

Python SOAP Tutorial

Erstellung eines WEB- Service mit ZSI

Richard Mutschler
Email: Mutschler.Richard@web.de

10. Februar 2007

Inhaltsverzeichnis

1	Vorwort	3
1.1	Lizens	3
2	Einführung	3
3	Einfache Datentypen: Ein rpc/literal MatheService	4
3.1	Die MatheService WSDL	4
3.2	Der Python ZSI Server für den MatheService	5
3.2.1	Methodenstubs aus der WSDL generieren	5
3.2.2	Implementation des MatheService Web- Servers	6
3.2.3	Die Service Implementation einfügen	7
3.3	Der Python ZSI Client für den MatheService	10
4	Komplexe Datentypen	12
4.1	Erweitern der WSDL mit Eclipse	12
4.2	Den Erweiterten Server erstellen	16
4.3	Der erweiterte Client	20
5	Deutsche Zusammenfassung des ZSI user manual	22
5.1	Übersicht	22
5.1.1	SOAP Bindings	22
5.1.2	Python Werkzeuge	22
5.2	Von der WSDL zum Python Code	22
5.2.1	wsdl2py - Die Grundlagen der Codeerstellung	23
5.2.2	Typecode Erweiterungen	24
6	Links	25

Abbildungsverzeichnis

1	Workspace in Eclipse	12
2	Design Ansicht der MatheService.wsdl	13
3	Design Ansicht mit neuer Operation	13
4	Design Ansicht mit neuer Operation und zugewiesenen Elementen	13
5	Element ModuloRequest	14
6	Komplexer Datentyp	14

1 Vorwort

Obwohl Web- Services eine immer größere Verbreitung finden, ist die Erstellung von Clients und Servern bei Weitem nicht so einfach wie an vielen Stellen versprochen wird. Dies liegt wohl nicht zuletzt daran, dass kaum einfache Einführungen in das Thema existieren. Gerade deutschsprachige Anleitungen sind äußerst rar. Dieses Tutorial soll den Schritt für Schritt Einstieg in das Thema Web- Services für Anfänger (wie mich) erleichtern. Zur Implementation wird Python verwendet. Besonderes Augenmerk wird hierbei auf die Verwendung der SOAP Bibliothek ZSI gelegt. Eine Einführung in die C/C++ Bibliothek gSOAP ist im Anschluss an die Fertigstellung dieses Tutorials geplant. Als Grundlage für dieses Tutorial dienten mir die folgenden Quellen:

- Interoperable WSDL/SOAP web services introduction: Python ZSI, Excel XP, gSOAP C/C++ and Applix SS, von Holger Joukl, LBBW Financial Markets Technologies.
- The Zolera Soap Infrastructure User's Guide von Joshua Boverhof.

In der Sektion Links findet ihr Links zu allen benötigten Downloads.

1.1 Lizens

Copyright c 2007 Richard Mutschler. All rights reserved.

Sources copied from other authors are labeled with a copyright note.

Redistribution and use in source (LYX, LATEX) and 'compiled' forms (SGML, HTML, PDF, PostScript, RTF and so forth) with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code (LYX, LATEX) must retain the above copyright notice, this list of conditions and the following disclaimer as the first lines of this file unmodified.
2. Redistributions in compiled form (transformed to other DTDs, converted to PDF, PostScript, RTF and other formats) must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS DOCUMENTATION IS PROVIDED 'AS IS' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2 Einführung

Es wird in diesem Tutorial von einer gewissen Grundkenntnis in Python ausgegangen. Einige grundlegende Konzepte von WSDL, SOAP und HTTP Servern werden hier beiläufig vorgestellt. Es wird hier nicht näher darauf eingegangen, da im Netz eine Menge Informationen darüber verfügbar sind. Mit diesen Softwareversionen wurde bei der Erstellung dieses Tutorials gearbeitet:

- Python 2.4
- PyXML 0.8.3

- ZSI 2.0 rc3
- soapUI 1.6
- Eclipse 3.2
 - PyDev für Eclipse 1.2.5
 - Web Standard Tools (WST) für Eclipse 1.5.2

Der Ausgangspunkt für das hier gezeigte Beispiel wird eine WSDL Datei sein. Eine große Sammlung an WSDL Dateien für weitere Projekte findet ihr unter: <http://www.xmethods.com>. Um eine möglichst große Interoperabilität zu gewährleisten, werden hier nur Web- Service 1 compatible WSDL Dateien im rpc/literal und document/literal Style verwendet.

3 Einfache Datentypen: Ein rpc/literal MatheService

Hier wird ein einfaches Beispiel implementiert, das anfangs nur eine Funktion anbietet, die ein double als Argument entgegennimmt und das Quadrat der Zahl zurückgibt. Später wird dieses Beispiel um eine Funktionen für die Modolodivision erweitert. Hier werden anfangs nur einfache skalare Datentypen verwendet, ein einzelnes Argument und ein Rückgabewert.

3.1 Die MatheService WSDL

Diese WSDL bildet das Interface zum MatheService ¹

MatheService WSDL

```
<?xml version="1.0"?>
<definitions name="MatheService"
  targetNamespace="http://MyNs:8080/MatheService"
  xmlns:tns="http://MyNs:8080/MatheService"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <message name="getSquareRequest">
    <part name="x" type="xsd:double"/>
  </message>
  <message name="getSquareResponse">
    <part name="return" type="xsd:double"/>
  </message>
  <portType name="SquarePortType">
    <operation name="getSquare">
      <documentation> the square method
      </documentation>
      <input message="tns:getSquareRequest"/>
      <output message="tns:getSquareResponse"/>
    </operation>
  </portType>
  <binding name="SquareBinding" type="tns:SquarePortType">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/
http"/>
    <operation name="getSquare">
      <soap:operation soapAction="http://127.0.0.1:8080/MatheService/
getSquare"/>
      <input>
        <soap:body use="literal" namespace="http://MyNS:8080/Mathe-
Service"/>

```

¹Original by Holger Joukl, modified by Richard Mutschler

```

    </input>
    <output>
      <soap:body use="literal"
        namespace="http://MyNS:8080/MatheService" />
    </output>
  </operation>
</binding>
<service name="SquareService">
  <documentation>Returns  $x^2$  ( $x**2$ , square(x)) for a given float x
  </documentation>
  <port name="SquarePort" binding="tns:SquareBinding">
    <soap:address location="http://127.0.0.1:8080/MatheService"/>
  </port>
</service>
</definitions>

```

3.2 Der Python ZSI Server für den MatheService

Das ZSI Web Service Paket ist ein Tool zur top- down Entwicklung von Web- Services. Dies bedeutet, dass eine vorhandene WSDL verwendet wird, um Client und Serveranwendungen zu erstellen. Im Kontext dieses Dokuments bezeichnet ein Webservice eine WSDL Datei, die das Service Interface beschreibt.

3.2.1 Methodenstubs aus der WSDL generieren

ZSI stellt zwei Python Skripte zur Verfügung, mit denen Quellcodegerüste für Server und Client aus der WSDL generiert werden können:

- **wsdl2py** wird verwendet, um die Python Bindings für den Service zu generieren.
- **wsdl2dispatch** erstellt ein Servergerüst, um den Service auszuführen. Die Verarbeitungsmethoden, bei uns also SquareService, werden hier eingebracht.

Wenn ZSI installiert ist, öffnet man nun am besten eine Konsole und wechselt in das Projektverzeichnis, in dem auch unsere MatheService.wsdl liegt. Nun werden die beiden Skripte ausgeführt:

- `wsdl2py -f MatheService.wsdl`
- `wsdl2dispatch -f MatheService.wsdl`

Die Option `-f` spezifiziert die Eingabedatei, also unsere WSDL Datei. Wenn eine WSDL aus dem Netz verwendet werden soll, beispielsweise von `http://www.xmethods.com`, kann hier durch `-u` statt `-f` auch die URL angegeben werden. Es sollten sich nun im Verzeichnis drei neue Dateien befinden:

- **MatheService_services.py** - Die durch `wsdl2py` generierten Bindings,
- **MatheService_services_types.py** - Durch `wsdl2py` generierten Typdefinitionen,
- **MatheService_services_server.py** - Das durch `wsdl2dispatch` generiertes Server Gerüst.

Was ist nun zu tun? Wir haben das Server Skelett und die Python Bindings um mit dem Service zu Kommunizieren. Was wir nun brauchen ist:

- Das Hauptprogramm, das den (HTTP -) Server und die Request Handler ausführt und
- die Methoden, die die Anfragen bearbeiten.

ZSI beinhaltet das Modul `ZSI.ServiceContainer`, welches den Server für uns implementiert.


```

xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body xmlns:ns1="http://MyNS:8080/MathService">
    <ns1:getSquareResponse>
      <return xsi:nil="1"/>
    </ns1:getSquareResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Da wir im Server ja noch keine Funktion geschrieben haben, die diese Anfrage bearbeitet, erhalten wir als Ergebnis : `return xsi:nil=1`. Das war soweit ja auch zu erwarten. Wir sehen vor allem aber, dass der Server läuft und unsere Anfrage entgegennimmt. Um nun sinnvolle Ergebnisse zu erhalten implementieren wir die Funktion `getSquare`.

3.2.3 Die Service Implementation einfügen

Das Einzige was wir nun tun müssen, ist die Implementation des Services in das durch `wsdl2dispatch` erzeugte Servergerüst einzufügen.

Hierzu sind folgende Schritte notwendig:

- Weiterleitung der Anfrage an die richtige Methode,
- der Methode die Parameter aus der SOAP- Anfrage übergeben,
- das Ergebnis der Methode in die SOAP- Antwort schreiben.

Wir öffnen zuerst die Datei `MatheService_services_server.py`:

`MatheService_services_server.py`

```

#####
# MatheService_services_server.py
#   Generated by ZSI.generate.wsdl2dispatch.ServiceModuleWriter
#
#####

from MatheService_services import *
from ZSI.ServiceContainer import ServiceSOAPBinding

class SquareService(ServiceSOAPBinding):
    soapAction = {}
    root = {}
    _wsdl = #Hier steht im Sourcecode nocheinmal die WSDL.
    #Ich habe sie zur besseren Lesbarkeit aus dem Listing entfernt

    def __init__(self, post='/MathService', **kw):
        ServiceSOAPBinding.__init__(self, post)

    def soap_getSquare(self, ps):
        self.request = ps.Parse(getSquareRequest)
        response=getSquareResponse
        response._return=self.getSquare(self.request._x)
        return response

    soapAction['http://127.0.0.1:8080/MathService/getSquare'] =
'soap_getSquare'
    root[(getSquareRequest.typecode.namespace, getSquareRequest.typecode.pname)]
='soap_getSquare'

```

Wir fügen nun die Funktion `getSquare` hinzu, die ausgeführt werden soll um das Quadrat einer Zahl zu berechnen:

`getSquare(..)`

```
def getSquare(self, x):
    return x**2
```

Nun verändern wir die Funktion `soap_getSquare(self, ps)`:

`soap_getSquare(self, ps)`

```
1 def soap_getSquare(self, ps):
2     args = ps.Parse(getSquareRequest)
3     response=getSquareResponse()
4     response._return=self.getSquare(args._x)
5     return response
```

In Zeile 2 `args = ps.Parse(getSquareRequest)` weisen wir der Variable `args` ein Array aus dem Request zu. Die Klasse `getSquareRequest` ist in der Datei `MatheService_services.py` definiert:

`MatheService_services.py`

```
.
.
.
class getSquareRequest:
    def __init__(self):
        self._x = None
    return
.
.
.
class getSquareResponse:
    def __init__(self):
        self._return = None
    return
.
.
.
```

In Zeile 3 `response=getSquareResponse()` weisen wir der Variable `Response` den Typ `getSquareResponse()` zu.

In Zeile 4 wird nun unsere `getSquare` Methode aufgerufen. Als Parameter wird ihr der `_x` Wert der Anfrage (das `'_'` ist wichtig!!!) übergeben. Das Ergebnis weisen wir dann der Variable `_return` unserer `Response` Variable zu.

Am Ende geben wir die mit einem Wert gefüllte `Response` Variable zurück.

Wir testen unseren Server nun wieder mit SOAPUI:

SOAPUI Request

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:mat="http://MyNS:8080/MatheService">
  <soapenv:Body>
    <mat:getSquare>
      <x>6</x>
    </mat:getSquare>
  </soapenv:Body>
</soapenv:Envelope>
```

SOAPUI Response

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body xmlns:ns1="http://MyNS:8080/MatheService">
    <ns1:getSquareResponse>
      <return>36.000000</return>
    </ns1:getSquareResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Herzlichen Glückwunsch, unser Server funktioniert!

3.3 Der Python ZSI Client für den MatheService

Um den Client zu implementieren, legen wir ein neues Python File an, in diesem Beispiel *MyMathServiceClient.py*:

MyMathServiceClient.py

```
1 from MatheService_services import *
2 import sys
3 from optparse import OptionParser
4 #####
5 #Parsen der Optionen #
6 #####
7 parser = OptionParser()
8
9 parser.add_option("-s", "--square", dest="square", type="string",
10 help="Quadrat_einer_Zahl_errechnen")
11
12 (options, args) = parser.parse_args()
13
14 loc = SquareServiceLocator()
15 proxy = loc.getSquarePortType(tracefile = sys.stdout)
16
17 #####
18 #Diese Funktion wird ausgeführt, #
19 #wenn die -s Option gesetzt ist #
20 #####
21 if options.square:
22     request = getSquareRequest()
23     request._x=(float)(options.square)
24     response= proxy.getSquare(request)
25     print "Das_Quadrat_von_", request._x,"_ist_" , response._return
```

Man erkennt schnell, dass der Client recht simpel ist. Die relevanten Codezeilen sollen nun erläutert werden:

- Zeile 1: Die von wsdl2py erstellten Bindings werden importiert.
- Zeile 14: Eine Instanz der Locator Klasse wird erzeugt. Die Locator Klasse ist in *MatheService_services.py* definiert. Der Locator repräsentiert die Bindings zu dem angegebenen Web- Service und die Port Klasse, die verwendet wird, um die Remote- Operationen des Web- Services einzubinden. Weiterhin werden hier verschiedene Message Klassen definiert, die die SOAP und XML-Schema Datentypen abbilden. **Der Name dieser Klasse hängt natürlich vom Namen des in der WSDL definierten Services ab. Das heißt, dass wenn eine andere WSDL verwendet wird, sich auch der Name ändert. Die Klasse findet man in der von wsdl2py generierten _services.py Datei recht einfach (Es ist die erste Klasse, die ihr in der Datei findet und endet immer auf *ServiceLocator*).**
- Zeile 15: Der Variablen proxy wird der Port der Locator- Klasse zugewiesen. Über diesen Port werden später die Operationen des Web- Services angesprochen. Der übergebene Parameter *tracefile = sys.stdout* gibt zum Debugging die SOAP Nachrichten auf der Standardausgabe aus.
- Zeile 22: Wie auch beim Server benötigen wir die Instanz der Klasse, die den Datentyp aus der SOAP Message repräsentiert.
- Zeile 23: Analog zum Server füllen wir unsere Anfrage mit einem Wert. *Hinweis: In Eclipse werden die möglichen Variablen, die in der Anfrage gesetzt werden können, durch die Codevervollständigung (Strg+ Space) angezeigt.*

- Zeile 24: Hier wird nun die Operation getSquare aufgerufen und das Ergebnis der Variable response zugewiesen.

Das folgende Listing zeigt die Konsolenausgabe unseres Clients:

Ausgabe des Clients

```
python MyMatheServiceClient.py -s 3.3
----- Tue Feb 6 16:48:44 2007 REQUEST:
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body xmlns:ns1="http://MyNS:8080/MatheService">
    <ns1:getSquare>
      <x>3.300000</x>
    </ns1:getSquare>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
----- Tue Feb 6 16:48:45 2007 RESPONSE:
200
OK

Server: ZSI/1.1 BaseHTTP/0.3 Python/2.4.4
Date: Tue, 06 Feb 2007 15:48:45 GMT
Content-type: text/xml; charset="utf-8"
Content-Length: 484

<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body xmlns:ns1="http://MyNS:8080/MatheService">
    <ns1:getSquareResponse>
      <return>10.890000</return>
    </ns1:getSquareResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
Das Quadrat von 3.3 ist 10.89
```

Herzlichen Glückwunsch, unser Client funktioniert auch!

4 Komplexe Datentypen

Im ersten Teil dieser Anleitung haben wir einen Service mit einfachen Datentypen erstellt. Dieses Beispiel soll nun um eine Operation mit komplexen Datentypen erweitert werden. Hierzu implementieren wir eine Operation, die eine Modulodivision zweier Zahlen vornimmt. Als Parameter übergeben wir der Operation zwei Zahlen vom Typ Integer. Um mehrere Werte in einem Request übergeben zu können, benötigen wir einen Datentyp, der folgendermaßen aussieht:

Komplexer Datentyp für den ModuleRequest

```
<xsd:complexType name="ModuloRequest">
  <xsd:sequence>
    <xsd:element name="Zahl1" type="xsd:int"></xsd:element>
    <xsd:element name="Zahl2" type="xsd:int"></xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Um die neue Operation hinzuzufügen, werden wir zuerst unsere WSDL Datei anpassen. Anschließend können wir uns die Bindings und das Server Skelett wieder durch die Skripte *wsdl2py* und *wsdl2dispatch* generieren lassen.

4.1 Erweitern der WSDL mit Eclipse

Zuerst sollte in Eclipse ein neuer Ordner im Projekt angelegt und die original WSDL importiert werden. Im ersten Teil dieser Anleitung haben wir eine rpc/literal encoded WSDL verwendet. Nun werden wir document/literal encoding verwenden.

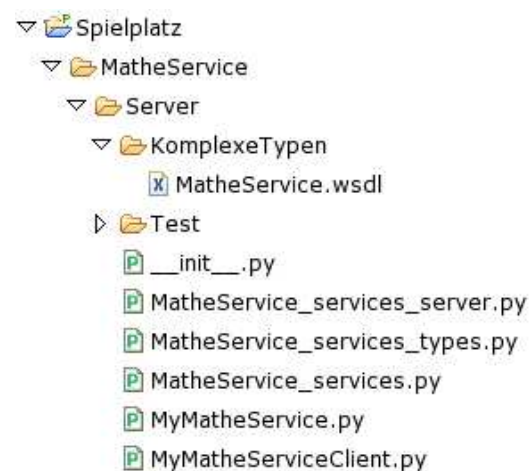


Abbildung 1: Workespace in Eclipse

Durch die Web Standard Tools können wir uns die WSDL in einer grafischen Darstellung anzeigen lassen. Das Erste was wir ändern, ist die Benennung. Da der Service im ersten Teil nur die

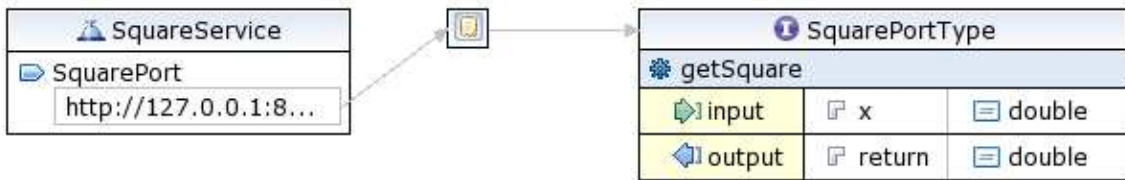


Abbildung 2: Design Ansicht der MatheService.wsdl

Operation zum Berechnen des Quadrates bot, machten Namen wie 'SquareService', 'SquarePort' etc. noch Sinn. Nun ändern wir die Namen durch doppelklicken in 'MatheService', 'MathePort'... . Um mit den ZSI Skripten wsdl2py und wsdl2dispatch Python Bindings für komplexe Datentypen zu erstellen, ist es nötig, dass den ein- und ausgehenden Nachrichten einer Operation ein Element und kein Typ zugewiesen wird. Ein Element wird zugewiesen, indem mit einem Rechtsklick auf die Nachricht das Kontextmenü geöffnet wird. Dort wählen wir die Aktion *Set Element -New Element*. Wir vergeben einen sinnvollen Namen, beispielsweise 'SquareRequest'. Dies führen wir für die ein- und die ausgehende Nachricht durch. Durch einen Rechtsklick auf den PortType erhalten wir ein Kontextmenü, in dem wir *Add Operation* auswählen. Wir sollten nun folgende Ansicht erhalten:

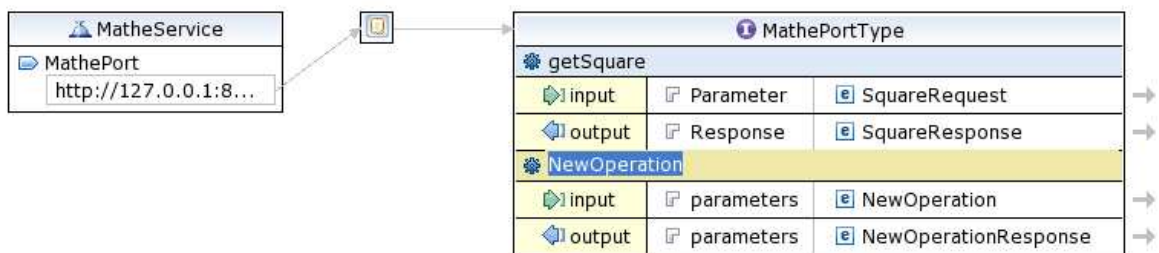


Abbildung 3: Design Ansicht mit neuer Operation

Zuerst ändern wir die vordefinierten Namen der Operation, der input- und output Messages und weisen ihnen neue Elemente zu. Das Ergebnis sollte folgendermaßen aussehen:

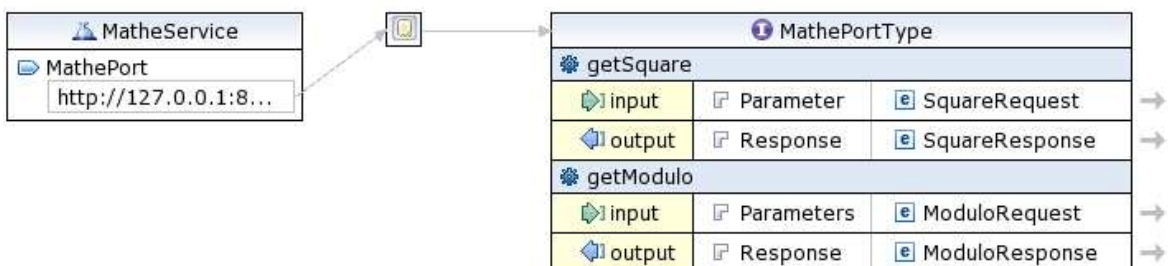


Abbildung 4: Design Ansicht mit neuer Operation und zugewiesenen Elementen

Durch Klicken auf den Pfeil hinter dem Messages öffnet Eclipse die Inline Schema Ansicht der WSDL und wir sehen unseren Typ *ModuloRequest*:



Abbildung 5: Element ModuloRequest

Über das Kontextmenü führen wir die folgenden Aktionen aus:

- Set Type ->New...
- *Complex Type* und *Create as local anonymous type* anwählen
- mit OK bestätigen

Dem jetzt noch leeren Typ fügen wir durch die Aktionen *Add Sequence* und 2 * *Add Element* zwei neue Elemente hinzu, ändern die Namen und den Typ der Elemente (auf Wunsch kann hier auch die Multiplizität angegeben werden). Folgende Abbildung zeigt das Ergebnis:

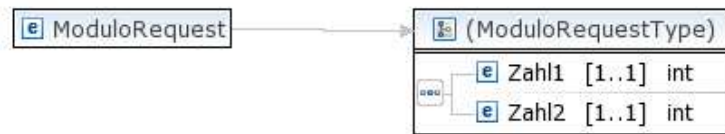


Abbildung 6: Komplexer Datentyp

Wir wechseln nun in Eclipse wieder zur WSDL Ansicht. Im Kontextmenü des Bindings (kleines Rechteck in der Verbindung zwischen Port und PortTyp) wählen wir die Aktion *Generate Binding Content...*. Im nun erscheinenden Binding Wizzard ändern wir noch den Namen von 'SquareBinding' zu 'MatheBinding', selektieren *Overwrite existing binding information* und wählen *document literal* als SOAP Binding Option aus. Wir können uns die WSDL nun durch wechseln des Reiters von 'Design' auf 'Source' ansehen:

Erweiterte WSDL

```
<?xml version="1.0"?>
<definitions name="MatheService"
targetNamespace="http://MyNs:8080/MatheService"
xmlns:tns="http://MyNs:8080/MatheService"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://MyNs:8080/MatheService">
      <xsd:element name="SquareRequest" type="tns:RequestTyp">
        </xsd:element>
      <xsd:element name="SquareResponse" type="tns:ResponseTyp">
        </xsd:element>
      <xsd:complexType name="RequestTyp">
        <xsd:sequence>
          <xsd:element name="x" type="xsd:double"
maxOccurs="1" minOccurs="1">
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      <xsd:complexType name="ResponseTyp">
        <xsd:sequence>
          <xsd:element name="result" type="xsd:double"
maxOccurs="1" minOccurs="1">
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      <xsd:complexType name="SquareRequest"></xsd:complexType>
      <xsd:element name="getModulo">

```

```

    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="in" type="xsd:string">
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="getModuloResponse">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="out" type="xsd:string">
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="ModuloRequest">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Zahl1"
          type="xsd:int" maxOccurs="1" minOccurs="1">
        </xsd:element>
        <xsd:element name="Zahl2" type="xsd:int"
          maxOccurs="1" minOccurs="1">
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="ModuloResponse">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="result" type="xsd:int"
          maxOccurs="1" minOccurs="1">
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
</types>
<message name="getSquareRequest">
  <part name="Parameter" element="tns:SquareRequest"/>
</message>
<message name="getSquareResponse">
  <part name="Response" element="tns:SquareResponse"/>
</message>
<message name="getModuloRequest">
  <part name="Parameters" element="tns:ModuloRequest"></part>
</message>
<message name="getModuloResponse">
  <part name="Response" element="tns:ModuloResponse"></part>
</message>
<portType name="MathePortType">
  <operation name="getSquare">
    <documentation> the square method </documentation>
    <input message="tns:getSquareRequest"/>
    <output message="tns:getSquareResponse"/>
  </operation>
  <operation name="getModulo">
    <input message="tns:getModuloRequest"></input>
    <output message="tns:getModuloResponse"></output>
  </operation>
</portType>
<binding name="MatheBinding" type="tns:MathePortType">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <operation name="getSquare">
    <soap:operation
      soapAction="http://MyNs:8080/MatheService/getSquare" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
  <operation name="getModulo">
    <soap:operation
      soapAction="http://MyNs:8080/MatheService/getModulo" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
<service name="MatheService">
  <documentation>Returns  $x^2$  ( $x**2$ , square(x)) for a given float x
  </documentation>
  <port name="MathePort" binding="tns:MatheBinding">
    <soap:address location="http://127.0.0.1:8080/MatheService"/>
  </port>
</service>
</definitions>

```

Um die WSDL mit SoapUI zu testen, exportieren wir sie in unser Projektverzeichnis und laden sie in SoapUI. Die durch SoapUI generierten Requests sollten folgendermaßen aussehen:

getSquare Request

```
<soapenv:Envelope
xmlns:soapenv=" http://schemas.xmlsoap.org/soap/envelope/"
xmlns:mat=" http://MyNS:8080/MatheService">
  <soapenv:Header/>
  <soapenv:Body>
    <mat:SquareRequest>
      <x>?</x>
    </mat:SquareRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

Modulo

```
<soapenv:Envelope
xmlns:soapenv=" http://schemas.xmlsoap.org/soap/envelope/"
xmlns:mat=" http://MyNS:8080/MatheService">
  <soapenv:Header/>
  <soapenv:Body>
    <mat:ModuloRequest>
      <Zahl1>?</Zahl1>
      <Zahl2>?</Zahl2>
    </mat:ModuloRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

4.2 Den Erweiterten Server erstellen

Wenn wir nun durch die Skripte *wSDL2py* und *wSDL2dispatch* die neuen Python Bindings und das Serverskelett erzeugen wollen, müssen wir beim Aufruf der Skripte die Option *-e* für extended Code generation und bei *wSDL2py* zusätzlich *-b* setzen, sonst werden die komplexen Datentypen nicht richtig abgebildet. Die Aufrufe lauten nun also:

- `wSDL2py -e -b -f MatheService.wsdl`
- `wSDL2dispatch -e -f MatheService.wsdl`

Wenn ihr hier eine Fehlermeldung erhaltet, empfiehlt es sich zu prüfen, ob den input und output Messages der Operationen wirklich Elemente und keine Typen zugewiesen wurde und ob der encoding Style auf document/literal gesetzt ist. Nun importieren wir die erstellten Dateien in Eclipse. Da das Hauptprogramm (*MyMatheService.py*) aus dem ersten Teil das durch *wSDL2dispatch* generierte Server Skelett verwendet, können wir es fast identisch übernehmen. Nur die Zeilen:

```
from MatheService_services_server import SquareService
```

der Importanweisung

```
from MatheService_services_server import MatheService
```

```
AsServer(port=8080, services=[SquareService()], RequestHandlerClass=MySOAPRequestHandler)
```

des Requesthandlers

```
AsServer(port=8080, services=[MatheService()], RequestHandlerClass=MySOAPRequestHandler)
```

sind umzubenennen, da wir vorhin den Namen unseres Services geändert haben.

Wenn wir die Datei *MatheService_services_types.py* öffnen, erkennen wir hier die Klasse, die unseren komplexen Datentyp abbildet:

MatheService_services_types.py

```
#####
# MatheService_services_types.py
# generated by ZSI.generate.wSDL2python
#####
```

```

import ZSI
import ZSI.TCcompound
from ZSI.schema import LocalElementDeclaration,
ElementDeclaration, TypeDefinition, GTD, GED

#####
# targetNamespace
# http://MyNs:8080/MatheService
#####

class ns0:
    targetNamespace = "http://MyNs:8080/MatheService"

    class RequestTyp_Def(ZSI.TCcompound.ComplexType, TypeDefinition):
        schema = "http://MyNs:8080/MatheService"
        type = (schema, "RequestTyp")
        def __init__(self, pname, ofwhat=(), attributes=None,
        extend=False, restrict=False, **kw):
            ns = ns0.RequestTyp_Def.schema
            TClist = [ZSI.TCnumbers.FPdouble(pname="x", aname="_x",
            minOccurs=1, maxOccurs=1, nillable=False, typed=False,
            encoded=kw.get("encoded"))]
            self.attribute_typecode_dict = attributes or {}
            if extend: TClist += ofwhat
            if restrict: TClist = ofwhat
            ZSI.TCcompound.ComplexType.__init__(self, None, TClist,
            pname=pname, inOrder=0, **kw)
            class Holder:
                typecode = self
                def __init__(self):
                    # pyclass
                    self._x = None
                return
            Holder.__name__ = "RequestTyp_Holder"
            self.pyclass = Holder

    class ResponseTyp_Def(ZSI.TCcompound.ComplexType, TypeDefinition):
        schema = "http://MyNs:8080/MatheService"
        type = (schema, "ResponseTyp")
        def __init__(self, pname, ofwhat=(), attributes=None, extend=False,
        restrict=False, **kw):
            ns = ns0.ResponseTyp_Def.schema
            TClist = [ZSI.TCnumbers.FPdouble(pname="result", aname="_result",
            minOccurs=1, maxOccurs=1, nillable=False, typed=False,
            encoded=kw.get("encoded"))]
            self.attribute_typecode_dict = attributes or {}
            if extend: TClist += ofwhat
            if restrict: TClist = ofwhat
            ZSI.TCcompound.ComplexType.__init__(self, None, TClist,
            pname=pname, inOrder=0, **kw)
            class Holder:
                typecode = self
                def __init__(self):
                    # pyclass
                    self._result = None
                return
            Holder.__name__ = "ResponseTyp_Holder"
            self.pyclass = Holder

    class SquareRequest_Def(ZSI.TCcompound.ComplexType, TypeDefinition):
        schema = "http://MyNs:8080/MatheService"
        type = (schema, "SquareRequest")
        def __init__(self, pname, ofwhat=(), attributes=None,
        extend=False, restrict=False, **kw):
            ns = ns0.SquareRequest_Def.schema
            TClist = []
            self.attribute_typecode_dict = attributes or {}
            if extend: TClist += ofwhat
            if restrict: TClist = ofwhat
            ZSI.TCcompound.ComplexType.__init__(self, None, TClist,
            pname=pname, inOrder=0, **kw)
            class Holder:
                typecode = self
                def __init__(self):
                    # pyclass
                    return
            Holder.__name__ = "SquareRequest_Holder"
            self.pyclass = Holder

    class SquareRequest_Dec(ElementDeclaration):
        literal = "SquareRequest"
        schema = "http://MyNs:8080/MatheService"
        def __init__(self, **kw):
            kw["pname"] = ("http://MyNs:8080/MatheService", "SquareRequest")
            kw["aname"] = "_SquareRequest"
            if ns0.RequestTyp_Def not in ns0.SquareRequest_Dec.__bases__:
                bases = list(ns0.SquareRequest_Dec.__bases__)
                bases.insert(0, ns0.RequestTyp_Def)
                ns0.SquareRequest_Dec.__bases__ = tuple(bases)
            ns0.RequestTyp_Def.__init__(self, **kw)
            if self.pyclass is not None: self.pyclass.__name__ =
            "SquareRequest_Dec_Holder"

    class SquareResponse_Dec(ElementDeclaration):
        literal = "SquareResponse"
        schema = "http://MyNs:8080/MatheService"
        def __init__(self, **kw):
            kw["pname"] = ("http://MyNs:8080/MatheService", "SquareResponse")
            kw["aname"] = "_SquareResponse"
            if ns0.ResponseTyp_Def not in ns0.SquareResponse_Dec.__bases__:
                bases = list(ns0.SquareResponse_Dec.__bases__)
                bases.insert(0, ns0.ResponseTyp_Def)
                ns0.SquareResponse_Dec.__bases__ = tuple(bases)

```

```

        ns0.ResponseTyp_Def.__init__(self, **kw)
        if self.pyclass is not None: self.pyclass.__name__ =
"SquareResponse_Dec_Holder"

class getModulo_Dec(ZSI.TCcompound.ComplexType, ElementDeclaration):
    literal = "getModulo"
    schema = "http://MyNs:8080/MatheService"
    def __init__(self, **kw):
        ns = ns0.getModulo_Dec.schema
        TClist = [ZSI.TC.String(pname="in", aname="_in", minOccurs=1,
maxOccurs=1, nillable=False, typed=False, encoded=kw.get("encoded"))]
        kw["pname"] = ("http://MyNs:8080/MatheService", "getModulo")
        kw["aname"] = "_getModulo"
        self.attribute_typecode_dict = {}
        ZSI.TCcompound.ComplexType.__init__(self, None, TClist, inOrder=0, **kw)
        class Holder:
            typecode = self
            def __init__(self):
                # pyclass
                self._in = None
            return
        Holder.__name__ = "getModulo_Holder"
        self.pyclass = Holder

class getModuloResponse_Dec(ZSI.TCcompound.ComplexType, ElementDeclaration):
    literal = "getModuloResponse"
    schema = "http://MyNs:8080/MatheService"
    def __init__(self, **kw):
        ns = ns0.getModuloResponse_Dec.schema
        TClist = [ZSI.TC.String(pname="out", aname="_out",
minOccurs=1, maxOccurs=1, nillable=False, typed=False,
encoded=kw.get("encoded"))]
        kw["pname"] = ("http://MyNs:8080/MatheService", "getModuloResponse")
        kw["aname"] = "_getModuloResponse"
        self.attribute_typecode_dict = {}
        ZSI.TCcompound.ComplexType.__init__(self, None, TClist, inOrder=0, **kw)
        class Holder:
            typecode = self
            def __init__(self):
                # pyclass
                self._out = None
            return
        Holder.__name__ = "getModuloResponse_Holder"
        self.pyclass = Holder

class ModuloRequest_Dec(ZSI.TCcompound.ComplexType, ElementDeclaration):
    literal = "ModuloRequest"
    schema = "http://MyNs:8080/MatheService"
    def __init__(self, **kw):
        ns = ns0.ModuloRequest_Dec.schema
        TClist = [ZSI.TCnumbers.Iint(pname="Zahl1", aname="_Zahl1",
minOccurs=1, maxOccurs=1, nillable=False, typed=False,
encoded=kw.get("encoded")), ZSI.TCnumbers.Iint
(pname="Zahl2", aname="_Zahl2", minOccurs=1, maxOccurs=1, nillable=False,
typed=False, encoded=kw.get("encoded"))]
        kw["pname"] = ("http://MyNs:8080/MatheService", "ModuloRequest")
        kw["aname"] = "_ModuloRequest"
        self.attribute_typecode_dict = {}
        ZSI.TCcompound.ComplexType.__init__(self, None, TClist, inOrder=0, **kw)
        class Holder:
            typecode = self
            def __init__(self):
                # pyclass
                self._Zahl1 = None
                self._Zahl2 = None
            return
        Holder.__name__ = "ModuloRequest_Holder"
        self.pyclass = Holder

class ModuloResponse_Dec(ZSI.TCcompound.ComplexType, ElementDeclaration):
    literal = "ModuloResponse"
    schema = "http://MyNs:8080/MatheService"
    def __init__(self, **kw):
        ns = ns0.ModuloResponse_Dec.schema
        TClist = [ZSI.TCnumbers.Iint(pname="result", aname="_result",
minOccurs=1, maxOccurs=1, nillable=False, typed=False,
encoded=kw.get("encoded"))]
        kw["pname"] = ("http://MyNs:8080/MatheService", "ModuloResponse")
        kw["aname"] = "_ModuloResponse"
        self.attribute_typecode_dict = {}
        ZSI.TCcompound.ComplexType.__init__(self, None, TClist, inOrder=0, **kw)
        class Holder:
            typecode = self
            def __init__(self):
                # pyclass
                self._result = None
            return
        Holder.__name__ = "ModuloResponse_Holder"
        self.pyclass = Holder

# end class ns0 (tns: http://MyNs:8080/MatheService)

```

In der Datei *MatheService_services_server.py* implementieren wir nun wieder unsere Verarbeitungsmethoden und sorgen dafür, dass sie beim Aufruf die geeigneten Parameter übergeben bekommen:

MatheService_services_server.py

```

def soap_getSquare(self, ps):
    self.request = ps.Parse(getSquareRequest.typecode)
    x = self.request._x
    Response = getSquareResponse()

```

```

    Response._result=self.getSquare(x)
    return Response

def getSquare(self, x):
    return x**2

soapAction ['http://MyNs:8080/MathService/getSquare'] = 'soap_getSquare'
root [(getSquareRequest.typecode.nspname, getSquareRequest.typecode.pname)]
= 'soap_getSquare'

def soap_getModulo(self, ps):
    self.request = ps.Parse(getModuloRequest.typecode)
    Parameters = (self.request._Zahl1, self.request._Zahl2)
    result = getModuloResponse()
    result._result = self.getModulo(Parameters[0], Parameters[1])
    return result

def getModulo(self, x, y):
    return x % y

soapAction ['http://MyNs:8080/MathService/getModulo'] = 'soap_getModulo'
root [(getModuloRequest.typecode.nspname, getModuloRequest.typecode.pname)]
= 'soap_getModulo'

```

Wir können nun den Server starten und mit SoapUI testen:

SOAPUI Request

```

<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:mat="http://MyNs:8080/MathService">
  <soapenv:Header/>
  <soapenv:Body>
    <mat:ModuloRequest>
      <Zahl1>9</Zahl1>
      <Zahl2>5</Zahl2>
    </mat:ModuloRequest>
  </soapenv:Body>
</soapenv:Envelope>

```

SOAPUI Response

```

<SOAP-ENV:Envelope
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body xmlns:ns1="http://MyNs:8080/MathService">
    <ns1:ModuloResponse>
      <result>4</result>
    </ns1:ModuloResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Glückwunsch, der Server funktioniert!

4.3 Der erweiterte Client

Der Client selbst ist einfach zu implementieren, da er nur die getter Methoden der *MatheService_services.py* aufruft. in unserem Beispiel sieht das so aus:

MyMatheServiceClient.py

```
from MatheService_services import *
from MatheService_services_types import *
import sys
from optparse import OptionParser

parser = OptionParser()

parser.add_option("-s", "--square", dest="square", type="string",
help="Quadrat_einer_Zahl_errechnen")
parser.add_option("-a", "--zahl1", dest="Zahl1", type="string",
help="Erster_Parameter_der_Modulodivision")
parser.add_option("-b", "--zahl2", dest="Zahl2", type="string",
help="Zweiter_Parameter_der_Modulodivision")

(options, args)=parser.parse_args()

loc = MatheServiceLocator()
proxy = loc.getMathePortType(tracefile = sys.stdout)

if options.square:
    response= proxy.getSquare((float)(options.square))
    print "Das_Quadrat_von_", options.square, " ist: ",response

elif options.Zahl1 and options.Zahl2:
    response= proxy.getModulo((int)(options.Zahl1),(int)(options.Zahl2))
    print "Der_Ganzzahlige_Rest_der_modulodivision_von_", options.Zahl1,
    "und", options.Zahl2, " betr" agt" ",response
```

Interessanter als der Client ist *MatheService_services.py*. Ich hatte hier oftmals Probleme beim Erzeugen des Codes. Folgender Fehler trat bei mir reproduzierbar auf:

Fehlerhafte MatheService_services.py

```
class MatheBindingSOAP:
    def __init__(self, url, **kw):
        kw.setdefault("readerclass", None)
        kw.setdefault("writerclass", None)
        # no resource properties
        self.binding = client.Binding(url=url, **kw)
        # no ws-addressing

    # op: <ZSI.wstools.WSDLTools.Message instance at 0x4070252c>
    def getSquare(selfx):

        request = getSquareRequest()
        request._x = x

        kw = {}
        # no input wsaction
        self.binding.Send(None, None, request, soapaction=
"http://MyNs:8080/MatheService/getSquare", **kw)
        # no output wsaction
        response = self.binding.Receive(getSquareResponse.typecode)
        return
```

```

# op: <ZSI.wstools.WSDLTools.Message instance at 0x4070296c>
def getModulo(self , Zahl1 ,Zahl2):

    request = getModuloRequest()
    request._Zahl1 = Zahl1
    request._Zahl2 = Zahl2

    kw = {}
    # no input wsaction
    self.binding.Send(None, None, request , soapaction=
" http://MyNs:8080/MathService/getModulo" , **kw)
    # no output wsaction
    response = self.binding.Receive(getModuloResponse.typecode)
    result = response._result
    return result

```

In der Funktion `getSquare` wurden die Parameter `self` und `x` nicht durch ein Komma getrennt, was beim Aufrufen der Funktion vom Client aus dazu führt, dass wir die Fehlermeldung, *getSquare erwartet genau einen Parameter, aber zwei wurden übergeben*, erhalten. Der nächste Fehler ist, dass die Funktion nichts zurück gibt. Hier musste in der Funktion `getSquare()` noch die Zeile: `result = response._result` und `return result` hinzugefügt werden.

Das komplette Projekt findet ihr im Downloadbereich

Das wars, der Client läuft!

5 Deutsche Zusammenfassung des ZSI user manual

Das folgende Kapitel ist NICHT die vollständige Übersetzung des ZSI user guide, sondern gibt nur Teile, die ich für wesentlich erachtet habe, wieder. Ich übernehme ausdrücklich keine Gewähr für die Richtigkeit der Übersetzung. Im Zweifelsfall bitte immer im Original nachlesen! Ihr findet es unter:

<http://pywebsvcs.sourceforge.net/guide.html>

ZSI, die Zolera SOAP Infrastructure, ist ein Python Paket, das eine Implementation von SOAP gemäß SOAP 1.1 Spezifikation bereitstellt.

5.1 Übersicht

Das ZSI Web- Service Paket ist ein Tool zur top- down Entwicklung von Web- Services. Dies bedeutet, dass eine vorhandene WSDL verwendet wird, um Client und Serveranwendungen zu erstellen. Im Kontext dieses Dokuments bezeichnet ein Web- Service eine WSDL Datei, die das Service Interface beschreibt. Solche Dokumente sind üblicherweise unter einer veröffentlichten URL zu finden. Die WSDL Datei definiert die SOAP Bindings zur Kommunikation mit dem Service. Diese Bindings werden verwendet, um die SOAP Nachrichten auszutauschen. Der Inhalt dieser Nachrichten muss entsprechend der Struktur, die durch das Schema gegeben wird, eingefügt werden. Dieses Schema ist entweder in der WSDL Datei enthalten, durch sie eingebunden oder es wird durch die zur Verfügung stehenden Typen (wie xsd:int, xsd:string, usw.) repräsentiert.

5.1.1 SOAP Bindings

Es gibt zwei Formen von SOAP Bindings, *rpc* und *document*. Die Verwendung der wörtlichen (literal) Kodierung ist die empfohlene Art neue Web- Service Anwendungen zu entwickeln. Die Unterstützung der *rpc/enc* SOAP Kodierung wird für den Gebrauch mit älteren Applikationen und anderen SOAP Toolkits, die auf *rpc/enc* beschränkt sind, beibehalten. Durch einen *doc/literal* Service wird gewöhnlich der Austausch von Dokumenten beschrieben, während ein *rpc/enc* oder *rpc/literal* Service für Remoteaufrufe gedacht ist. Ob diese Unterscheidung des Zweckes sinnvoll oder nützlich ist, ist strittig. ZSI unterstützt alle drei Arten, aber *rpc/literal* und *doc/literal* sind der Fokus der laufenden Entwicklung.

5.1.2 Python Werkzeuge

wSDL2py Das `wSDL2py` Skript erzeugt den Pythoncode, der die verschiedenen Bestandteile darstellt, die in einem WSDL Dokument definiert werden.

wSDL2py Das Skript `wSDL2dispatch` sollte nach `wSDL2py` ausgeführt werden, da `wSDL2dispatch` den durch `wSDL2py` generierten Code importiert. Dieses Skript erzeugt ein Modul, das eine Service-Schnittstelle enthält, die von der WSDL definiert wird. Diese Schnittstelle wird für gewöhnlich durch einen HTTP Server eingebunden.

5.2 Von der WSDL zum Python Code

Das `wSDL2py` Skript ist das Primärwerkzeug. Es erzeugt den ganzen Code, der benötigt wird, um einen Web- Service durch eine veröffentlichte WSDL zugänglich zu machen. Normalerweise ist die WSDL über eine URL erreichbar, die dem Skript zur Verfügung gestellt wird.

`wSDL2py` erzeugt ein 'stub' Modul aus den WSDL SOAP Bindings. Dieses Modul enthält verschiedene Klassen, einschließlich einem Locator, der die Bindings zum tatsächlichen Web Service darstellt, einige Port Klassen, die benutzt werden, um Operationen auf dem Web- Service aufzurufen sowie verschiedene Nachrichten Klassen, welche die Datentypen des SOAP und XML Schemas darstellen. Die Instanz einer Nachricht wird in XML dargestellt. Eine Nachricht, die als Argument einer Portmethode übergeben wird, wird dann in einen SOAP Envelope überführt und zum WEB-

Service übertragen. Der Client erwartet dann eine Antwort. Schließlich wird die SOAP Antwort zurück in die Nachricht überführt und zum Benutzer zurückgegeben.

Ein zweites Modul, das 'types' Modul, enthält die Typecodes, die alle Bestandteile des Schemas darstellen, welche durch die WSDL spezifiziert werden (außer den 'build in types' wie xsd:int, usw.). Alle Schemabestandteile, die im top- level deklariert sind, also die unmittelbaren Kinder des Schema- Tags, haben einen globalen Geltungsbereich. Durch Einbinden des 'types' Moduls erhält eine Anwendung Zugriff auf die globalen Element und / oder Typ- Definitionen.

5.2.1 wsdl2py - Die Grundlagen der Codeerstellung

Client Modul: Durch den Aufruf von wsdl2py ohne zusätzliche Parameter wird aus der WSDL ein Client Gerüst erstellt, das die in der WSDL hinterlegten Informationen über den Service, die Bindings, den Porttyp und über die Nachrichten enthält.

- class Locator(**keywords)

Die folgenden Schlüsselwörter können verwendet werden:

Die vier Einzelteile (Service, Binding, Port und Nachricht) werden durch drei Abstraktionen dargestellt. Diese bestehen aus einer Locator- Klasse, einer PortType -Klasse und einigen Nachrichten-Klassen. Der Locator hat zwei Methoden für jeden Service- Port, der in der WSDL definiert wird. Eine Methode gibt die Adresse zurück, die im Binding spezifiziert wird und die andere ist eine Factory- Methode für die Rückgabe der PortType Instanz. Jede Nachrichtenklasse repräsentiert Teile der Bindings auf der Ebene der Operationen und darunter sowie alle Typinformationen, die durch Nachrichtenteile beschrieben werden.

Types Modul: Das Typen-Modul wird zusammen mit dem Client-Modul erstellt, es kann aber auch unabhängig davon erzeugt werden. Es eignet sich besonders zum Umgang mit Schema Definitionen, die nicht in einer WSDL spezifiziert sind.

Die Klassen auf Modulebene stellen einen einzigartigen Namespace dar, der einfach als Verpackung der individuellen Namespaces dient. Die inneren Klassen sind die typecode Repräsentation der globalen Typ- (vorgestelltes _Def) und Elementdefinitionen (vorgestelltes _Dec).

Typecodes verstehen: Wie oben ausgeführt, gibt es zwei Arten von typecode Klassen. Elementdefinitionen können direkt in XML überführt werden, Typdefinitionen im Allgemeinen aber nicht. Einfach ausgedrückt bedeutet dies, dass das Attribut 'name' einer Elementdeklaration in ein XML- Tag überführt wird. Da Typendefinitionen kein Attribut 'name' besitzen, können sie nicht direkt in einer XML Instanz abgebildet werden. Die meisten Elemente definieren ein 'type' Attribut, welches auf eine Typendefinition verweisen muss. Wenn man also die vorangegangene Ausführung beachtet kann man feststellen: Eine generierte TypeCode Klasse, die eine Elementdefinition darstellt beinhaltet eine TypeCode Sub-Klasse mit der Typdefinition.

pyclass: Alle Instanzen von generierten TypeCode Klassen haben ein Attribut 'pyclass'. Instanzen von Pyclass werden erzeugt, um die Daten zu speichern, die eine Elementdeklaration darstellen. Die Pyclass selbst hat ein Attribut 'typecode', das ein Verweis auf die TypeCode Instanz darstellt, welche die Daten beschreibt. Dies macht pyclass Instanzen selbst beschreibend. Wenn eine XML Instanz geparkt wird, werden die Daten einer pyclass Instanz zugeordnet.

ANAME: Aname ist ein Attribut einer TypeCode Instanz. Sein Wert ist ein String, der den Attributnamen darstellt, welcher verwendet wird, um die Daten zu verweisen, die eine Elementdeklaration darstellen. Der Satz an XMLSchema Elementnamen wird als 'NCName' bezeichnet, dieses ist eine Obermenge der gewöhnlichen Bezeichner in Python.

Namespaces in XML:

```
From Namespaces in XML
NCName ::= (Letter — '_' ) (NCNameChar)*
NCNameChar ::= Letter — Digit — '.' — '-' — '_' — CombiningChar — Extender
From Python Reference Manual (2.3 Identifiers and keywords)
identifier ::= (letter—'_') (letter — digit — '_')*
Default set of anames
ANAME ::= ('_') (letter — digit — '_')*
```

NCName in AName transformieren:

1. Vorangestellter Unterstrich '_'
2. Zeichen, die nicht in dem Zeichensatz(Buchstabe, Ziffer, '_') vorkommen, werden durch '_' ersetzt.

Attribut Deklarationen: attrs_aname: Attrs_aname ist ein Attribut einer TypeCode Instanz. Sein Wert ist eine Zeichenkette, die den Attributnamen darstellt, der verwendet wird, um auf ein Dictionary zu verweisen. Dieses enthält Daten, welche die Attributdeklaration darstellen. Die Schlüssel dieses dictionary sind die (namespace, Name) Tupel. Der Wert jedes Schlüssels stellt den Wert des Attributes dar.

5.2.2 Typecode Erweiterungen

Die Option `-complexType`, `-b`: Die Option `-complexType` beim Aufruf von `wsdl2py` stellt dem Programmierer viele Vereinfachungen zur Verfügung. Diese Option ist getestet und die Verwendung wird von den Autoren empfohlen.

low-level Beschreibung: Wenn die Option `-complexType` gesetzt ist wird allen generierten pyclasses das Attribut `__metaclass__` hinzugefügt. Die Metaklasse prüft die TypeCode Attribute von pyclass und erstellt einen Satz von Hilfsmethoden für jedes Element und Attribut, das in der Definition des ComplexType angegeben ist. Diese Option fügt Wrapper für den Umgang mit dem Inhalt hinzu, ohne das Generierungsschema zu ändern.

Getter, Setter: Eine getter und eine setter Methode wird für jedes Element eines komplexen Typen definiert. Diese Methoden werden `get_element_ANAME` und `set_element_ANAME` genannt.

Factory Methoden: Wenn ein Element eines komplexen Typs selbst wieder ein komplexer Typ ist, so wird zum einfacheren Umgang eine factory Methode erzeugt, die eine Instanz der holder Klasse des Typs zurückgibt. Die factory Methode wird `'newANAME'` genannt.

Eigenschaften: Für Python Klassen werden für jedes Element eines komplexen Typs Eigenschaften (properties) erzeugt. Diese werden auf die entsprechenden getter und setter Methoden des Elements abgebildet. Um Namensüberschneidungen zu vermeiden, werden diese Eigenschaften `'PNAME'` genannt, wobei der erste Buchstabe des pname Attributs eines Typs groß geschrieben wird.

6 Links

Hier findet ihr Links zum Download der verwendeten Tools:

- EntwicklungsIDE Eclipse: <http://www.eclipse.org>
- Zolera SOAP Infrastructure (ZSI) <http://pywebsvcs.sourceforge.net/>
- Python 2.5 <http://www.python.org/download/releases/2.5/>
- PyXML <http://sourceforge.net/projects/pyxml/>
- soapUI 1.5 <http://www.soapui.org/>