Università degli Studi di Pisa

Laurea Magistrale in Informatica

Laurea Magistrale in Informatica e Networking

Advanced Programming

Final Term Paper

Start Date: 21/01/2011
Submission deadline: 28/02/2011 (send a single PDF file to attardi@di.unipi.it)

Rules:

The paper must be produced personally by the student, signed implicitly via his mail address.

You are allowed to discuss with others the general lines of the problems, provided that each student eventually formulates his own solution. Each student is expected to understand and to be able to explain his solution.

You are allowed to consult documentation from any source, provided that references are mentioned.

It is not considered acceptable:

- to develop code or pseudo-code with others
- to use code written by others
- to let others use someone's code
- to show or to examine the work of other students.

Violation of these rules will result in the cancellation of the exam and a report to the Presidente del Consiglio di Corso di Studio.

For the programming exercises you can choose a programming language among C++, C# and Java.

The paper must:

1. be in a single PDF file, formatted readably (font size ≥ 9pt with suitable margins), of **no more than 10 numbered pages**, including code: for each extra page one point will be subtracted from the score.
2. include **the student name**
3. **provide the solution and the code for each exercise separately**, referring to the code of other exercise if necessary.
4. cite references to literature or Web pages from where information was taken.

Introduction

The goal of the project is to develop a simplified Web Services engine. The engine provides means to turn classes providing services into Web Services invoked using SOAP and allowing clients to invoke those services. The process involves generating WSDL from code and code from WSDL.

Consider a class providing access to an address book like this:

```
public class Name {
    public String first;
    public String last;
}
public class Entry {
    public Name name;
    public String email;
}
public class AddressBook {
    public int add(Entry entry) { ... }
    public Entry find(String name) { ... }
    ...
}
```

We will use a simplified version of WSDL that has the following structure:

```
<definitions>
 <types>
  <complexType name="Name">
   <sequence>
      <element  name="first" type="String" />
      <element  name="last" type="String" />
   </sequence>
  </complexType>
  <complexType name="Entry">
   <sequence>
      <element  name="name" type="Name" />
      <element  name="email" type="String" />
   </sequence>
  </complexType>
 </types>
 <message name="addRequest">
    <part  name="entry" element="Entry" />
 </message>
 <message name="addResponse">
    <part  name="result" element="int" />
 </message>
 <operation name="add">
    <input  message="addRequest" />
    <output  message="addResponse" />
 </operation>
 <message name="findRequest">
    <part  name="name" element="String" />
 </message>
 <message name="findResponse">
    <part  name="result" element="Entry" />
 </message>
 <operation name="find">
    <input  message="findRequest" />
    <output  message="findResponse" />
 </operation>
 <service name="AddressBook"
   location="http://localhost/AddressBook" />
 </definitions>
```

Simplified SOAP (SSP) messages will have the following formats:

```
<ssp:request>
  <add>
    <Entry>
        <Name name="name">
              <String name="first">Linus</String>
              <String name="last"> Torvalds</String>
        </Name>
        <String name="email">torvalds@linuxfoundation.org</String>
    </Entry>
  </add>
</ssp:request>

<ssp:response>
  <int>0</int>
</ssp:response>
```

Exercise 1

Given a Java class that implements a service, generate the corresponding WSDL, using the Doclet API (http://download.oracle.com/javase/1.5.0/docs/guide/javadoc/doclet/spec/index.html) to parse the Java code.

The WSDL should contain an operation for each public method of the class; the types in the definitions should be the non primitive types used as parameters or results for those methods and it is assumed that those types will be defined in the same file as the service class.

Exercise 2

Implement a stub generator that, given com.sun.javadoc.ClassDoc for a service class, generates a stub for the service, i.e. a class whose methods receive a string containing an SSP request, unmarshal the parameters, invoke the corresponding method of the service class and return an SSP response representing the result of the invocation. Hint: generate unmarshalling methods for each non primitive type.

Show the code generated for the types in the example.

Exercise 3

Given a WSDL file generated from Exercise 1, write a code generator, which generates two methods for each type in the WSDL:

- one that reads a parameter of that type from an SSP message and returns the corresponding object.
- one that marshals a parameter of that type into an SSP message.

Show the code generated for the types in the example.

Exercise 4

Using the code produced by the generators in the previous exercises, write a program which reads an SSP request for the AddressBook service, invokes the proper stub and prints its result.

Exercise 5

Introduce Generic types and in particular discuss and compare C# constrains on type parameters and Java bounded polymorphism.

Exercise 6 (for students of the Laurea Specialistica)

Specialize the generator in Exercise 1 to only produce WSDL for those methods that are annotated as `@WebMethod`.