UNIVERSITEIT TWENTE.

instructors only

Exercise series 1-2

Building a web service

DATE	:	*06-03-2015*
COURSE	:	Service-Oriented Architectures with Web Services
COURSE CODE	:	192652150
GROUP	:	8
STUDENTS[S]	:	*Floris Smit, s1253999*
		Jochem Verburg, s1233998

Table of contents

1. WSDL DEFINITION 1
1.1 OPERATIONS AND DATA TYPES 1
1.2 WSDL SPECIFICATION 1
1.2.1 Modeling choices 2
1.3 WEB SERVICE IMPLEMENTATION 3
1.4 TESTING 3
1.4.1 Automated tests 3
1.4.2 GUI 4
1.4.3 Conclusion 5

2. RADIOLOGY 5
2.1 DEVELOPMENT STEPS 5
2.2 SOFTWARE DESIGN 6
2.2.1 Server-side 6
2.2.2 Client-side 7
2.3 SYSTEM CONFIGURATION 8
2.4 DEPLOYMENT AND ADDRESSING 9
2.5 TESTING 10

Introduction

For this series a WSDL-definition and an implementation of a hotel booking web service have been made and using existing WSDL-definitions a radiology web service has been implemented. Since the second WSDL-definition uses document style, the first WSDL specification was made using RPC style to get a bit of experience with both.

1. WSDL Definition

1.1 Operations and data types

The web service needs two operations: *checkHotelAvailability* and *bookHotelRoom*. Each of these fulfill one of the requirements.

The data types needed as input *checkHotelAvailability* for are: *String* (city), *Date* (date) and *int* (number of guests). The output is a list of integers (the matching hotel codes), which is empty if there are no hotels available.

The data types needed as input for *bookHotelRoom* are: *int* (hotel code), *Date* (date) and *int* (number of guests). The output is *Boolean* defining whether the booking was successful or not.

1.2 WSDL specification

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HotelWS" targetNamespace="http://www.example.com/Hotel" xmlns:mh="http://www.example.com/Hotel"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/">
        <types>
                <xsd:schema targetNamespace="http://www.example.com/Hotel"
xmlns:mh="http://www.example.com/Hotel" >
                        <xsd:complexType name='availableHotelList' >
                                <xsd:sequence>
                                        <xsd:element
                                        minOccurs='0'
                                        maxOccurs='unbounded'
                                        name='hotelCode'
                                        type='xsd:int'/>
                                </xsd:sequence>
                        </xsd:complexType>
                </xsd:schema>
        </types>

        <!-- Message describes input and output parameters -->
        <message name="CheckHotelAvailabilityRequest">
                <part name="city" type="xsd:string"/>
                <part name="numberOfGuests" type="xsd:int"/>
                <part name="date" type="xsd:date"/>
        </message>
        <message name="CheckHotelAvailabilityResponse">
                <part name="hotelCodeList" type="mh:availableHotelList"/>
        </message>
        <message name="BookHotelRoomRequest">
                <part name="date" type="xsd:date"/>
```

```
                <part name="hotelCode" type="xsd:int"/>
                <part name="numberOfGuests" type="xsd:int"/>
        </message>
        <message name="BookHotelRoomResponse">
                <part name="confirmed" type="xsd:boolean"/>
        </message>
        <!-- PortType describes abstract interface of Web service -->
        <portType name="Hotel">
                <operation name="checkHotelAvailability">
                        <input name="availabilityInput" message="mh:CheckHotelAvailabilityRequest"/>
                        <output name="availabilityOutput" message="mh:CheckHotelAvailabilityResponse"/>
                </operation>
                <operation name="bookHotelRoom">
                        <input name="hotelBookingInput" message="mh:BookHotelRoomRequest"/>
                        <output name="hotelBookingOutput" message="mh:BookHotelRoomResponse"/>
                </operation>
        </portType>
        <!-- Binding defines the protocols and encoding styles  -->
        <binding name="HotelBinding" type="mh:Hotel">
                <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
                <operation name="checkHotelAvailability">
                        <soap:operation style="rpc"/>
                        <input>
                                <soap:body use="literal" namespace="http://www.example.com/Hotel" />
                        </input>
                        <output>
                                <soap:body use="literal" namespace="http://www.example.com/Hotel"/>
                        </output>
                </operation>
                <operation name="bookHotelRoom">
                        <soap:operation style="rpc"/>
                        <input>
                                <soap:body use="literal" namespace="http://www.example.com/Hotel" />
                        </input>
                        <output>
                                <soap:body use="literal" namespace="http://www.example.com/Hotel"/>
                        </output>
                </operation>
        </binding>
        <!-- Service defines the address of Web service -->
        <service name="HotelService">
                <port name="HotelPort" binding="mh:HotelBinding">
                        <soap:address location="http://localhost:8080/Hotel/services/HotelService"/>
                </port>
        </service>
</definitions>
```

1.2.1 Modeling choices

For the message part of the wsdl-specification a simple solution was chosen. Instead of modeling the object which would hold all the data for an operation, separated parts were used. The 'OO' way of doing it would be to use an object HotelBooking which maps directly to a java object. Instead, we put all the arguments required to complete operations in separate parts. The reason we did this was to reduce the amount of custom types used.

Both operations require that a response is sent back from the webservice. The checkHotelAvailability operation has to return whether the hotel is available. Likewise the bookHotelRoom operation has to return a confirmation or rejection of the booking.

As an interaction style RPC was used. We wanted to try both of them and assignment 2, radiology uses document style. So for assignment 1 the RPC interaction style was chosen.

1.3 Web service implementation

The first step in the implementation of the web service was creating a model. This model contains all the business logic and is kept in a separate model package. In that package there are some classes that contain the model of the Hotels. Hotel and Room, contain all the necessary information about Hotels and Rooms. Room has a HashMap<Date, Boolean> that stores when it is occupied. HotelModel contains hotels and has an init function that creates some hotels and rooms to test the webservice. In HotelServiceSkeleton a static variable is added to store the HotelModel. In the generated stub functions the HotelModel is called to execute the desired operations or retrieve the desired information.

1.4 Testing

There were two test methods used: an automated test client, and a GUI.

1.4.1 Automated tests

Having an automated way of testing the deployed service is a nice thing. After deploying a service anywhere this program can be used to test all the edge usecases of the service. The test program uses a stub generated by axis2 based on the Hotel.wsdl file.

The following cases were tested:
- CheckHotelAvailability for 3 valid cities.
- CheckHotelAvailability for an invalid city.
- CheckHotelAvailability for an unsupported number of guests.
- Booking 3 rooms for 2 persons in 1 hotel(the maximum per hotel).
- Booking a 4th room on the same day with the same amount of guests should fail.
- Booking a room in a different city should work.
- Booking a room with different amount of guests should work.
- Booking a room on a different date should work.

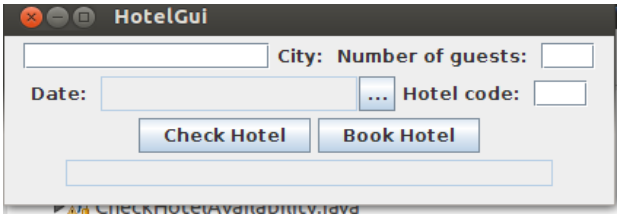
The outpout of these tests were the following:

Testing CheckHotelAvailability for Amsterdam
Test passed: HotelCodes returned equal to [2,3]
Testing CheckHotelAvailability for Enschede
Test passed: HotelCodes returned equal to [0,1]
Testing CheckHotelAvailability for Utrecht
Test passed: HotelCodes returned equal to [4,5]
Testing CheckHotelAvailability for non existing city abcd
Test passed: HotelCodes returned equal to null
Testing CheckHotelAvailability for non existing room capacity: 3
Test passed: HotelCodes returned equal to null
Testing BookHotelRoom: booking 3 standard rooms for today (3 max available)
Test passed: 1st booking succes
Test passed: 2nd booking succes
Test passed: 3rd booking succes
Test passed: 4th booking was rejected
Testing BookHotelRoom: booking a standard room in a hotel should succeed
Test passed: Booking room in different hotel succeeded
Testing BookHotelRoom: booking a room with different amount of guests should succeed
Test passed: Booking room for a different number of guests succeeded
Testing BookHotelRoom: booking a room on a different date should succeed

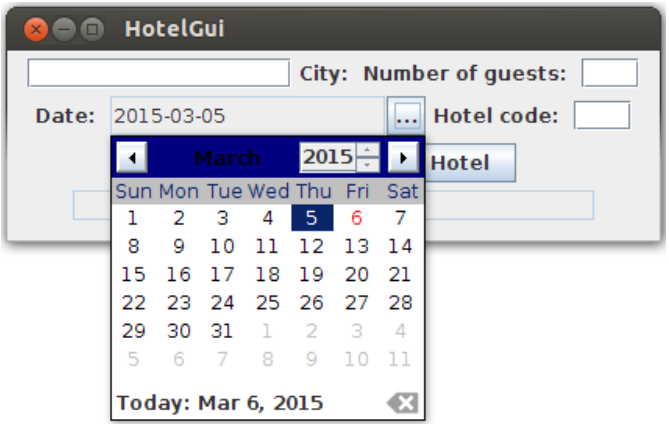
Test passed: Booking room on a different date succeeded

1.4.2 GUI

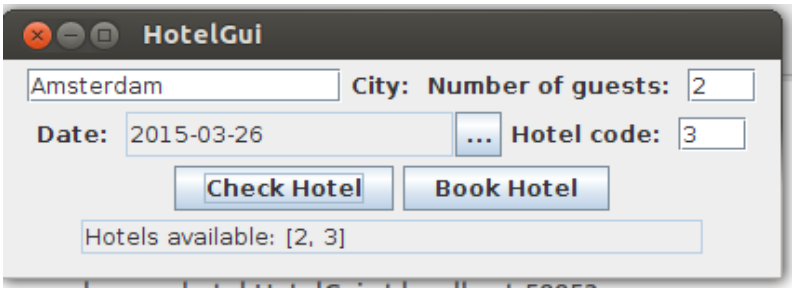
The GUI can be used to check the availability of hotels in cities and booking hotel rooms. The general look of the GUI is visible in the image below:



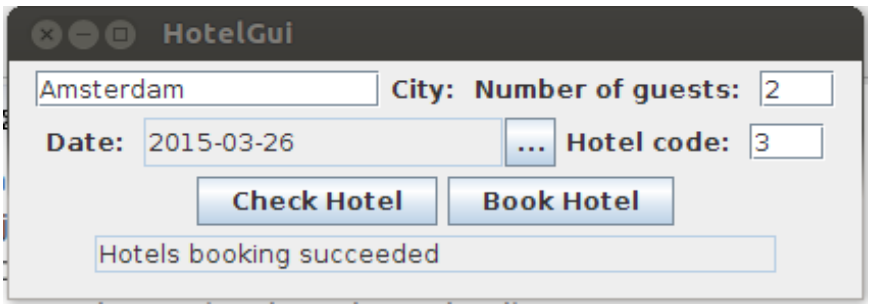
For an easy date selection a java swing date picker was used.[1]



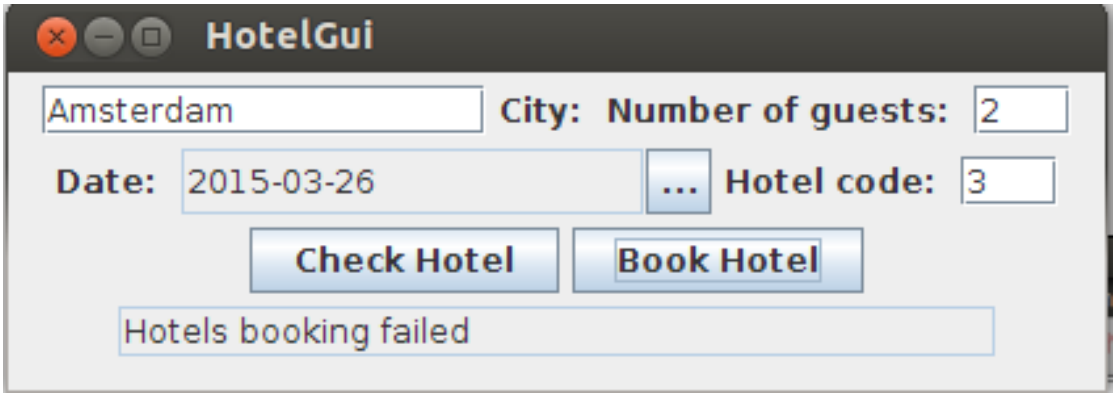
Filling in the values and checking available hotels prints a line with hotel codes in the un-editable response text field.



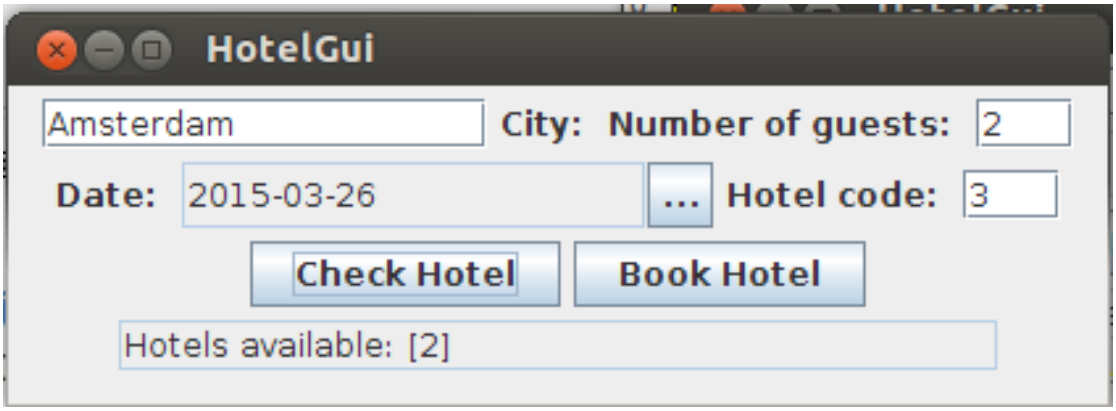
Booking a hotel room gives a confirmation message.



After booking 3 rooms, when trying to book the same room again, a message is shown to inform the user that the booking has failed.



When checking available hotels for the same city and date again, only hotel with hotel id 2 is available. Hotel 3 is full.



1.4.3 Conclusion

These two testing clients provide thorough testing. The automated tests allow for systematic review of functionality. The GUI allows for user testing and dynamically adding data to the web service.

2. Radiology

2.1 Development steps

First, the tutorial was followed to generate the web service skeleton using the WSDL-file of the RadiologyService for the server. This generated several classes like the skeleton, the interface, but also classes for the arguments of each of the operations. Secondly, a client was generated using the same WSDL-file after changing the location of the server in the file: this gave the classes RadiologyServiceStub and RadiologyServiceCallbackHandler. To be able to implement the RadiologyCallback service, the WSDL-file was used to generate the server-side of this service on the client. This generated now in the client-application, the skeleton, skeleton interface, message receivers, and also a class for the data which is sent to this

service. Afterwards for RadiologyCallback service, the client was generated for the server-application, so it could call the RadiologyCallback service.

The next step in development was to implement the server, which could be done in many different ways. For this project Lists were used into which every new radiology order and its information would be saved. In more practical settings probably it would be better to make a connection with a database (service) or use other ways of storing the data. The server-side was implemented in a simple way, for example throwing errors if appointments were already made. During every call affecting the report of an order, the report was edited. In this case it was chosen to send the reports in a random time interval after an appointment was ordered. The sending of the report was done by using callback-stub. In practice the report would probably be send only after the date of the appointment or when the results were entered by someone.

Since it was not clear to us how to run a client as a service at the same time we made a small separation in the client-application. A GUI is implemented which uses the RadiologyServiceStub to send commands. The responses are shown in the GUI. This is a very simple GUI. In practice web services would very well lend itself for example to be accessed through a web interface. This GUI can be run as a Java-application.

Next to this the RadiologyCallbackServiceSkeleton is implemented in the same application, which makes sure that the service can be run on a (Tomcat-)server. To show how this can still be used as a client, a GUI is created every time a report is received to show the report (this could for example be useful if further operations with this report has to be done, the GUI could be extended to for example give buttons to sign the report or do other actions using other services). We were not able to define this service to also have a GUI show up on start-up (which could then be the ClientGUI), this could help show the results on the GUI which was used to send requests.

2.2 Software design

2.2.1 Server-side

Every class generated to be used as arguments for an operation has a Factory class in it, these have been left out of the diagram since they do not really add any value as long as you know that they’re there.

The automatically generated files make sure that the methods of the RadiologyServiceSkeleton are called with the arguments already put into objects and with no need for the implementation to think of how the connections work. The Skeleton can use all the generated classes to get the arguments of operations, but also to return and save results.

Besides this, the Skeleton uses the generated client-side of the callback-service. It uses the RadiologyCallbackServiceStub to asynchronously send the reports. Since the RadiologyReport defined in the RadiologyService is the same as the one defined in the RadiologyCallbackService, the Stub inside the server was changed so the implementation of the server could send objects of the type RadiologyReport in the RadiologyService-package. This could be done since both the inner-class in the RadiogyCallbackServiceStub as the class for the RadiologyService were the same.

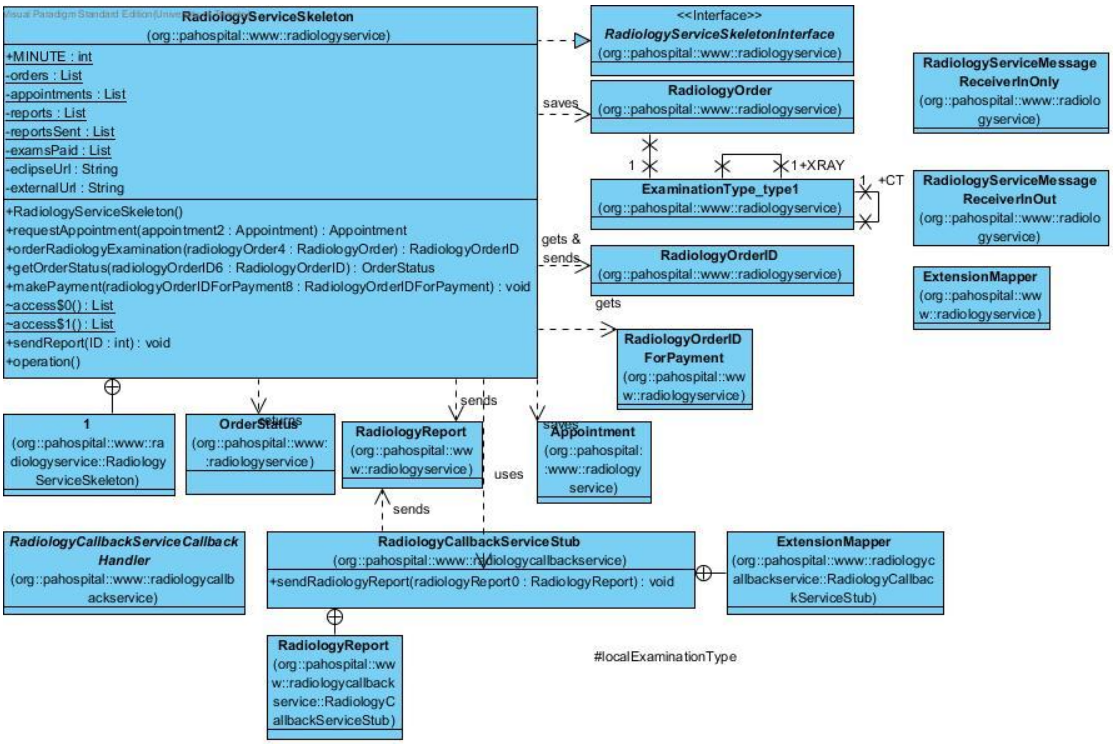


Figure 1 Server-side

2.2.2 Client-side

As explained earlier the client-side is kind of divided into two parts. The part implementing the RadiologyCallback and the part implementing the client-side of the RadiologyService. Currently these are not really inter-connected.

2.2.2.1 Client-side RadiologyService

The client-side of the RadiologyService is implemented by making a GUI. The Stub was generated from wsdl-file and this Stub was used to execute the operations. The GUI has a ButtonListener which is added to each of the Buttons and looks which button was pressed and depending on that sends a message to the service. It uses the inner classes of the Stub to put the data entered on the GUI into objects which can be send. Then the Stub is used to send the objects and afterwards if there's a response the inner classes are used again to put the information inside the objects onto the screen.

2.2.2.2 RadiologyCallbackService

The RadiologyCallbackService is really simple, it should handle the incoming report. This could be done by putting the report on the System-output, but to show the possibilities a GUI was chosen. As soon as a report has come in, a GUI is started which shows the report. As explained before, this GUI could then also be used to execute further actions. There are of course numerous other possibilities for the implementation such as sending emails, giving people the opportunity to subscribe to get the correct reports or saving the reports and sending them upon request. Since the wsdl-file was already specified in this way, it was chosen to generate a GUI.

2.2.2.3 Combination

Currently, to let it function as a combined client, the RadiologyCallbackService should be run on a Tomcat-server on the same computer as the ClientGUI is opened. In that case the requesting unit would really see the report. We did not manage to generate a ClientGUI on start-up of the service. This could give the opportunity to show the report in the same GUI as the requests were made.

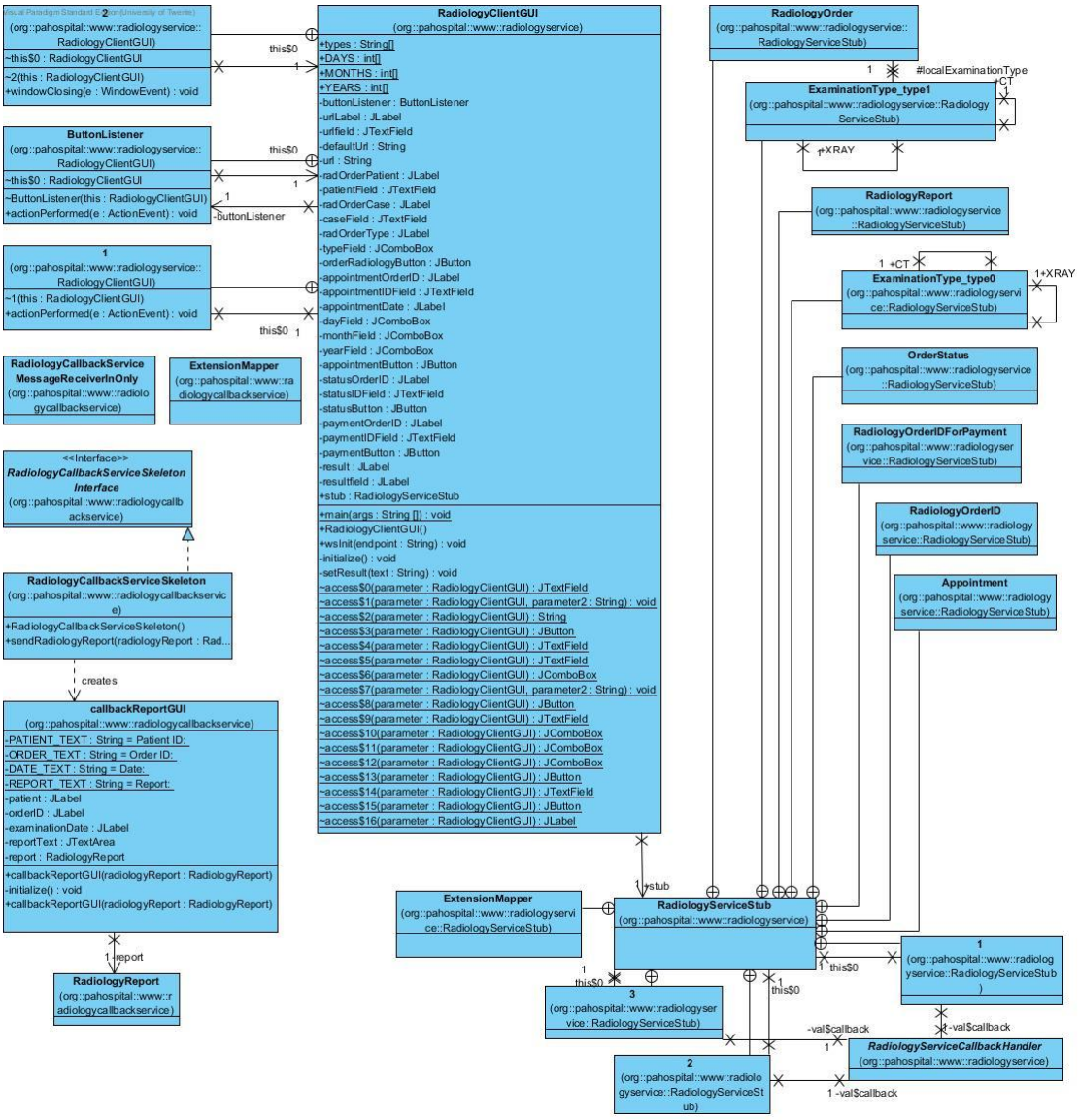


Figure 2 Client-side

2.3 System configuration

Both the client and the server, are also a server and client respectively since both have implemented a service and use the other's service. Therefore both have a web service implementation in the skeleton. They use each other's service implementation by using the stubs.

The generated files make sure that the application can use normal method calls, which are then transformed into SOAP messages. These SOAP messages are sent using HTTP. The HTTP messages encapsulate the SOAP messages.

Due to the nature of the implementation at the client-side, the client-side is not run in one, but in two JVMs. During the testing, the client service and server service were both in one JVM though, because Tomcat uses one JVM for all services, but in practice these would usually run on different machines.

It is not clear what is meant by the database management service, the RadiologyService could already be seen as a database management service since it only stores the data in the current description. In real world the application would be more difficult since it would also have to check whether the appointment is possible and give the opportunity to write a description in the report and much more. If the relation with another database management service is meant, the RadiologyService could use this service to store and retrieve the data and only take care of the logic (for planning the appointments). Of course it could also use other service for example to retrieve patient information.

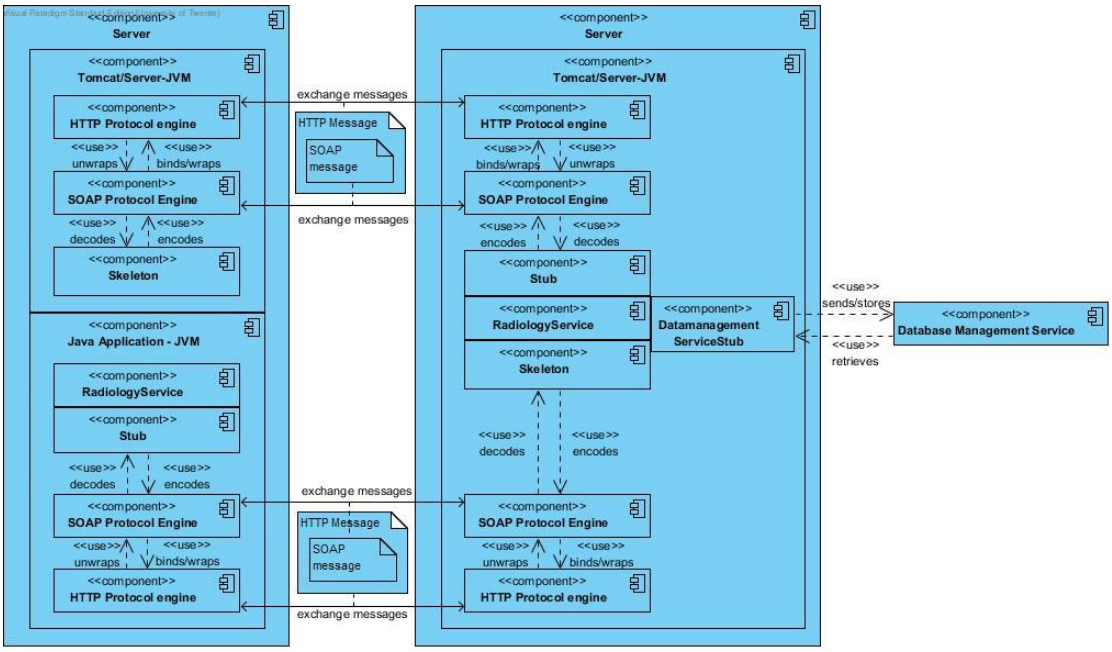


Figure 3 Illustration system configuration

2.4 Deployment and addressing

For deploying the service on Tomcat the tutorial was used. The build.xml in Eclipse was used to generate .aar-files. Then the server was started and then the axis administration panel was used to add/upload both services. These tasks could usually also be taken over by using a continuous integration solution like Jenkins. A correct deployment was verified by checking whether both services were available and also whether all operations were mentioned as available. Of course it could also be tested by using the description in the next question (2.5).

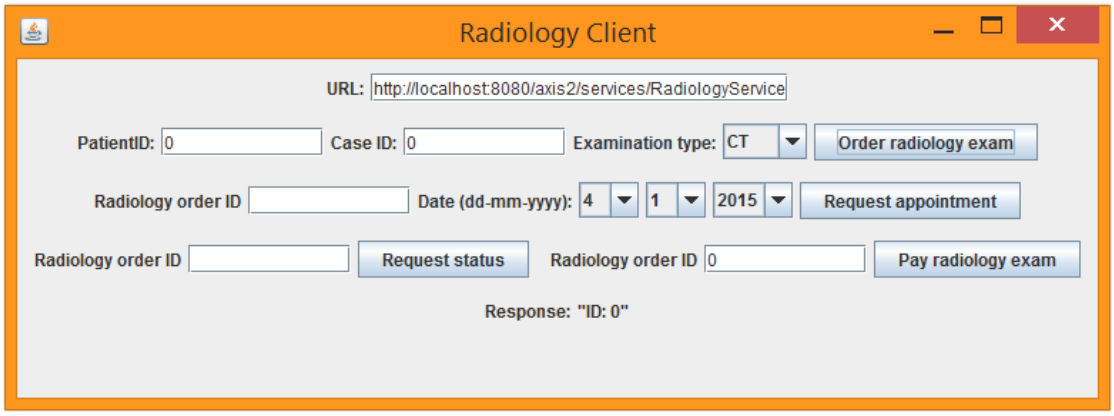
The client(GUI) has the url for the server standard mentioned in the GUI and this url can be edited to the correct url. This url could for example be found by going to the administration panel in axis2, but is also highly predictable: http://localhost:8080/axis2/services/<ServiceName>. At the server-side the url of the callback-service was hard-coded as well as the url for running it inside Eclipse. The server

tries to send the reports to both, hopefully succeeding at one of them. Usually obtaining the URL could better be done by using automated discovery mechanisms by looking into UDDI-registries but Axis2 does not have support for this [2].

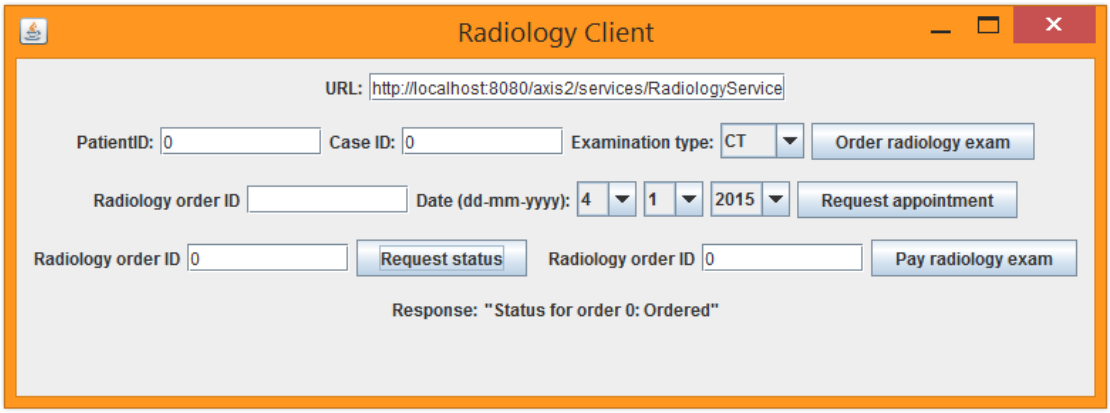
2.5 Testing

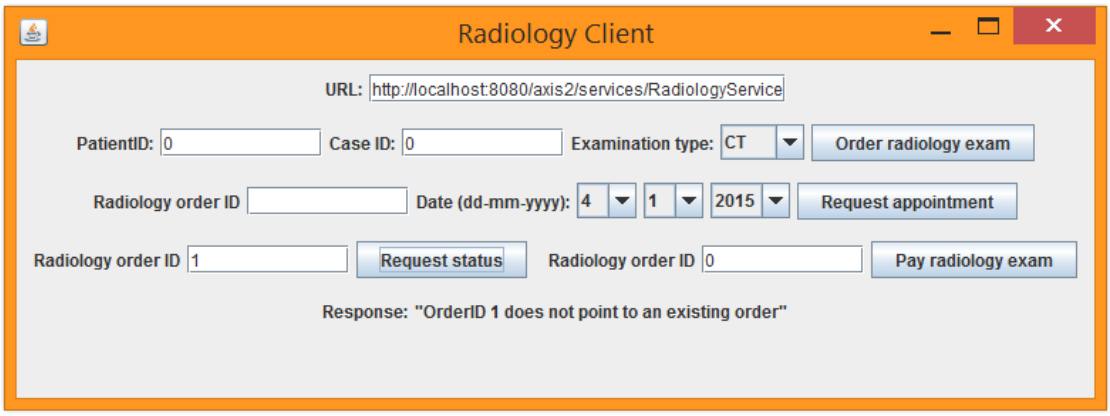
The implementation can be tested by using automated tests like done in assignment 1, but can also already be extensively tested using the GUI. This can be done by doing operations using the GUI and then checking the responses and status.

First try to order a radiology exam. This should generate a response with an ID, the first time ID 0 in this specific implementation.

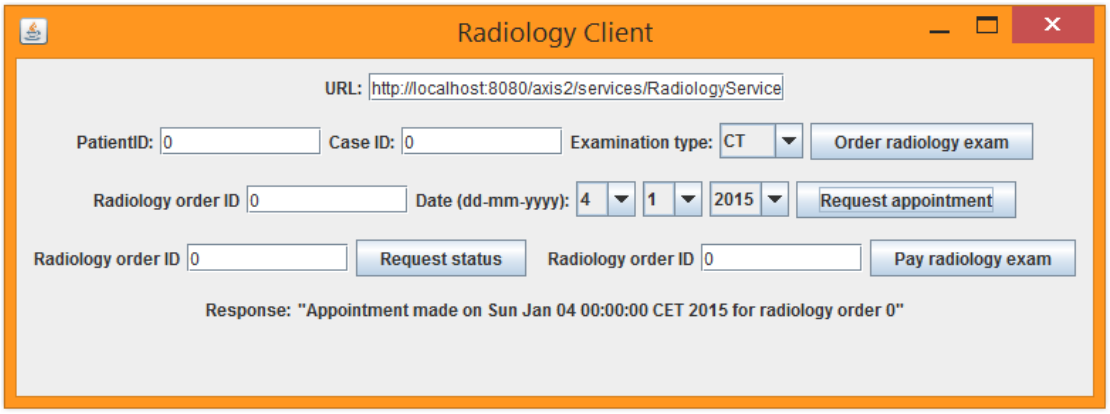


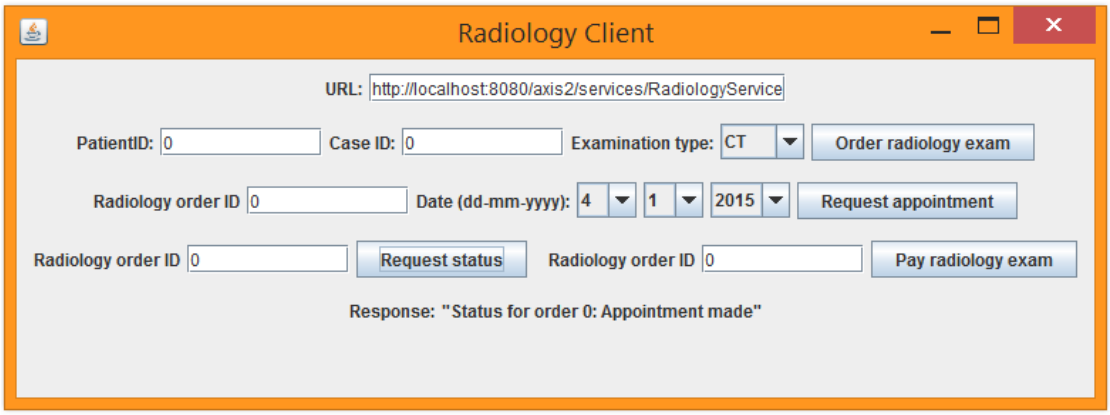
The status of OrderID 0 be that the examination has been ordered, but OrderID 1 should still be non-existent. This can be checked by requesting the status.

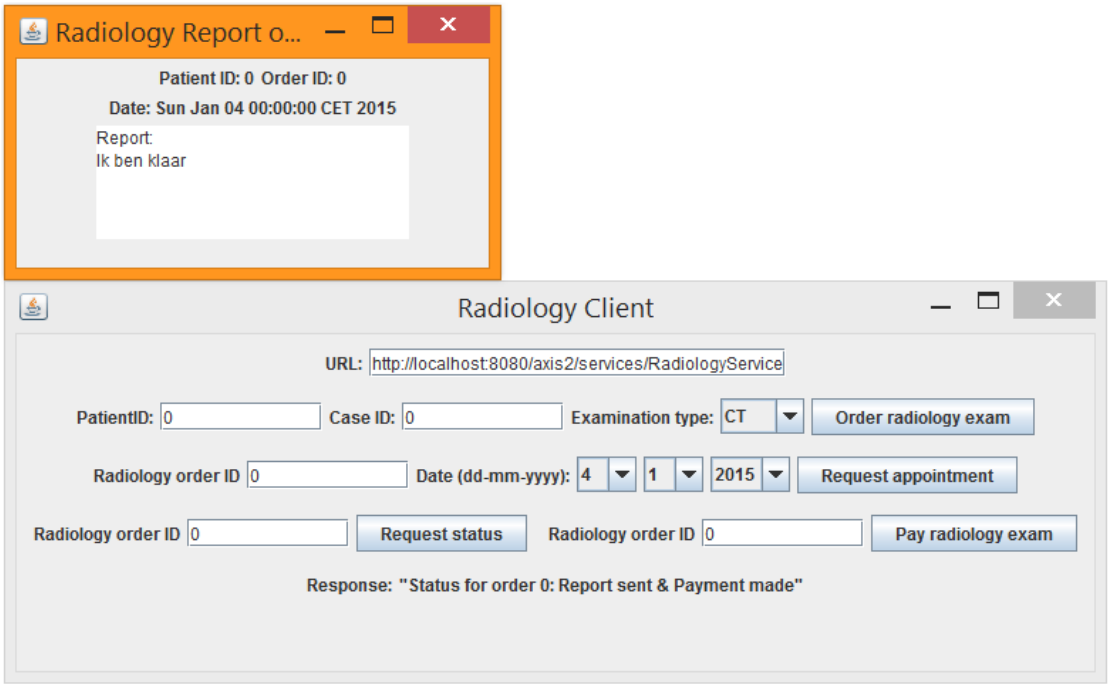




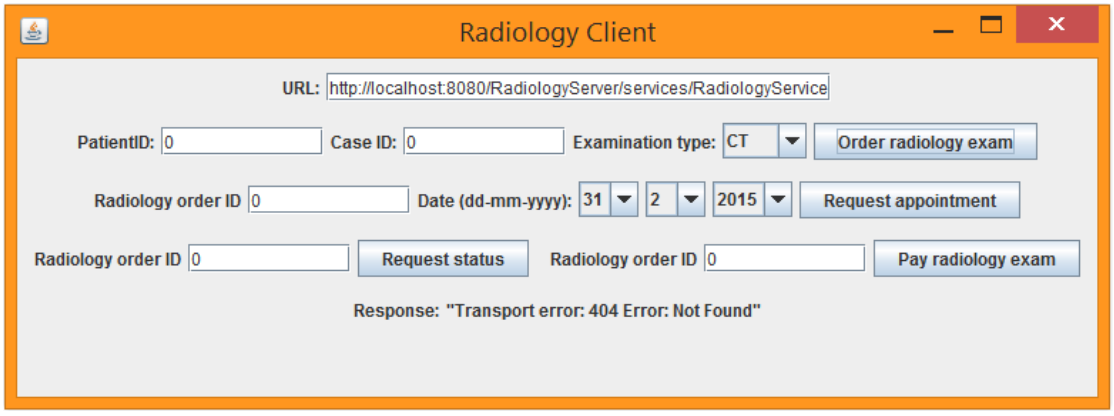
This way also the other operations can be tested (requesting the status has of course already tested that operation). The appointment can be requested, and after 6-12 seconds (randomly) the report will be send, so requesting the status beforehand the appointment should be made. Afterwards a report should have been sent, which can be checked by the status but also it should pop-up a new screen with the report (this is done by the RadiologyCallbackService). The pop-up can be used to check if the original PatientID is still the same and whether also the correct date had been saved. By paying the bill this should give the status report sent & payment made (as seen underneath).

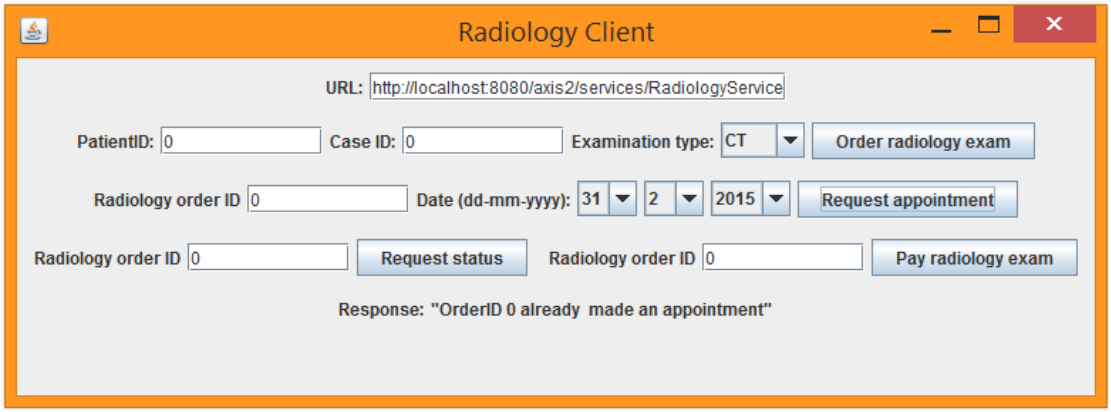






Of course also wrong things can happen. Exceptions should be thrown which arrive at the GUI. The GUI then prints the messages inside these exceptions. Two examples are shown here: pointing to the wrong url, and trying to make an appointment when one has already been made. Also these kind of tests were passed.





Many more test cases exist but it seems abundant to put all images here, test cases can be:

- Trying operations with a non-existing order ID (or even an order ID which is a normal String and not an int)

- Testing with more data

- Trying a wrong date (this is currently partially possible, 31-02-2015 will be translated to 03-03-2015)

References

[1] http://jdatepicker.org

[2] https://axis.apache.org/axis/java/client-side-axis.html#Dynamically_Discovering_and_Binding_to_a_Web_Service