Chapter 3

NSP Coding Guidelines

This chapter defines the NSP support for gathering array data and data of user defined type. The NSP coding guidelines and rules are elaborated.

The object of the NSP coding guidelines and rules are dynamic document fragments with respect to the generated user interface description. There are two kinds of coding guidelines and rules. The parameter guidelines and rules are dealing with client page type safety, the structure guidelines and rules are dealing with client page description safety. The guidelines are declarative characterizations of valid NSP code, they are informal descriptions of demands placed on NSP code. The NSP coding guidelines are accompanied by NSP coding rules that define how to achieve these demands. The resulting notion of correctness can be checked statically and guarantees client page type and description safety.

The NSP parameter guidelines and rules contribute the adaptation of typed programming discipline to the context of server pages development.

The NSP coding guidelines target the developer: correct NSP code is very natural and the NSP coding guidelines are easy to learn. The NSP coding guidelines and rules are formalized in chapter 7 by a Per Martin-Löf style type system.

In the discussion of coding guidelines passive NSP code is considered as generated code. Such passive NSP code is a variant of XHTML code. It is ensured that generated NSP tags will cause the generation of valid XHTML eventually. The mapping of NSP tags to valid XHTML is given canonically and described further in chapter 5.

3.1 The Object of Parameter Guidelines

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 based on a server pages technology. 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 form in the output. In NSP the static typing 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 code fragment in listing 3.1.

Listing 3.1

```
01   {
02     int x;
03     for(int i=0; i<3; i++){
04       x = 810;
05     }
06     m(x);
07   }
08
09 void m(int x){}
```

This is a correct program, though lines 3 and 5 are superfluous and actually can be deleted from an optimizing compiler. In NSP a similar form declaration has to be considered wrong; in the form generated by listing 3.2 the user may enter three int data, however the targeted server page expects exactly one int value. That is, in NSP user interface description code that is created dynamically by a form declaration, cannot be considered just as a block for computing actual parameters comparable to line 3, 4 and 5 in the above code fragment. Instead of this it is considered as an editable 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.

Listing 3.2

```
01 <form callee="m"><java>
02   for(int i=0; i<3; i++){</java>
03     <input type="int" param="x"></input><java>
04   }</java>
05   <submit></submit>
06 </form>
07
08 <nsp name="m">
09   <param name="x" type="int"></param>
10   <html>
11     ...
12   </html>
13 </nsp>
```

The object of the parameter guidelines is dynamic NSP code that generates form content.

Definition 3.1.1 (NSP Parameter Guideline) *An NSP parameter guideline describes, what kind of and how many controls must be provided for what kind of formal parameter. The statement that a certain kind and number of control must be provided for a certain kind of formal parameter means: the body of every form targeting a web signature that encompasses such a formal parameter will always only generate the specified kind and correct number of controls. These controls are said to target the formal parameter in quest.*

In section 3.3 so called parameter rules are given for the parameter guidelines. The NSP parameter rules are the coding rules that help to achieve the parameter guidelines. They describe the effect of conditional control structures, loops, and sequencing of document parts on the guaranteed number of a certain kind of control.

The first fundamental NSP parameter guideline is independent from the type of the targeted formal parameter: a generated control must not target a formal parameter that does not belong to the targeted web signature.

3.2 Basic Parameter Guidelines

In this section the NSP parameter guidelines for formal parameters of basic type are given. NSP distinguishes between basic types, array types and form message types. The notion of form message type is introduced in section 3.5: a user defined type may be explicitly defined as a form message type; then NSP offers tag support for gathering composed data of form message type. In NSP/Java the basic types encompass the Java primitive types and every object type, i.e. every user defined type or arbitrary Java API type. The parameter guidelines explained in this section are summarized in Figure 3.1. Parameter guidelines for arrays are explained in section 3.4.

Formal Parameters of Primitive Type

For a formal int parameter either one type-correct input control, hidden control, single select menu or at least two type-correct radio buttons must be provided.

A form encompassing a type-correct input control cannot be submitted, if the data entered by the user is not a number. Similarly a formal int parameter implicitly requires an entry by the user. The necessary concept of active direct input controls has been explained in section 2.3.1. A hidden control may target a formal int parameter, as long as its contained expression has int as type; such an expression cannot evaluate to a null object, because int is a primitive Java type. For the same reason a single select menu may target a formal int parameter. Alternatively a set of radio buttons may target the parameter. It must contain at least two radio buttons. An int radio button set or single select menu always produces a unique value, because of the concept of NSP active single select controls introduced in section 2.3.3.

A formal int parameter may be constrained to be targeted by an interactive widget. The param tag has a widget attribute, which may be set to required

for this purpose. If a widget is required, the targeting control must not be a hidden parameter. Despite this exception the parameter guideline is the same. Requiring an interactive widget is a powerful concept that allows for the precise specification of a dialogue method as editable method call offered to the user by means of a web signature.

The int type is the only NSP/Java instance of the general notion of an NSP direct input supported primitive type. Other language amalgamations may support more primitive types by a direct input widget in a way that follows the parameter guidelines just introduced.

The dual notion of an NSP direct input supported primitive type is that of an NSP ordinary primitive type. The float type is an example for such a type. For a formal float parameter either exactly one type-correct hidden control, single select menu or at least two type-correct radio buttons must be provided.

Formal boolean Parameters

The boolean type needs specific parameter guidelines. It is supported by an instance of the check box mechanism as explained in section 2.3.4. The boolean type is an NSP specific primitive type.

For a formal boolean parameter either exactly one check box or one type-correct input hidden control must be provided. If the widget attribute is set to required, the formal parameter must be targeted by exactly one check box. It is not allowed to use other controls like direct input or a select menu for targeting a boolean formal parameter. We feel that with respect to user interaction the boolean type is inextricably connected to the notion of check box and this is expressed by the current parameter guideline.

Formal Object Type Parameters

Every non-primitive Java type is termed object type. For a formal parameter of object type at most one type-correct hidden control, single select menu or at least two type-correct radio buttons must be provided. Most importantly it is allowed not to provide any control for a formal parameter of object type. If no control is provided, the system will fill in a null object on submission automatically.

For a formal parameter of object type the widget attribute may be set to required, too. Then exactly one type-correct single select menu or at least two type-correct radio buttons must be provided. It is not longer allowed just to omit any kind of control: the purpose of the required widget attribute is to ensure that an interaction capability is offered to the user. In general there is no direct input capability for an object type and possible interaction capabilities are single select menus or a set of radio buttons. However, as opposed to formal parameters of primitive type, requiring a widget does not guarantee valuable data on submission, because nothing prevents a data item provided by e.g. a select menu to be a null object. This leads to another kind of NSP object type.

As for primitive types, it is distinguished between direct input supported object types and ordinary object types. Examples for direct input supported object types in NSP/Java are the Integer type and the JDBC API type Date [186]. For a formal parameter of such type the above parameter guidelines are applicable, except for the possibility to additionally target a parameter by the type-correct input control. Beyond this for such types the formal parameter tag has an entry attribute. This entry attribute may be set to required. Then for a formal parameter exactly one type-correct input control must be provided. Furthermore the system statically ensures that every targeting input control dynamically ensures data entry. As discussed above, only this can prevent a null object from being submitted.

Formal String Parameters

The Java type String is another example for an object type that is supported by an active direct input control. Actually it has three such controls, the usual input field, a password field and the text area control. The parameter guidelines are the same as the ones for types like Integer or Date, up to the additional input controls that are handled the same way as the input field. The only subtle difference concerning the input controls has already been described in section 2.3.1: no null object will be transmitted anytime.

Furthermore it can be specified that a formal String parameter must be targeted by the specific password control or by the specific text area control. The widget attribute is used for this purpose. If it is set to password exactly one password field must be provided, and analogous for the text area widget.

3.3 NSP Parameter Rules

An NSP parameter guideline is a demand on the kind and number of controls generated by a piece of NSP code that targets a certain kind of formal parameter. This section gives rules that define how to achieve these demands.

Determining the number of a certain control in a static part of an NSP document is trivial. But NSP code is a mixture of static and dynamic parts. However, in a valid NSP document only Java code and control tags are relevant with respect to the parameter rules[1], because all the tags concerning layout and the tags that switch between active and passive document parts have no influence on the control number.

Indeed only a Java subset is relevant; the parameter rules can be given with respect to just a few concepts, i.e. sequencing, switch structures and loops.

Consider the NSP code fragment in listing 3.3. Lines 1 to 6 generate four hidden parameters that target the formal parameter p1. Thereby it does not matter that the first two hidden parameters are provided merely by composition of document parts, whereas the latter are interlaced by sequences of Java code.

[1] The object tags explained in section 3.5 are relevant with respect to the parameter rules, too.

Table 3.1: Valid Controls. This table visualizes the NSP parameter guidelines for formal parameters of basic type by specifying kind and number of valid controls for each formal parameter type. For every type only the fields that are filled out are the valid controls. A formal parameter of a certain type must be targeted by one of the valid controls, whereas the specified number must be accomplished. The types int, float and Integer are representatives for the classes of NSP direct input supported primitive types, NSP ordinary primitive types and direct input supported object types, respectively. ObjectType stands for every ordinary object type.

	input field	hidden	single select menu	radio button	value check box	select menu	password	text area	check box
int	1	1	1	≥2					
int widget=required	1		1	≥2					
float etc.		1	1	≥2					
boolean		1							1
boolean widget=required									1
Integer (Date)	0,1	0,1	0,1	0,≥2					
Integer widget=required	1		1	≥2					
Integer data=required	1								
String	0,1	0,1	0,1	0,≥2				0,1	
String widget=required	1		1	≥2				1	
String data=required	1							1	
String widget=password							1		
String widget=textarea								1	
ObjectType		0,1	0,1	0,≥2					
ObjectType widget=required			1	≥2					

Furthermore the switch control structure and the loop must to be taken into account. However selectors, branching conditions, and loop conditions cannot be considered at compile time; the parameter rules must be independent from them. Altogether we need to determine the number of generated controls with respect to the NSP code skeleton in listing 3.4, which consists of relevant structure and elements only. Sure, it is not possible to determine the exact numbers in general, but it is safe, that exactly four hidden parameters are generated that target p1, at most one input control is generated that targets p2[2], at most one input control is generated that targets p3, and an arbitrary number of input controls is generated that target p4.

In Java there exist three different kinds of conditional control structure, i.e. the if-structure, the if-else-structure, and the switch structure. For the purpose of NSP parameter rules it is necessary to distinguish between completed switches and uncompleted switches. A completed switch is one that is ended by a default branch while an uncompleted switch is the opposite. The if-structure can be construed as a uncompleted switch with just one branch. The if-else-structure can be construed as a completed switch with two branches, whereas the else branch becomes the default branch of the switch. As a coding convention, in NSP code it is forbidden to end a switch branch without a break statement[3]. There are three different kinds of loops in Java, the for-loop, the while-loop and the do-while-loop. From the viewpoint of the NSP parameter rules the different kinds of loops are the same. Furthermore the NSP parameter rules do not distinguish between sequencing of NSP blocks that stems from Java code sequencing and such sequencing that stems from composing document parts.

Definition 3.3.1 (NSP parameter rule) *The NSP parameter rules describe how many controls of a certain kind are generated by a piece of NSP code. The number of generated controls is described by a lower and an upper bound on the exact number of generated controls. The NSP parameter rules are given in terms of building blocks and sequences of building blocks. The basic building blocks are the controls in quest. The only further building blocks are completed switch structures, uncompleted switch structures and loop structures. A completed switch is a switch that is ended by a default branch. All case branches and the default branch together are named the branches of the switch. An uncompleted switch is a switch that is not ended by a default branch.*

[2] At most one input control is generated that targets p2. This is an example for a type error: if a formal int parameter is targeted it must be ensured that exactly one appropriate control is provided.

[3] No doubt, it is possible to elaborate rules for switches with a fallthrough facility, but they are unnecessarily complicated. We argue that mixtures of branches that end with a fallthrough and branches that end with a break lay in a gray area between structured and unstructured code. We argue that there are no urgent examples that rely on the use of fallthrough. Interesting examples that rely on fallthrough live in the world of code bumming like e.g. Duff's device [153] (interestingly Duff's device is no valid Java Code [79] anyway). Convenient NSP parameter rules can only be given in terms of completely structured code. For example in every language amalgamation goto statements would be forbidden [54]. Anyway in Java no goto statement is realized (though goto is a reserved keyword).

These are the parameter rules:

- Single control. A single control of a certain kind generates exactly one control of this kind in the output user interface description language.
- Sequence of building blocks. Assume that every building block is safe to generate at least a specific lower bound of a fixed certain kind of control. Then it is safe, that the sequence will generate at least as much controls as the sum of all these single lower bounds. Analogously for upper bounds.
- Completed switch structures. Assume that every branch is safe to generate at least a specific lower bound of a fixed certain kind of control. Then the switch structure is only safe to generate as least as much controls as the smallest of all these single lower bounds. Analogously, if it is safe that every branch generates at most a specific upper bound of a fixed certain kind of control, then it is only safe that the switch structure generates at most as much controls as the highest of all the single upper bounds.
- Uncompleted switch structures. With respect to upper bounds on the number of generated controls the parameter rule for uncompleted switch structures is the same as the parameter rule for completed switch structures. With respect to lower bounds it must differ. Nothing can be assumed about the number of controls of any kind that will be generated at least by an uncompleted switch structure, because in general it is always possible, that none of the branches is selected.
- Loop. If it is safe that a loop body does not generate a certain kind of control, the loop trivially will not generate this control, too. If it is safe that the loop body generates at least one control of a certain kind, it is safe that the loop will generate an arbitrary number of this kind of control. Nothing more can be assumed about the number of controls generated by a loop, i.e. the lower bound is zero and no upper bound can be determined.

From the NSP parameter rules the NSP developer can derive compound rules that define how to achieve the complex demands described by the NSP parameter guidelines. As an example we define how to write code for a form that targets a formal int parameter. The respective parameter guideline prescribes, that such code must always generate either one type-correct input control, hidden control, single select menu or at least two type-correct radio buttons.

If the code is a sequence of blocks, there are three valid possibilities. The first is that exactly one of the blocks generates one type-correct input control, hidden control or single select menu. The second is that at least one of the blocks generates at least two radio buttons. The third is that at least two blocks generate at least one radio button. If the code is a completed switch, all branches of it must provide the correct number of valid controls. Controls targeting the formal parameter in quest must not occur in a branch of an uncompleted switch that is contained in the code, because it is not decidable whether such a switch generates the control or not. Similarly such a control must not occur inside a

loop that is contained in the code. The latter might seem to be the most severe constraint, but we argue that it does not prevent the developer from writing code for all reasonable applications. Placing a control inside a loop obviously has the reason of gathering array data; NSP parameter guidelines for arrays are given in section 3.4.

Listing 3.3

```
01 <hidden param="p1">new Person();</hidden>
02 <hidden param="p1">new Person();</hidden><java>
03 x:=1;</java>
04 <hidden param="p1">new Person();</hidden><java>
05 x:=2;</java>
06 <hidden param="p1">new Person();</hidden><java>
08 switch (y) {
09   case (y==1) : </java>
10     <input type="int" param="p2">
11     </input><java>;break;
12   case (y==2) : </java>
13     <input type="Integer" param="p3">
14     </input><java>;break;
15   case (y==3) : for (int i=0; i<fooBound; i++) {</java>
16                   <input type="String" param="p4">
17                   </input><java>
18                 };break
19 }</java>
```

Listing 3.4

```
01 <hidden param="p1">new Person();</hidden>
02 <hidden param="p1">new Person();</hidden>
03 <hidden param="p1">new Person();</hidden>
04 <hidden param="p1">new Person();</hidden>
05 switch () {
06   case () : <input type="int" param="p2">
07   case () : <input type="Integer" param="p3">
08   case () :
09      for () {
10        <input type="String" param="p4">
11       }
12 }
```

3.4 Parameter Guidelines for Arrays

A web signature may encompass formal array parameters. A formal array parameter may be targeted by an arbitrary mix of controls, as long as all controls are type-correct and valid with respect to the contained items' type. The valid controls for a formal array parameter encompass the controls that are valid for a completely unconstrained formal parameter[4] of the respective contained items' type. Furthermore type-correct value check boxes and select menus are valid controls for array parameters. The number of controls is not constrained for all kinds of controls, except for radio buttons. If a radio button targets a formal array parameter there must be at least a second radio button targeting the same parameter. The valid controls for the several array types are summarized in Figure 3.2.

For example, a formal int array parameter may be targeted by an arbitrary mix and number of type-correct input controls, hidden controls, single select menus value check boxes, select menus and perhaps a set of at least two radio buttons.

A formal parameter of array type is optional. If no control targets the parameter the system will automatically pass a null to the dialogue method on submit. It is not possible to give requirements for a formal array parameter. The emphasis of the NSP array mechanism is on the support for gathering array data in forms. If a constrained dynamic data structure is desired, complex message types and their respective facilities described in section 3.5 may be used.

NSP offers sophisticated support for gathering array data in forms. Listing 3.5 shows a form that targets a dialogue method that excepts an Integer array x and an int array y. Nine input fields are generated for the parameter x. Input widgets for Integer values are optional by default. If the user does not enter a value into a field this field is just ignored. All fields that have been received valid user data together provide the submitted actual array parameter. If the user did not enter any data a null object will be provided by the system for the parameter x on submission. Another nine input fields are generated for the parameter y. Input widgets for int values are required by default. The user must enter valid data in all the fields, otherwise the form cannot be submitted. The submitted array is guaranteed to have a length of nine.

As a second opportunity it is possible to explicitly specify the index of an array item. For this purpose the input field tag may contain an index element. The content of such an index element must be a Java int expression. If explicit definition of array element indexes is chosen, the specification should be unique. Furthermore the specification should be complete. Array items, that have a redundant index or don't have an explicit index are attached arbitrarily to the array.

Listing 3.6 is oriented towards the example from listing 3.5. A loop generates nine input fields for the parameter x. This time fields not filled out by the user,

[4] An unconstrained formal parameter is a parameter for which no widget is required or specified and no entry is explicitly required.

Listing 3.5

```
<form callee="N"><java>
  for {int i=0; i<9; i++){</java>
    <input type="int" param="x"></input><java>
  }
  for {int i=0; i<9; i++){</java>
    <input type="Integer" param="y"></input><java>
  }</java>
  <submit></submit>
</form>
```

cannot simply be ignored. Instead of this a null objects are inserted at the specified positions. Another input field is generated behind the loop. It has index 10. There is no input field with index 9 targeting the parameter x. If the form is submitted, again a null object will be inserted as ninth element. Another ten input fields are generated for the parameter y. These fields must be filled out. Again there is no input field with index 9 targeting the parameter y. If the form is submitted, this will result in an dynamic type error, because the system must fail to insert an appropriate value for this missing element of primitive type.

Listing 3.6

```
<form callee="N"><java>
  for {int i=0; i<9; i++){</java>
    <input type="Integer" param="x">
      <index> i </index>
    </input><java>
  }</java>
  <input type="Integer" param="x">
    <index> 10 </index>
  </input><java>
  for {int i=0; i<9; i++){</java>
    <input type="int" param="y">
      <index> i </index>
    </input><java>
  }</java>
  <input type="int" param="x">
    <index> 10 </index>
  </input>
  <submit></submit>
</form>
```

Table 3.2: Valid Controls. This table visualizes the NSP parameter guidelines for formal array parameters.

	input	hidden	single select menu	radio button	value check box	select menu	password	text area	check box
int [ ]	×	×	×	×	×	×			
float [ ] etc.		×	×	×	×	×			
boolean [ ]		×							×
Integer [ ] (Date)	×	×	×	×	×	×			
String [ ]	×	×	×	×	×	×		×	
ObjectType [ ]		×	×	×	×	×			

3.5 Parameter Guidelines for Form Messages

In NSP objects of user defined types can be used as hidden parameters, select menu items, and radio button or check box values. But NSP offers tag support for gathering data of user defined type in forms, the so called NSP form message mechanism. Input or selection capabilities for the single attributes of an object or of a complex object net may be offered to the user. Thereby a special object element, that resembles the with construct in MODULA-2 [187], enables the construction of data records[5].

In order to be supported in the way described above a user defined type must be explicitly marked as a form message type[6]. In general a form message is a record of attributes. The attributes are the fields that are supported by the form message mechanism. The user defined type underlying a form message type may have auxiliary fields[7]. Therefore, for every programming language amalgamation a naming convention must be defined for form message types, that distinguishes attributes from auxiliary fields. In NSP/Java the naming convention of the Java Beans [85] component model is chosen. A form message type is a Java Bean. A form message type attribute is a Java Bean property.

In this chapter we visualize form message types as UML class diagrams [13]. These diagrams are drawn within a defined perspective, which is a kind of specification perspective in the sense of [74][41]. The perspective abstracts away

[5]Similarly the group element of the XForms[69] technology enables the construction of semi-structured data entered by the user.

[6]In a concrete language amalgamation marking a user defined type as a form message type may be defined by a naming convention or as implementing a marker interface.

[7]So far visibility mechanisms are not an issue, they must be considered with respect to concrete programming language amalgamations only.

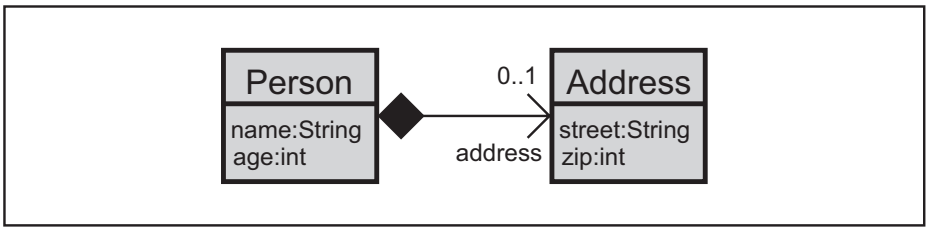

Figure 3.1: Example Form Message Type.

from possible concrete naming conventions. Only form message type attributes occur in the diagram, i.e. auxiliary fields are not visualized. Attributes that have form message type are visualized as associations, all other form message type attributes are visualized as UML object attributes. The associations are navigated and carry the attribute's name as role name. No methods occur in the diagrams. For instance, with respect to the NSP/Java naming convention, a class denotes a Java Bean, object attributes and associations denote Java Bean properties. In the diagrams all associations are compositions in order to emphasize, that all object nets gathered by the user are trees[8].

A first introductory example is given by the form message type Person in Figure 3.1 and the form in listing 3.7. The example is already sufficient in order to state the NSP parameter guidelines for form message types. The form targets a web signature that consists of a formal parameter customer of type Person. Within the form data for an object net consisting of a Person object and an Address object may be entered; on submit the respective object net is constructed[9]. A formal parameter of form message type must be targeted by at most one object element. Like the control tags, the opening object tag has a param attribute for this purpose. The object element contains controls and possibly further object elements for the attributes of the targeted parameter. Thereby the NSP parameter guidelines apply recursively, i.e. from a form's viewpoint a form message type can be understood as a nested formal parameter type.

The form in listing 3.8 targets a web signature that consists of a formal array parameter customers, the array items' type is again the type Person given in Figure 3.1. A formal array parameter of form message type may be targeted by an arbitrary number of object elements[10], as long as the NSP parameter guidelines are fulfilled recursively for each object element. The single object nets may be explicitly indexed. An object element may contain an index element for

[8]Following [156] aggregations may form cycles, but constrain the respective link relationship to be transitive and antisymmetric. Compositions are aggregations with an additional constraint: a part may only be part of one composite.

[9]More concrete, on submit a Person object and an Address object are created, thereby a pointer to the Address object is created which automatically becomes the address attribute of the Person object.

[10]Figure 3.8, line 3, line 9

Listing 3.7

```
01 <form callee="target">
02   <object param="customer">
03     <input type="String" param="name"></input>
04     <input type="int"    param="age"></input>
05     <object param="adress">
06       <input type="String" param="street"></input>
07       <input type="int"    param="zip"></input>
08     </object>
09   </object>
10 </form>
11
12 // web signature of target
13 <param name="customer" type="Person"></param>
```

this purpose[11].

An object element may be given a uniquely identifying name with the optional id attribute[12]. Then controls and object elements for gathering data for form message type attributes can occur anywhere. They have to reference the object element they belong to by its identifying name. Opening control[13] and object tags[14] have an optional in-attribute for this purpose. Identifying and referencing object elements provides a convenient way for allowing arbitrary form layout. With object element nesting only, some desired occurrences of controls that contradict the rigid structure of layout elements could not be realized[15]. Listing 3.8 yields an example. Data for two object nets are gathered. Data for the attributes of a each object net are gathered in one table column at each case. Thereby the referencing mechanism just introduced is needed[16].

A formal parameter of form message type is optional. It is possible not to provide an object element for it. On submit the system will pass a null object to the receiving dialogue method. Again it becomes important, that the notion of form message type attribute is subsumed under the notion of formal parameter. For example the first object element in line 3 of listing 3.8 possesses an object element for the Address attribute in line 7, whereas the second object element in line 9 does not posses an object element for this attribute. Most importantly, though a formal parameter of form message type is optional, this is not carried over to its contained attributes. An object element may not be provided, but if it is provided, the NSP parameter guidelines apply to the contained attributes,

[11] Figure 3.8, line 4, line 10

[12] Listing 3.8, line 3, line 9

[13] Listing 3.8, e.g. line 18

[14] Listing 3.8, line 7

[15] The XForms technology even introduces decoupling of controls from forms [69] for similar reasons. The respective attributes are the id-attribute and the ref-attribute.

[16] A solution based on nested tables cannot achieve the same layout effect. If the data cells of the given minimal example contain more information, so that data cells of a row have different heights, correct alignment is not ensured any more.

Listing 3.8

```
01 <form callee="target">
02   <table>
03     <tr><td><object param="customers" id="first">
04               <index>1</index>
05               <input type="String" param="name"></input>
06             </object>
07             <object param="adress" id="firstPart" in="first"></object>
08         </td>
09         <td><object param="customers" id="second">
10               <index>2</index>
11               <input type="String" param="name"></input>
12             </object>
13         </td>
14     </tr>
15     <tr><td><input type="int" param="age" in="first"></input></td>
16         <td><input type="int" param="age" in="second"></input></td>
17     </tr>
18     <tr><td><input type="String" param="street" in="firstPart"></input>
19         </td>
20         <td></td>
21     </tr>
22     <tr><td><input type="int" param="zip" in="firstPart"></input>
23         </td>
24         <td></td>
25     </tr>
26   </table>
27 </form>
28
29 // web signature of target
30 <param name="customers" type="Person[]"></param>
```

for example an int attribute would be required.

For formal parameters data entry may be specified as required with the data-attribute[17]. For a parameter of form message type this enforces that exactly one object element for it is provided.

The fact that formal parameters of form message type are optional enables the support of cyclic user defined data in forms. The form in listing 3.9 targets a web signature that consists of a formal parameter of type ParticipantList. The type ParticipantList, which is a dynamic data structure, is given in Figure 3.2. In the example form input capabilities for three list elements are given. On submit an object net of three list elements is created. For the next-pointer of the third element a null object is filled in. Dynamic creation of input capabilities for dynamic data structures in a statically type-safe manner is possible by recursive

[17] Listing 3.10, line 2

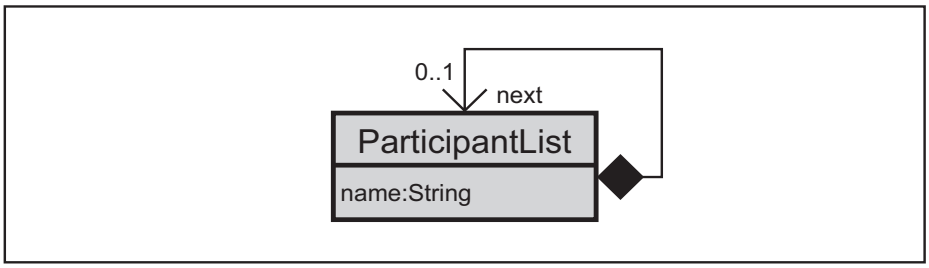

Figure 3.2: Example Cyclic Form Message Type.

definitions of dialogue submethods, which are introduced in section 4.2.

Listing 3.9

```
01 <form callee="target">
02   <object param="participant">
03     <input type="String" param="name"></input>
04     <object param="next">
05       <input type="String" param="name"></input>
06       <object param="next">
07         <input type="String" param="name"></input>
08         <object param="next">
09           <input type="String" param="name"></input>
10         </object>
11       </object>
12     </object>
13   </object>
14 </form>

// web signature of target
<param name="participants" type="ParticipantList"></param>
```

NSP defines a fine granular mechanism for putting constraints on the attributes of user defined data. For this purpose a parameter-element that defines a formal parameter of form message type may contain constraints-elements. Each of the constraints-elements must uniquely refer to a form message type that is involved as a part in the definition of the formal parameter type. A constraints-element contains a param-element for every attribute of the form message type it refers to. The param-element can be used to pose on the attributes one or several of the constraints that have been introduced in section 3.2[18][19].

Listing 3.10 gives examples for constraints on user defined data attributes.

[18] data=required, widget=required, widget=password, widget=textarea

[19] The mechanism is not recursive. The param-elements of a constraint specification must not contain further constraints-elements.

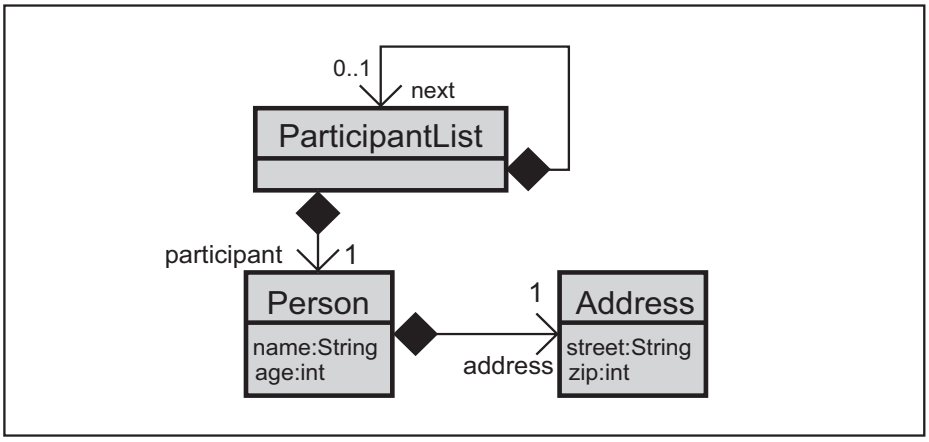

Figure 3.3: Example Complex Form Message Type.

Consider the form message type given in Figure 3.3. Again it is ParticipantList - like in Figure 3.2 - but this time the type's attribute has a form message type. The attribute's type is Person, which already served as an example before (Figure 3.1), but this time the cardinality of the address-association has changed from 0..1 to 1, which depicts that the address attribute is required this time. In listing 3.10 a formal parameter of type ParticipantList is defined. The constraint specification in line 3 to line 6 causes, that the participant-attribute is required. That is, whenever an object element for gathering data for a list element for the actual parameter is generated, it must contain an object element for its participant-attribute. There is no additional constraint posed on the next-attribute. The next-attribute must be optional, because it forms a cycle in the class diagram. Further constraint specifications ensure for example, that for every Person object, the user has to enter a name, a street and a zip code. Note that it would not be sufficient to constrain the street-attribute and zip-attribute of the type Address to be required in order to achieve this[20]. The address-attribute of the type Person must be required for this purpose, too[21]. Without requiring the address-attribute, the whole address is optional. Requiring street and zip code only, just enforces that these are required in the case that an object element for the address is actually generated.

3.6 Document Structure Guidelines and Rules

The NSP coding guidelines for ensuring client page type safety has been described in sections 3.2 to 3.5, this section completes the discussion by describing remaining coding guidelines and rules concerning client page description safety.

[20] Listing 3.10, line 13, line 14
[21] Listing 3.10, line 10

Listing 3.10

```
01 // web signature of target
02 <param name="participants" type="ParticipantList" data="required">
03  <constraints type="ParticipantList">
04    <param name="participant" data="required"></param>
05    <param name="next"></param>
06  </constraints>
07  <constraints type="Person">
08    <param name="name" data="required"></param>
09    <param name="age"></param>
10    <param name="address" data="required"></param>
11  </constraints>
12  <constraints type="Address">
13    <param name="street" data="required"></param>
14    <param name="zip" data="required"></param>
15  </constraints>
16 </param>
```

The first basic structure guideline demands that an NSP server page always only generates well-formed XML. The corresponding coding rule for NSP code states the following: the NSP document must be a well-formed XML document and the Java block structure must be compatible with the XML block structure. The latter means that for every XML tag the corresponding dual tag must occur in the same block, whereas as a necessary exception to this the Java tags are ignored. Java tags must be considered merely as switches between the programming language and the markup language with respect to the document structure[22] [23].

The second structure guideline demands that an NSP server page always only generates valid passive NSP/Java code. That means that an active NSP document fragment must only generate valid content elements with respect to its directly encompassing tags. Passive NSP code is essentially XHTML up to the following differences:

- some element properties are supported by content elements instead of attributes,
- some elements, e.g. the form element and the control elements, are modified,
- some new elements are introduced, e.g. the param element and the call element,

[22] In technical terms java opening and closing tags are go-betweens of different lexical states.

[23] Alternatively the given coding rule can be reformulated more concisely in the following way: The document must be well-formed XML. Then Java tags are ignored, instead every Java block is considered a new document element. The document in quest must be well-formed with respect to the resulting element set.

- some content elements are not required, though their counterparts are required in XHTML.

The first three differences pose no problems. The introduced modifications are resolved when the final page, i.e. the XHTML page that is sent to the browser, is generated. The fourth difference is discussed in the sequel, thereby the coding guideline must be discussed with respect to the finally called page.

In XHMTL [172] some elements require that a certain kind of content element appears at least once. In NSP these demands are dropped for pragmatic reasons, not for technical reasons. For example in XHTML a list must contain at least one item. A table must contain at least one row. A table row must contain at least one table data cell. A select list must contain at least either one option or one group of option. However most of these constraints are artificial[24], because browsers can, and current browsers do, cope with violations of these constraints in a natural sensible way. For example, lists without items, tables without rows and table rows without data cells are just not displayed. In NSP it would be possible to define coding rules that ensure validity with respect to the aforementioned constraints, but this would lead to unjustifiably complex demands on the NSP code. For example, list items are typically provided by a loop and in such a case the developer would have to add for example an initial list item by sequencing. Therefore the NSP concept of client page description safety is weaker than full XHTML validity[25].

The coding rule for the second structure guideline is given, though a bit verbose, with respect to sequencing, switch structures, and loops again. The demands are irrespective of the parameter guidelines and rules given so far. Pure Java code is considered neutral. In a sequence of document parts, all document parts must be neutral or generate valid elements. In a loop the body must be neutral or generate valid elements. In a switch structure all branches must be neutral or generate valid elements.

[24] Moreover some of these constraints are ineffectual. For example though an item is required for a list, it is not required that a list must not be empty. As another example, an interesting constraint for table rows is not that a row contains at least one data cell, but that all rows of a table contain the same number of data cells up to grouping with the column-span mechanism.

[25] Note that NSP client page description safety is partly stronger than XHTML validity as well, e.g. at least two radio buttons are needed in order to form a valid composed radio button control in the NSP approach.