Strongly Complex Typed, Dialogue-Oriented Server Pages

Technical Report B-02-05

Dirk Draheim and Gerald Weber
Institute of Computer Science
Free University Berlin
email: {draheim,weber}@inf.fu-berlin.de

March 2002

Abstract

We present NSP, a new, statically typed server pages technology. NSP supports user defined types within input forms, provides an exception mechanism and allows automatic reverse engineering. NSP is the result of designing server pages from scratch. It addresses the needs of server pages developers who want to build cleaner, better reusable, more stable systems and is oriented towards several accepted best software engineering practices.

Contents

1 Introduction

Next Server Pages (NSP) is a programming model for system dialogues based on dynamically generated pages, like today's web interfaces. In the following we summarize NSP's original contributions to server pages development.

- Strong type system
  - Static object system. A web application is accessed through a set of single pages, which together comprise the system interface. In NSP, the system interface is viewed as a static object system. NSP pages offer remote methods that send HTML output back to the caller.
  - Static type checker. NSP defines a strong type system that incorporates NSP methods that are called across the net. A new widget set is supporting technology for it. At deployment time the system interface is statically checked as a whole against sophisticated rules for writing method calls and writing forms. The NSP rules guarantee that no type error occurs concerning the interplay between the pages of the system.
  - Complex types. Arrays and user defined types may be incorporated in writing forms by using special mechanisms.
  - Native typed form parameters in the scripting language. The scripts within the server page receive the values passed via a form in the native data types of the respective scripting language, and not only as string values.
  - No unresolved links. Since NSP uses its own syntax for links and forms to other pages within the same system interface, it is possible to check, whether these links and forms point to existing targets within the system dialogue.
- Exceptions. An exception concept for forms allows implementing constraint checking in compliance with accepted good practice.
- Reverse engineering. From a system written in NSP the developer can generate a specification in the high level specification language Gently.

For the time being the NSP approach is oriented mainly towards building enterprise information systems [12] rather than building information architectures [18]. NSP does not compete with content management systems or content management frameworks.

In section 2 we motivate NSP by showing up the deficiencies of current server scripts and our theoretical model for the development of NSP. In section 3 we give a hands on introduction to NSP, afterwards an overview of the whole NSP language in section 4. We outline an operational semantics in section 5. Section 6 explains the use of NSP with respect to documentation and reverse engineering, and section 7 derives from this discussion new directions for further work. Finally in section 8 we describe the relation of NSP to other approaches, which will turn out to be orthogonal to NSP.

NSP is a paradigm for dynamically generated server pages that is independent from the chosen scripting language. However, for every language supported by NSP a nontrivial language mapping has to be provided. The current main development and therefore the presentation of NSP in this paper is oriented towards Java. The Java language mapping exemplifies the amalgamation of NSP with the scripting language.

The source code of an NSP component is an XML document. The language NSP is the result of changing and extending XHTML with respect to the new NSP features. A system developed with NSP runs with standard browsers as ultra-thin clients - no plug-ins are necessary.

2 Motivation

Today many web applications rely heavily on dynamically generated pages. They are often built from HTML pages with embedded server side scripts. Well known technologies in this domain are PHP, Active Server Pages (ASP), or Java Server Pages (JSP).

The interaction between browser and server is synchronous communication: the user of an HTTP dialogue causes the browser to send page requests to the server, which can be seen as remote method calls on the server. The HTML form standard offers a text based remote parameter passing mechanism for dynamic websites. The submitted parameters are called HTML form parameters in the following. The described parameter passing mechanism is untyped with respect to the parameter set, the single parameters, and finally with respect to the return value, namely the generated page. The same mechanism can be used in HTTP links as well: they can contain URLs following the syntax for HTML form parameter transmission. These links have to be created on the server side. NSP aims to bring the benefits of static type checking to the programming of such dialogues. For simplicity, wherever we want to reason about the system interface, we suppose that the system interface is based completely on pages generated by server scripts. Furthermore we will discuss in the following NSP in the concrete combination with Java, which also allows us to discuss the improvements to JSP achieved by NSP.

We want to recall that the JSP technology is a server side scripting technology in that it is a proposed standard technique to implement servlets [3][17][7]. JSP let the programmer feel like programming a preprocessor for servlets, because many details of the Java Servlet API are visible.

The previously explained lack of typing in the HTTP form mechanism is not mitigated in any form by the server side scripting techniques like JSP. JSP offers the HTML form parameters to script code through an object oriented mechanism via the HTTPRequest parameter. The interesting parameters, namely the HTML form parameters provided by the forms or links calling the page, are neither typed, nor statically type checked. The static type checking is performed only on the technical level, e.g. concerning the HTTPRequest and HTTPResponse parameters. In the same way, JSP does not offer any support for creating correct calls to other JSP pages in forms and especially not in links. For links, the programmer must code the parameter passing to another server script within the result page by manually concatenating script output in the special HTML form syntax.

The server script concept is open for overloading, i.e. polymorphic use of one server page: one server page can respond to different sets of HTML form parameters, especially it can use default values if parameters are not provided. Again, the server script mechanisms like JSP do not perform any checking on the parameter set. Moreover, within the JSP technology there is no guidance for using this mechanism in accordance with common good practices.

The NSP approach does not restrict the potentials of JSP in any way, but has overcome the aforementioned problems, mainly because it is based on a clear model of page based dialogues. We now describe the model of such page based dialogues, which serves as a basis for identifying and clarifying the NSP notions.

In the NSP model, a page based system interface built with scripted HTML pages is seen as a static object system, each page is seen as a single object. If a single scripted page is open for polymorphic use, and can be used with different parameter signatures, the object shall have a method for each such signature. For example, if a JSP page that puts an item into the shopping cart can be called either with a product ID only, or with a product ID plus a quantity, the page shall have two methods, the first having one parameter, the second having two parameters. Each method within that object system is called a dialogue method, in order to distinguish them from business methods that reside on a different layer in a broader architectural discussion.

Dialogue methods produce HTML pages as return values. Within one NSP component the pages generated by the different dialogue method may even differ in their static part. Therefore within the terminology of NSP the code unit replacing a JSP page will be called a component rather than a page, while the produced HTML document will be called the result page. A further justification of this usage of the term component is given in the section on related work.

Each result page offers to the user the choice between different links or forms as the next step in the dialogue. These links and forms are calls to dialogue methods. Forms can be seen as editable method calls offered to the user. For each dialogue method, NSP will require the specification of all links and forms contained in the result page. One of the main contributions of the NSP static type checker is to check the link and form syntax against the method declaration of the called dialogue method.

This introduces one of the most important observations in the NSP concept: from the viewpoint of the type system [4] of NSP the generated result pages are the actual code, which has to be considered. Although the generated code is naturally not available at deployment time, NSP defines rules for writing the server script parts, the so called NSP coding rules, which are

- non prohibitive: all reasonable applications of scripting are still allowed,
- statically checkable: accordance with NSP coding rules can be checked at compile time,
- sufficient: if NSP coding rules are followed, the question of whether the generated HTML code will be type-safe or not can be decided.

The viewpoint that forms are editable method calls is a starting point to understand the motivation of the NSP exception mechanism: this mechanism is designed to deal with exceptions caused by user input that violates constraints, which are required by the system.

The NSP technique is independent from any particular server side scripting technology, it could even be abstracted from HTTP/HTML and could lead to a general standard for specifying page based dialogues.

3 A First Example

The following example points up the most important NSP notion, i.e. the division of an NSP component in methods that are called in a type-safe manner across the net. This minimal example allows the registration of a webshop customer, i.e. the storage of its data to a database.

A standard HTML page may reference the NSP component "Shop" by a hyperlink. If this link is followed by the user, the NSP component's main method is invoked. As a result an HTML page is sent that consists of the head of the NSP component and the method body of the main method as HTML body. Then the user may follow a link to a registration page which is generated after a call to the shop's registration method. Note that the hyperlink to the registration method is not realized by a standard <a href> tag, but by a new NSP tag structure for links. The reason for this will become obvious later.

Within the registration page a form is offered to the user. This form sends the user input to the dialogue method newCustomer. An NSP form targets a component's method that is specified via its callee attribute. Note that in the same terminology the action attribute of an HTML form specifies a complete component as the form's target. The called method has a well-defined signature. Somewhat similar to the parameter mechanism in ADA, a targeted formal parameter of a called method is explicitly referenced by its name. For this purpose the <input> tag has a param attribute. A form must provide actual parameters exactly for the formal method parameters, either by user input or hidden parameters. In the present simple example this is fulfilled. In general the overall demand for type-safe calls of NSP methods causes sophisticated rules for writing forms and a new innovative widget set.

If the dialogue method newCustomer is invoked, inline Java code is executed. This Java code calls a user defined imported business method.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE nsp SYSTEM "nsp.dtd">

<nsp name="Shop">
  <head>
    <title>
      Shop
    </title>
  </head>
  <body>

    <java>
      import myBusinessModel.CustomerBase;
    </java>

    <method name="newCustomer">
      <param name="customer" type="String"/>
      <param name="age" type="int"/>
      <methodbody>
        <java>
          CustomerBase.createCustomer(customer,age);
        </java>
        <redirect callee="Shop">
        </redirect>
      </methodbody>
    </method>

    <method name="registration">
      <methodbody>
        <form callee="Shop.newCustomer">
          <input widget="textfield" param="customer"></input>
          <input widget="intfield" param="age"></input>
          <submit></submit>
        </form>
      </methodbody>
    </method>

    <method name="main">
      <methodbody>
        <link callee="Shop.registration">
          <linkbody>
            Customer Registration
          </linkbody>
        </link>
      </methodbody>
    </method>

  </body>
</nsp>
```

Note that a call to a method of the caller's hosting component may omit the component's name in the callee specification.

4 Language Description

4.1 Top-level Structure

An NSP component consists of a head and a body. The body is divided into methods. Java code may be placed inside <java> tags. Outside methods Java code is used for import statements or global variable declarations for instance. The signature of the component's methods is defined with appropriate <param> tags, which have attributes for specifying names and types of parameters. The method parameters are accessible in the inline Java code of the respective method.

In addition to <java> tags it is possible to use <javaexp> tags inside methods as a controlled variant of direct, i.e. Java coded, writing to the output stream. In NSP no special non-XML syntax for expression scriptlets like the JSP <%= %> signs is available. Because of that it is not possible to program inside NSP tag text, especially attribute values may not be computed. Element properties that may have to be provided dynamically, are supported by elements rather than attributes in NSP. As a result an NSP component is a valid XML document. Therefore NSP will benefit immediately from all new techniques developed in the context of XML. In particular, NSP can be used in combination with style sheet technologies [31].

4.2 Type System

In a typed programming language the call to a method must fit exactly the method signature. This notion is picked up but elaborated further due to the special needs of programming a web interface. In order to understand the type system of NSP one has to realize that in a scripted server page typically a whole block in the scripting language generates one HTML form in the output. In NSP the static type rules apply to the whole form, because within the NSP paradigm the whole form is the analogue of only a single method call. As an instructive example for this, have a look at the following code fragment.

```
1   {
2     int x;
3     x = 1;
4     x = 2;
5     x = 3;
6     m(x);
7   }
```

This is a correct program, though lines 3 and 4 are unnecessary and actually will be deleted from an appropriate optimizing compiler. In NSP a similar form declaration has to be considered wrong.

```
<method name="m">
  <param name="i" type="int"/>
  <methodbody>
  </methodbody>
</method>

<!-- form inside another method -->
<form callee="m">
  <java>
  for (int j=1;j<3;j++) {
    </java>
    <input widget="intfield" param="i"></input>
    <java>
  }
  </java>
  <submit></submit>
</form>
```

That is, in NSP all HTML code which is created dynamically by a form declaration, is not just considered as a block for capturing actual parameters comparable to line 3, 4 and 5 in the above code fragment. Instead it is considered as a method call offered to the user as a whole. It is comparable to line 6 in the above example. Precisely in this sense the form has to support the signature of the called method exactly. Altogether this leads to rules for writing NSP code which essentially is a mix of HTML dialect and scripting code. The aim of the subsection 4.2 is to elaborate these so called NSP coding rules for writing type-safe forms, which are designed in a way that the resulting notion of type correctness may be checked statically. Correct NSP code is very natural and the NSP coding rules are easy to learn, though the formal description may seem to be complex on first sight. Altogether NSP contributes the adaptation of typed programming discipline to the context of programming web interfaces.

The rules for writing type-safe forms are presented as a combination of declarative characterization and informal guidelines for coding through examples and counter-examples.

4.2.1 Input Capabilities

In NSP an actual form parameter may be provided by user input or by a hidden parameter. Note that we use the term ”input capability” for input fields, auxiliary input controls, and - a bit sloppy, though in accordance with usual HTML terminology - for hidden parameters, too. NSP offers at least the usual HTML form input capabilities but defines their usage with respect to type safety, i.e. a textfield may be used for String values for instance, comboboxes and radiobuttons for enumeration types, multiple select lists for array types. But NSP improves the HTML form controls in several ways, as will be outlined now.

Consider the following problems with HTML forms. In forms often so called required fields occur. These fields must be filled out by the user. With standard HTML the developer has to deal with required input fields explicitly. For example the submission of an invalid form may be prevented with client side scripting plus disabling the submit button as described in [11]. Alternatively a server side script may check if the actual required parameter is an empty string and must provide appropriate error messages and a new input capability if necessary. A similar problem arises with the entry of integer values. Usual scripts have to react on non-convertible values with extra dialogue.

In NSP these and similar problems are tackled from the outset. Consider the following examples.

```
1 <input widget="textfield" param="s" entry="required"></input>
2 <input widget="textfield" param="s" entry="optional"></input>
3 <input widget="integerfield" param="i" entry="required"></input>
4 <input widget="intfield" param="i"></input>
```

A textfield may be specified to be required (1). The encompassing form cannot be submitted, unless this field is filled out by the user. If a textfield is optional (2) and the field is not filled out, a null object becomes the actual parameter. For textfields, the entry attribute's default value is ”optional”. A textfield may target only a String value. For numbers extra widgets are provided. A Java Integer object may be gathered with an integerfield (3). Like a textfield the integerfield may be required or optional. Furthermore an entered value must be a number, otherwise it is not possible to submit the form. An intfield (4) is used to target parameters of type int. It behaves like an integerfield, but it is immutable required, because there exist no canonical mapping of an undefined value into a Java primitive type. The advanced developer can customize the default behavior of the NSP widgets. Further considerations led to a new innovative set of widgets in NSP. For example the JDBC API types [27] are supported. The widget set's implementation is discussed in subsection 5.1.

4.2.2 Forms and Links

Note that we use the term "basic type" for every Java primitive type, for every wrapper of a Java primitive type, for String, and for every JDBC API primitive SQL type throughout the paper. For a formal method parameter of basic type, there must be exactly one input capability in the resulting HTML form, i.e. in the form that is sent to the browser. The generating NSP form code must ensure this for every possible execution path.

Therefore in NSP input capability tags may occur within conditional control structures, as long as all alternative branches produce exactly one input capability for the respective parameter. Furthermore in NSP input capabilities for basic type values must not occur within loops. With these rules, the type safety with respect to basic types may be statically checked.

For example the following form declaration is considered wrong.

```
<form callee="m">
  <java>
  if (x==3) {
    </java>
    <input widget="intfield" param="i"></input>
    <java>
  }
  </java>
  <submit></submit>
</form>
```

Instead the following form declaration is a correct alternative.

```
<form callee="m">
  <java>
  if (x==3) {
    </java>
    <input widget="intfield" param="i"></input>
    <java>
  } else {
    </java>
    <hidden param="i" type="int">
      <value>
        815
      </value>
    </hidden>
    <java>
  }
  </java>
  <submit></submit>
</form>
```

Note that a static type system like NSP's cannot prevent such dynamic type errors, that result from using the output stream to send dynamically generated form fragments to the browser directly. Generally, using the output stream for sending HTML is considered bad style and may lead to dynamic errors. It is an NSP rule that the output stream must not be used in a way that corrupts the otherwise type-safe NSP system. A corrupted form that is caused by a prohibited use of the output stream, may contain too much input capabilities. This is just considered as a dynamic error, like division by zero. But even though such a dynamic type error should not occur, NSP provides sophisticated support for it. In the default case an appropriate exception is thrown. But beyond this a switch is offered by the tolerant attribute for every form. A tolerant form may have too much input capabilities for a parameter of basic type. Appropriate rules then guarantee that exactly one actual parameter is chosen. Note that it is not possible to switch off static type checking with the tolerant attribute. A form may only be tolerant with respect to provoked dynamic type errors.

In NSP the syntax for hyperlinks follows the form syntax. In NSP no tedious handling with special signs is needed in order to use hyperlinks with parameters.

```
<link callee="Shop.newCustomer">
  <hidden param="customer" type="String">
    <value>
      John Q. Public
    </value>
  </hidden>
  <hidden param="age" type="int">
    <value>
      32
    </value>
  </hidden>
  <linkbody>
    underlined link name
  </linkbody>
</link>
```

Links may only include hidden parameters as input capabilities.

4.2.3 Arrays

NSP offers sophisticated type-safe support for gathering arrays of values in forms. In the following form fragment, an Integer array parameter articleIds is targeted.

```
<form callee="m">
  <java>
  for (int i=0;i<8;i++) {
    </java>
    <input widget="integerfield" param="articleIds">
      <index>
        <javaexp>i</javaexp>
      </index>
    </input>
    <java>
  }
  </java>
  <input widget="integerfield" param="articleIds">
    <index>
      <javaexp>9</javaexp>
    </index>
  </input>
  <submit></submit>
</form>
```

All input capabilities that target the same array parameter together provide the submitted actual parameter. The index specifications must be unique. The greatest index determines the length of the array. In the example the invoked method receives an array of fixed length ten. Array elements that correspond to fields that were not filled out, are null objects. Furthermore, the ninth element is a null object, because no input capability exists for this index.

But NSP offers further features concerning arrays in forms. Consider for instance the following example.

```
<form callee="m">
  <input widget="integerfield" param="articleIds"></input>
  <input widget="integerfield" param="articleIds"></input>
  <input widget="integerfield" param="articleIds"></input>
  <submit></submit>
</form>
```

In this examples the index tags are omitted. Now the invoked method just receives actually entered values. If an optional field is not filled out, no object at all will occur for it in the actual parameter array. This NSP feature frees the programmer from tedious work in many cases. All joint input capabilities of a targeted array parameter must either provide a unique index or otherwise omit the index.

In contrast to parameters of basic type no restriction on the occurrence of input capabilities is necessary, because both arrays of length zero and arrays containing null objects are considered correct.

4.2.4 User Defined Types

NSP allows the usage of user defined types in forms. One possibility to target a formal parameter of user defined type and its fields is the usage of path expressions. The following example demonstrates how a complete object tree concerning a user defined type structure may be gathered by a form this way. User defined types that should be used in NSP forms must follow the Java Beans naming convention. We use an obvious notation for user defined types in the following examples.

```
Address {
  String street;
  int zip;
}

Customer {
  String name;
  Integer age;
  Address address;
}

<method name="newCustomer">
  <param name="customer" type="Customer"/>
  <methodbody>
  </methodbody>
</method>

<method name="registration">
  <methodbody>
    <form callee="newCustomer">
      <input widget="textfield" param="customer.name></input>
      <input widget="integerfield" param="customer.age"></input>
      <input widget="textfield" param="customer.address.street">
      </input>
      <input widget="intfield" param="customer.address.zip"></input>
      <submit></submit>
    </form>
  </methodbody>
<method>
```

Under all circumstances and for each type of a type structure, there must be either exactly one input capability for each of its fields of basic type or no input capabilities at all. Fields that have a user defined type must not have an input capability. Appropriate object references are generated automatically. Following a type structure path, the first time a type occurs that does not have any input capabilities, the generated object reference is the null object. Altogether these rules enable the usage of cyclic type structures in forms. This is exemplified in the following example.

```
A {
  A next;
  int i;
}

<method name="m">
  <param name="anA" type="A"/>
  <methodbody>
  </methodbody>
</method>

<method name="n">
  <methodbody>
    <form callee="m">
     <input widget="intfield" param="anA.i"></input>
     <input widget="intfield" param="anA.next.i"></input>
     <input widget="intfield" param="anA.next.next.i"></input>
     <submit></submit>
   </form>
  </methodbody>
<method>
```

Beside path expressions, NSP offers a parenthesis mechanism for targeting parameters of user defined type, that is similar to the "with" construct of MODULA2.

```
<form callee="newCustomer">
  <with param="customer">
    <input widget="textfield" param="name"></input>
    <input widget="integerfield" param="age"></input>
    <with param="address">
      <input widget="textfield" param="street"></input>
      <input widget="intfield" param="zip"></input>
    </with>
  </with>
  <submit></submit>
</form>
```

Beyond the usage for abbreviation, the "with" tag becomes an important construct if a formal array parameter of user defined type is targeted, because the input capabilities of fields that constitute a single array element must be grouped together. Not all layouts may be realized with this mechanism, because it is possible that a required layout structure contradicts the necessary structure of an XML document. Therefore NSP provides another low-level opportunity to group input capabilities together via tag keys.

4.3 Exceptions

NSP provides a flexible concept for exceptions concerning methods called across the net. Dialogue methods may react on the violation of constraints by gathering information in an exception object and throwing an exception. An exception handler analyzes the exception and proceeds. It is important that each calling form may specify another exception handler for a called method. The necessary reaction to an exception may depend on the caller, because of the existence of several different layouts for instance.

In the following example the method newCustomer checks the received data against a constraint. A detected constraint violation is memorized in an NSPException object. An NSPException object is a multi-valued String hashtable. A usual way to use it is to annotate the formal parameters of a method with one or more error messages. An NSPException object provides a rich set of appropriate query methods. The method newCustomer throws the exception by calling the static NSP API method raise. The form generated by the registration method uses an attribute to specify a handler. If newCustomer has been called by such a form and throws an exception, this is caught by regHandler.

A handler is related to a thrower, which is the respective form's callee. Following NSP conventions, regHandler must have the same signature as the related thrower, i.e. newCustomer, up to renaming of parameters, augmented by an NSPException parameter. If regHandler catches an exception, it receives the actual parameters that have been provided by the initially calling form and the exception object provided by newCustomer. Note that the form generated by regHandler recursively specifies regHandler as its handler.

```
<method name="newCustomer">
  <param name="customer" type="String"/>
  <param name="data" type="int"/>
  <methodbody>
    <java>
    if (check(data)){
      NSPException e = new NSPException();
      e.addError("data",ERROR);
      NSPException.raise(e);
    } else {
      // auxiliary code
    }
    </java>
    <redirect callee="main">
    </redirect>
  </methodbody>
</method>

<method name="registration">
  <methodbody>
    <form callee="newCustomer" handler="regHandler">
      <input widget="textfield" param="customer" entry="required">
      </input>
      <input widget="intfield" param="data"></input>
      <submit></submit>
    </form>
  </methodbody>
</method>
```

```
<handler name="regHandler">
  <param name="c" type="String"/>
  <param name="d" type="int"/>
  <param name="e" type="NSPException"/>
  <handlerbody>
    Dear Mister <javaexp> c </javaexp> ! <br/>
    <form callee="newCustomer" handler="regHandler">
      <hidden param="customer" type="String">
        <value>
          <javaexp>c</javaexp>
        </value>
      </hidden>
      <java>
      if (e.containsError("data",ERROR)) {
        </java>
        <!-- error message for -->
        <input widget="intfield" param="data"></input>
        <java>
      }
      </java>
      <submit></submit>
    </form>
    <link callee="exit">
      <linkbody>
        quit
      </linkbody>
    </link>
  </handlerbody>
</handler>
```

The NSP approach for handling constraint violation with a specially designed exception mechanism fosters low coupling and high cohesion. Take the following common dialogue pattern as an example: a form is redisplayed if its submission led to constraint violation, but with the difference that input fields that led to a constraint violation are emphasized, e.g. colored, and the remaining input fields contain the values entered before as default values. The NSP solution encompasses a helper method for displaying the form. This method is generic in so far that its signature contains parameters for the input field default values and arbitrary complex additional information on layout. The actual method which is called across the net immediately calls the helper method through NSP's server side call mechanism, which is explained below. This call passes appropriate actual default parameters. Similarly, the helper method is reused in the recursively called exception handler with appropriately prepared actual parameters.

4.4 Functional Decomposition

NSP provides tags for the usual redirect and include capabilities known from the Servlet API's RequestDispatcher, but this time in the method oriented NSP style.

```
<method name="helloWorld">
  <methodbody>
    <redirect callee="m">
      <hidden param="head" type="String">
        <value>
          Hello
        </value>
      </hidden>
    </redirect>
  </methodbody>
</method>

<method name="m">
  <param name="head" type="String"/>
  <methodbody>
    <call callee="n">
      <hidden param="message" type="String">
        <value>
          <javaexp>head</javaexp> World !
        </value>
      </hidden>
    </call>
  </methodbody>
</method>

<method name="n">
  <param name="message" type="String"/>
  <methodbody>
    <javaexp>message</javaexp>
  </methodbody>
</method>
```

As demonstrated in the above example, in NSP an include capability is given consequentially by a server side call of a dialogue method. Again NSP gains from its solid conceptual basis: if the Servlet API is used it must be ensured that a servlet targeted by the include capability does not send a head. This problem is tackled in NSP from the outset, because method bodies syntactically never contain heads. If and only if a method is called across the net, the component's head is pushed into the output stream.

5 Operational Semantics

An operational semantics of NSP may be given by specifying a transformation of NSP components to JSP pages and auxiliary technology. This section serves as an outline of such a semantics. The operational semantics considers only NSP components that are correct with respect to NSP's static semantics.

5.1 Transformation of NSP to JSP

Each NSP component is mapped to a JSP page with the same name.

Each method within an NSP component is mapped to a block in the JSP page. All these blocks reside within a switch, i.e. within an appropriate structure of Java conditional expressions that select a method block depending on the value of a method identifying key. Every NSP formal parameter is mapped to a block local variable.

NSP forms are mapped to JSP forms. The targeted method of an NSP form is split into a targeted component and a targeted method name. The targeted component becomes the action target of the respective JSP form. The method name becomes the value of the method identifying key, which is transmitted as a hidden parameter. User input and hidden parameters are transmitted as key value String pairs. Based on the transmitted method identifying key an invoked JSP branches to the appropriate method block. The transmitted parameter stream is parsed, parameters of basic type are converted to their respective Java types, in the case of user defined types object nets are instantiated, in the case of arrays the several array points are gathered. The result of this process is assigned to the block local representatives of the NSP formal method parameters.

The NSP widget set is realized both with client side and server side scripting technology. A switch is offered. As an example consider the realization of an Integer input field with server side scripting technology. During the parsing of the transmitted parameter stream, it is checked whether the parameter is convertible. If not, the calling form is redisplayed with correctly entered values as default values. In order to treat required input fields, the entry specifications of the form fields must be transmitted and evaluated. The client side realizations guarantee performance and offer fancy look and feel based on sound and color.

Within the described framework, the mapping of other NSP features, like functional decomposition and exceptions, is straightforward.

5.2 Implementation

The syntax of NSP is given by an XML DTD. The reference implementation straightforwardly realizes the operational semantics outlined in subsection 5.1 using Sun's Java API for XML Processing JAXP [13] and the compiler generator ANTLR [19], based on both standard compiler construction techniques [25] and proposed object oriented patterns [10].

6 Reverse Engineering of User Dialogues with Gently

NSP also aims at improving the quality of documentation of the system interface. For this purpose, NSP allows reverse engineering, namely the automatic generation of a documentation of the whole system interface in the easy to read specification language Gently. An earlier version of Gently was developed by the authors as a specification language for forward engineering of system dialogues [8] [9]. Gently extracts from all pages of a system interface the methods with their signatures together with the links and forms. A further very useful information generated in Gently is the list of callers for each method. The generated Gently document is the suitable place for comments, which are passed through from the NSP files, and therefore is amenable to proposed specification techniques like [14] or [28], too. The standard reverse engineering tool of java, namely javadoc, is not suitable for documenting servlets. Applying javadoc to the customized server script classes leads only to the documentation of the technical parameters HTTPRequest and HTTPResponse. The interesting parameters, which are significant for the business logic, namely the HTML form parameters provided by the forms or links calling the page, cannot be documented automatically.

The following brief example is to give an impression of Gently and the way it fosters the understanding of systems with complex linkage structure.

```
nsp Task {
  taskForm(){
       called by link 7 from Home.menu
    and
       form 1 calls Service.newCustomer(
                                  hidden employeeId,
                                  textfield topic,
                                  datefield begin,
                                  datefield end
                                 )
    or link 1 calls Home.main()
  }
}

nsp Service {
  /* Object Constraint Language
     pre:    (first.before(last) or first.equals(last))
          and Task.allInstances->forAll(
             end.before(first) or begin.after(last)
          )
  */
  newCustomer(int id, String topic, java.util.Date first,
                                    java.util.Date last){
       called by form 1 from Task.taskForm
    and
       redirect Home.main()
  }
}
```

7 Further Work

NSP is the product of an in depth analysis of form based user interfaces. With the NSP concept a complete and consistent programming model for server pages is created. The ongoing work is now directed towards developing a dialogue pattern catalogue for NSP, which leverages the potentials of NSP for rapid development. Each of the following topics will be addressed by an appropriate pattern set:

- The support of a meaningful history within a form based dialogue is a nontrivial task, since old pages are invalid after updates. A set of patterns is provided for that problem. For example: the back button used after a form input should either lead to a view of submitted forms, e.g. after paying a shopping cart the back button leads to the view of ordered carts. A fully customized substitute for the browser history will be given, which is provided by the server.
- More generally, support of comprehensive navigation elements, as they are discussed and evaluated in literature for homepages [15][16], pose design problems in the context of form based system interfaces.

- NSP exceptions support the use of different layouts. For example, suppose that buying a shopping cart leads from the cart layout to the order tracking layout. Now, if an exception occurs during ordering, control remains in the cart layout. Intuitive handling of special cases within the system interface will be supported.
- Session support: good practices for the realization of sessions with and without login, and for the use of session objects.
- Stateful forms: suspension of form input as well as multi-paged forms with tab control look and feel, which rely solely on current technology.
- Bookmark support: within a form based HTML dialogue that produces generated lists or tables the user still wants to use the bookmark mechanism. However, the support of the mechanism must comply with the system designer's intentions. Good practices on when to support bookmarks will be stated. Furthermore advices on the decision between HTML GET and HTML POST usage, and on the usage of the redirect mechanism, will be given.

8 Related Work

8.1 NSP Components versus Different Component Notions

In this paper we use the term component for the NSP analogue of a JSP page. In order to justify this we first identify three accepted non-mutual exclusive notions of the concept software component in computer science. One of them, which is rather found in academic research, defines components as subsystems with a defined interaction model [1]. These components are either the result of functional system decomposition [20] or the development of strong reusable COTS functionalities [26]. The second one is the notion of components as entities that fit to a certain object-oriented technology that addresses crosscutting architectural concerns like transactions, persistency, or security for instance [21]. The third one defines a component just as a deployed object with respect to an appropriate deployment model [23]. We stress that an NSP page is a deployed static object-oriented entity of coarse granularity, that relies on container services for web computing. This justifies the usage of the term component for an NSP page in this paper.

8.2 Web Services contra HTML/HTTP User Dialogues

Web Services [22] are widely regarded as a major technological shift in the usage of the web. Web services are primarily discussed in the B2B domain. Web services can be used synchronously in a request/response style or they can be asynchronous, message oriented. Web services possess a type system, the Web service definition language [29].

At first glance, web services may seem inspired from and similar to user interfaces, using the same protocol, namely HTTP. With respect to the NSP static type checking we can identify a clear difference between web services and HTTP/HTML user interfaces: in web services, synchronous or asynchronous, there is no necessary type relation between different messages, although there may be a type system. The types of different messages can be chosen freely according to the needs of the business case.

In HTML dialogues however, if the user is supposed to send a form with data to the server, then a page containing this form must have been previously sent to the user. More generally speaking, we have a necessary relation between a typed user request and a systems response before that request. This relation is based in the mechanism itself.

In other words, in HTML dialogues, the type information is not transmitted once and used for all interactions since that, but it is transmitted before every typed request, and it is directly transformed into the displayed form, which the user actually fills out.

This is in our view not an accidental design mismatch between web services and HTML/HTTP dialogues, but a fundamental difference caused by the fact that web services are accessed automatically, and HTML dialogues are designed for end users.

The end user can react to every input form offered to him immediately, the web service can only repeat those interaction sequences that have been in principle laid out and validated once manually beforehand.

On the other hand it is clear that web service technologies are not able to address this unique feature of NSP.

8.3 Other User Interface Technologies

NSP delivers genuinely new features, and is therefore not comparable to other approaches [2] [5] [6] [24] [30]. However, NSP can be easily mistaken to be a variant of architectural approaches or approaches for separating logic and presentation. We therefore give an overview of current efforts in these two areas, which receive widespread attention. However one should note that not a single one of these projects deliver any of the unique selling points of NSP.

Server side scripting technologies, like ASP, JSP and PHP allow to embed script code into HTML, or more generally XML. None of these technologies offer NSP's static type checking. Another approach is chosen by the WebMacro [24] system, which allows the separation between Java and HTML by following the Model-View-Controller pattern. WebMacro uses the Java reflection mechanism to access Java Beans from HTML Templates. Again, WebMacro does not offer a specialized support for type-safe dialogues.

WebMacro leads over to the architectural frameworks for web interfaces, which allow to create presentation layers that follow a certain software architecture. An example is Struts [6], a Framework for User Interfaces, which results in a peculiar Model 2 Architecture. Another Model 2 architectural framework is Barracuda [2]. Both do not offer the type check facilities of NSP. Another framework, which lies explicitly the emphasis on separation of content and presentation, is the Cocoon [5] framework A WWW working draft for a next generation of forms is XForms [30], a new form standard, which does not deliver in itself the features of NSP.

9 Conclusion

The abstract notions developed in NSP are independent from the underlying scripting language. NSP does not impose a special architecture onto the system, it does not prescribe a server side technology, it does not appear on the network protocol, and therefore needs neither client side installations, nor does NSP reduce performance. In effect, NSP causes no burden for development, openness or availability. NSP has no necessary source code footprint, no network footprint, no architecture footprint, no footprint on the running system. NSP allows the use of complex types in input forms. System interfaces developed with NSP are automatically documented by the reverse engineering tool and language Gently. The exception mechanism of NSP allows to catch exceptions virtually within the HTML form, hence in the browser: if a HTML input form calls a dialogue method for processing the data, the developer can specify within that form a catch block, which in case of an exception redirects the form data to an appropriate handler.

NSP is the only approach for HTML interfaces that allows static type checking of links and forms against called server scripts and delivers parameters to server scripts type-safe and in the native types of the language. NSP makes it possible for the first time during the evolution of web interface frameworks, that the whole uppermost layer for the direct conversational interaction between browser and server is freed of type and linkage errors. To that purpose, NSP has a unique feature: it can statically check, whether the dynamically generated code will be type-safe. This is made possible by the NSP coding rules.

References

[1] Allen, R.J.; Garlan, D.; and James Ivers. Formal Modeling and Analysis of the HLA Component Integration Standard. In Proceedings of the Sixth International Symposium on the Foundations of Software Engineering, November 1998

[2] Barracuda. http://barracuda.enhydra.org/.

[3] Brown, S. et. al. Professional JSP, 2nd edition. Wrox Press, April 2001.

[4] Cardelli, L. Type systems. In: Handbook of Computer Science and Engineering. CRC Press, 1997.

[5] Cocoon. http://xml.apache.org/cocoon/.

[6] Malcolm, D. Struts, an open-source MVC implementation. IBM developerWorks, February 2001.

[7] Davidson, J.D.; and Coward D. Java Servlet Specification, v2.2. Sun Press, 1999.

[8] Draheim, D.; and Weber, G. Specification and Generation of JSP Dialogues with Gently. In Proceedings of NetObjectDays 2001. tranSIT, ISBN 3-00-008419-3, September 2001.

[9] Draheim, D.; and Weber, G. Gently - A Specification driven Generator for Java Server Pages. http://www.inf.fu-berlin.de/projects/gently/.

[10] Gamma, E. et al. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[11] HTML 4.01 Specification. W3C, 1999.

[12] Kassem, N.; and the Enterprise Team. Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition. Sun Microsystems, 2000.

[13] Mordani, R.; Davidson, J.D.; and Boag, S. Java API for XML Processing Specification, v1.1. Sun Microsystems, 2001.

[14] Meyer, B. Applying "design by contract". IEEE Computer, 25(10):40-51, October 1992.

[15] Nielsen, J. User interface directions for the Web. In Communications of the ACM 42(1):65-72, January 1999.

[16] Neerincx, M.A.; Lindenberg, J.; and Pemberton, S. Support concepts for Web navigation: a cognitive engineering approach. In Proceedings of the Tenth International World Wide Web Conference, May 2001.

[17] Pelegri-Llopart, E.; and Cable, L. Java Server Pages Specification, v.1.1. Sun Press, 1999.

[18] Rosenfeld, L.; and Morville, P. Information Architecture for the World Wide Web. O'Reilly, 1998.

[19] Schaps, G.L. Compiler Construction with ANTLR and Java. In Dr. Dobb's Journal, March1999.

[20] Shaw, M.; and Garlan, D. Software Architecture - Perspectives on an emerging Discipline. Prentice-Hall, 1996.

[21] Shannon, B. et al. Java 2 Platform, Enterprise Edition: Platform and Component Specifications. Addison-Wesley, 2000.

[22] SOAP. http://www.w3.org/TR/SOAP/.

[23] Szyperski, C. Component Software - Beyond Object-Oriented Programming. Addison-Wesley and ACM Press, 1998.

[24] Webmacro. http://www.webmacro.org/, 2002.

[25] M.Waite, W.M; and Goos, G. Compiler Construction. Springer-Verlag, 1984.

[26] Whitehead, E.J. et al. Software Architectures: Foundation of a Software Component Marketplace. In Proceedings of the First International Workshop on Architectures for Software Systems, April 1995.

[27] White, S. et al. JDBC API Tutorial and Reference. Addison-Wesley, 1999.

[28] Warmer, J.; and Kleppe, A.G. The Object Constraint Language. Addison-Wesley, 1999.

[29] Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[30] XForms 1.0. http://www.w3.org/TR/xforms/.

[31] The Extensible Stylesheet Language (XSL), http://www.w3.org/Style/XSL/. Addison-Wesley, 2000