with too many return points may be confusing, and hard to refactor. A return point is either a return statement or the falling through the bottom of a void method or constructor.

In this class the method `invoke` has 7 return points.

## 4.5 Assignments Issues

### Assignment to 'null'

The assignment of a variable to `null`, outside of declarations may be an issue. While occasionally useful for triggering garbage collection, this construct may make the code more prone to `NullPointerExceptions`, and often indicates that the developer doesn't really understand the class's intended semantics.

At line 95, in `Invoke` method there is the following assignment to `null`:

```
if (serviceName != null) {
    Document wsdl = null;
    try {
        wsdl = dctx.getWSDL(serviceName,
            locationUri);
    } catch (GenericServiceException e) {
        serviceName = null;
    } catch (WSDLException e) {
        sendError(response, "Unable to obtain
            WSDL", serviceName);
        throw new EventHandlerException("
            Unable to obtain WSDL", e);
    }
}
```

The variable `serviceName` is assigned to `null` in the `catch` clause. This could lead to `NullPointerException`.