Java-basierten Web Service erstellen

Einleitung

Dieses Tutorial zeigt, wie der Web Service ***AutoKauf*** erstellt werden kann. Dieser Web Service bildet die Grundlage für alle anderen Tutorials. Deshalb ist es sinnvoll, dieses Tutorial als erstes durchzuarbeiten. Wenn möglich, sollte dies im Team erfolgen.

Inhaltsverzeichnis

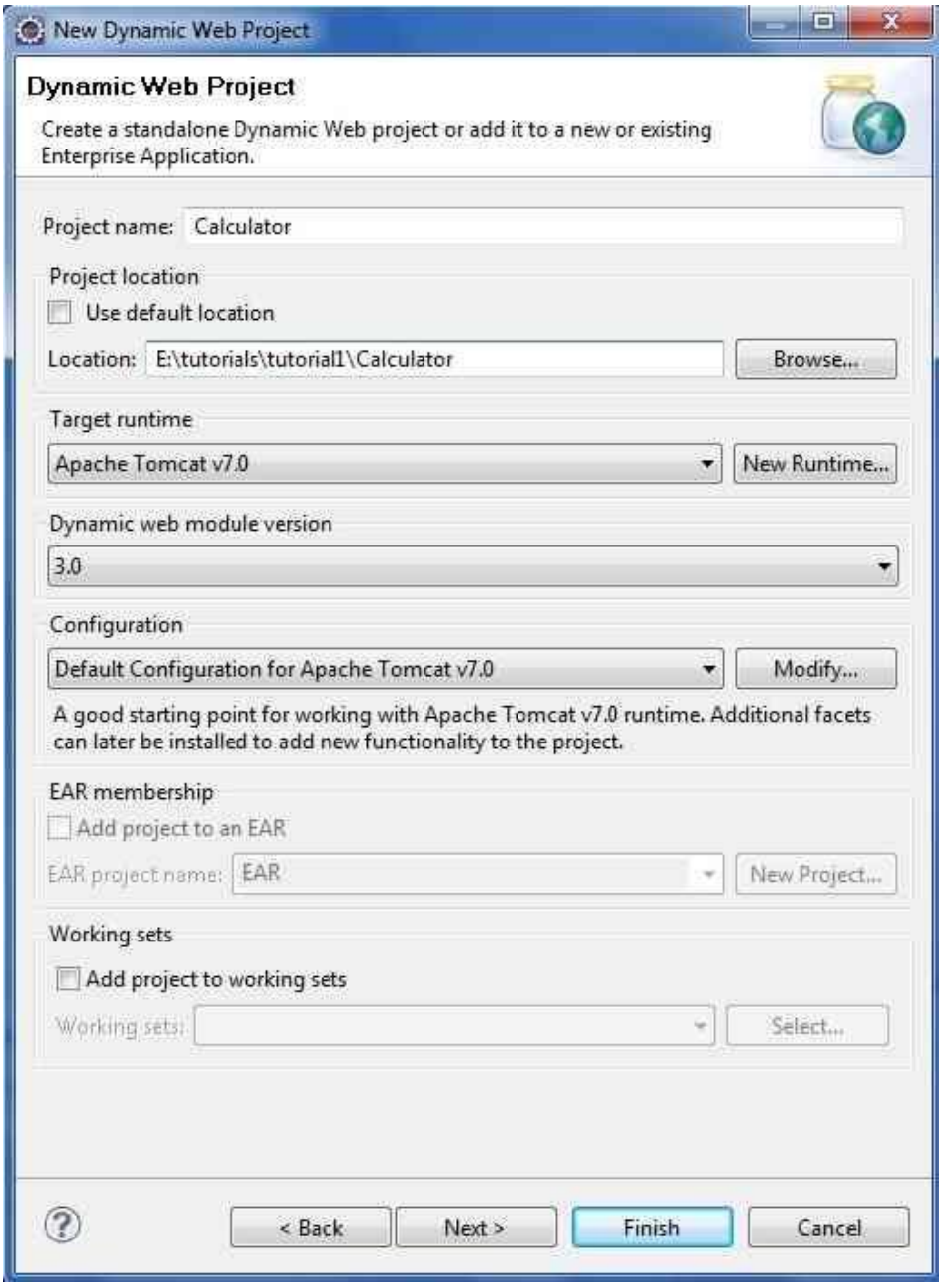
1 Beispiel Web Service

Um ein Gefühl für Web Services zu bekommen, erstellen wir bottom-up einen einfachen Web Service ***Calculator***. Bottom-up bedeutet, dass zuerst die Klasse(n) mit der Funktionalität erstellt und daraus die WSDL-Datei generiert wird.

Der Service soll zwei übermittelte Zahlen addieren und als Antwort das Ergebnis der Addition zurückliefern. Im Interesse der Teammitglieder sollte dieses Beispielprojekt durchgeführt werden, da es in den anderen Tutorials ebenfalls benötigt wird.

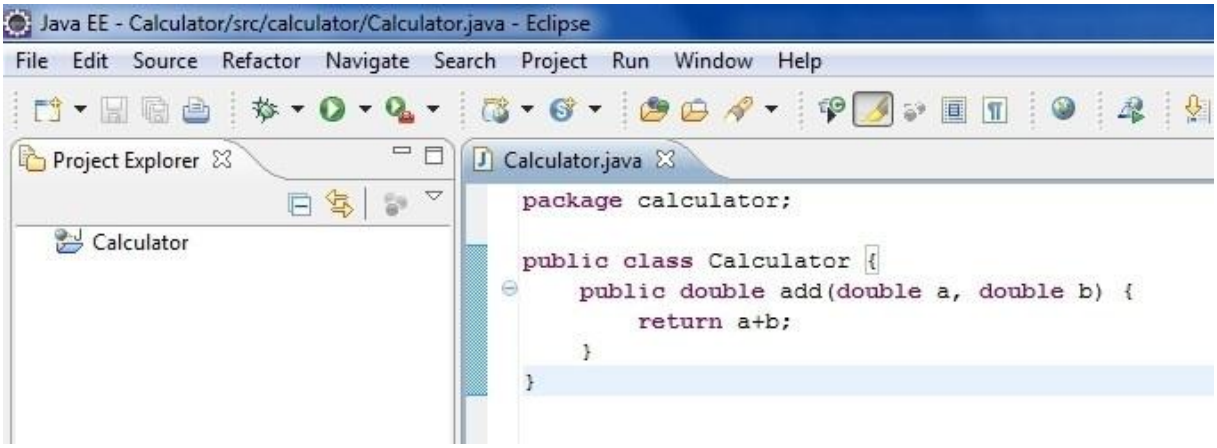
1.1 Calculator Web Service erstellen

Nach dem öffnen von Eclipse wird ein neues **Dynamic Web Project *Calculator*** erstellt.



Nun erstellen wir eine Klasse `Calculator`, die die Funktionalität bereitstellt – also die Klasse, in der die Methode zur Addition der Zahlen implementiert wird.

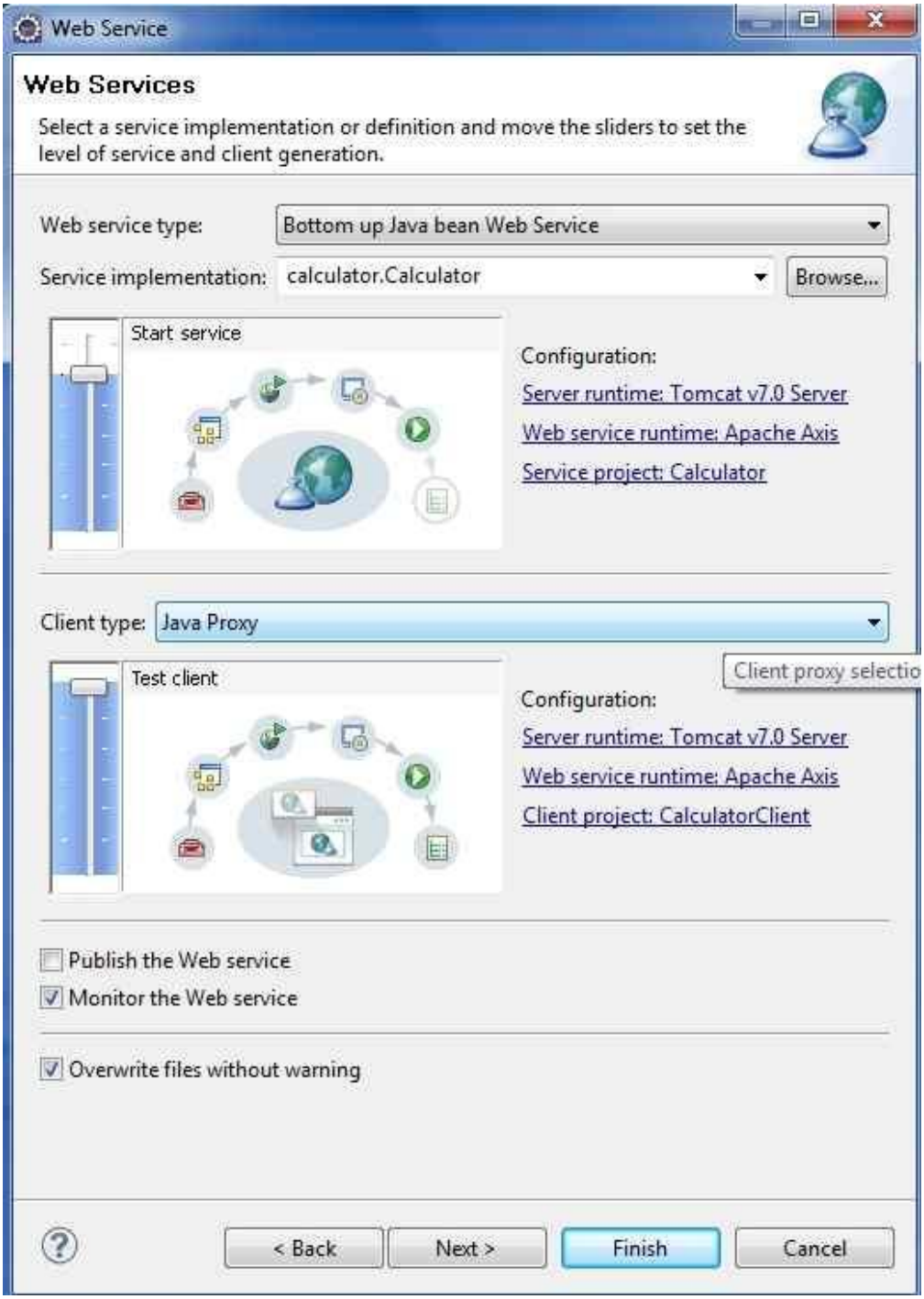
Nun fügen wir die Funktion `add` hinzu.



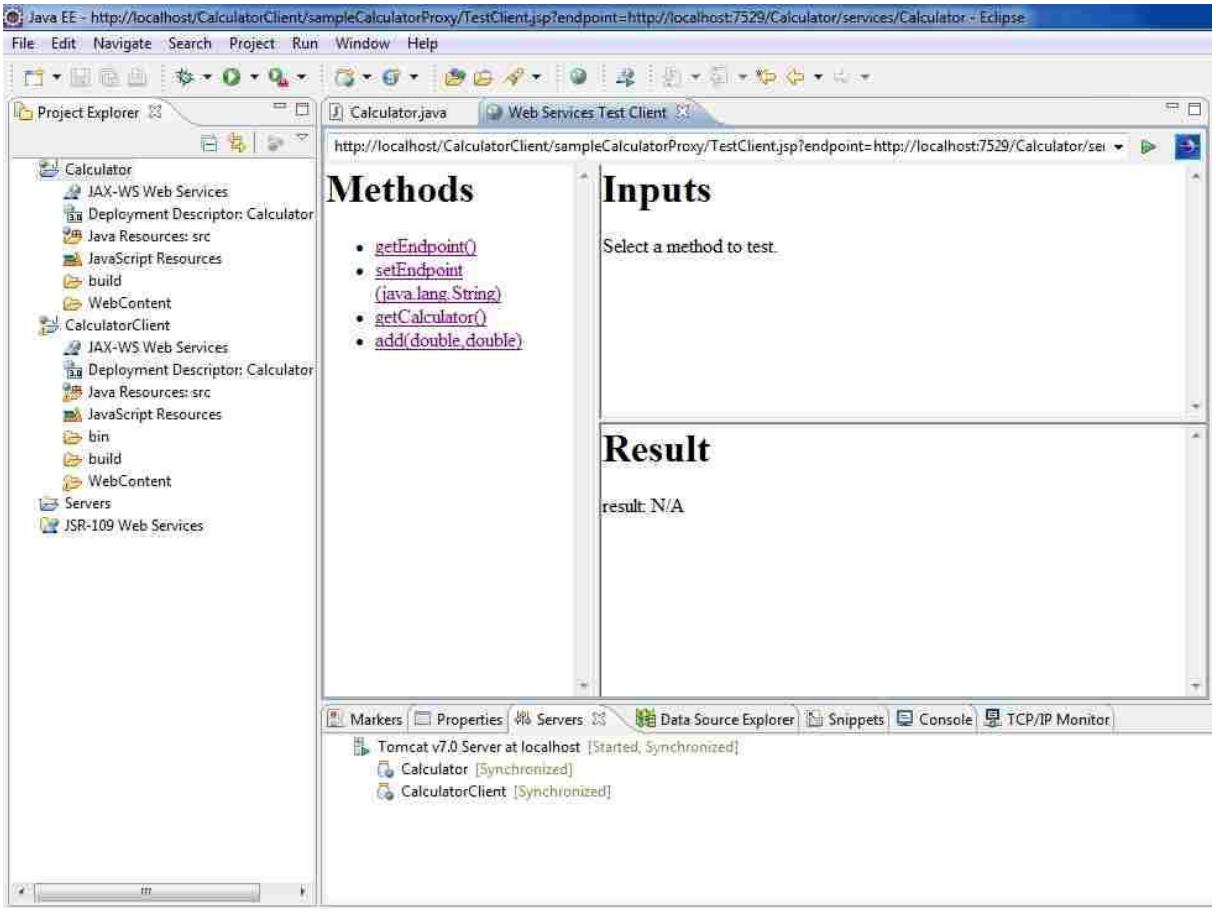
Die Funktion `add` soll nun als Web Service Operation zu Verfügung gestellt werden.

Mit Rechtsklick auf das Projekt ***Calculator:*** **New -> Other... -> Web Services -> Web Service** öffnen wir den Wizard für die automatische Generierung des Web Service.

An dieser Stelle können wir auch einen Client zum Testen des Service erstellen und die Kommunikation zwischen dem Web Service und dem Web Service Client überwachen lassen (Monitor).



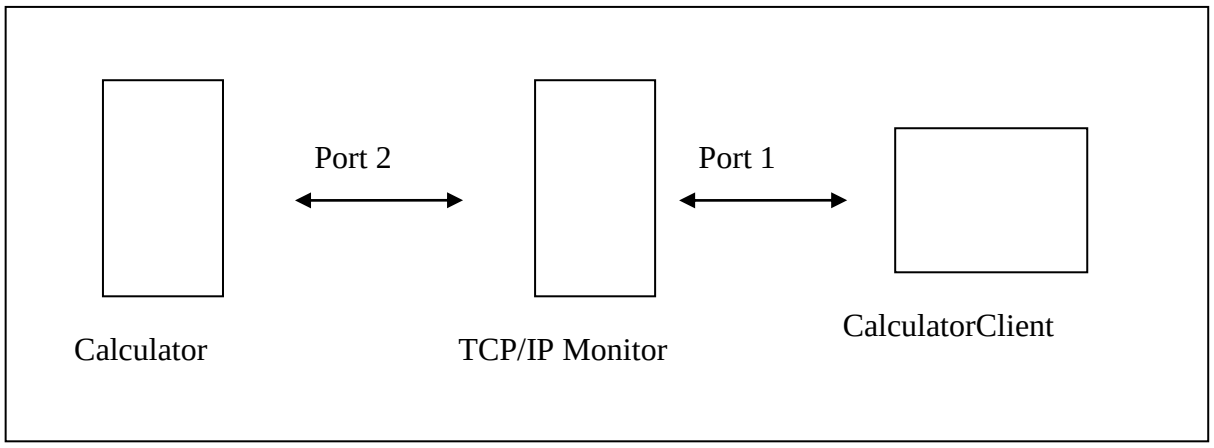
Das Ergebnis sollte das folgende sein.



In der Server-Ansicht ist nun zum einen das Tomcat-Plugin gestartet und zum anderen sind die beiden Web-Kontexte ***Calculator*** und ***CalculatorClient*** hinzugefügt wurden.

1.2 TCP/IP-Monitor

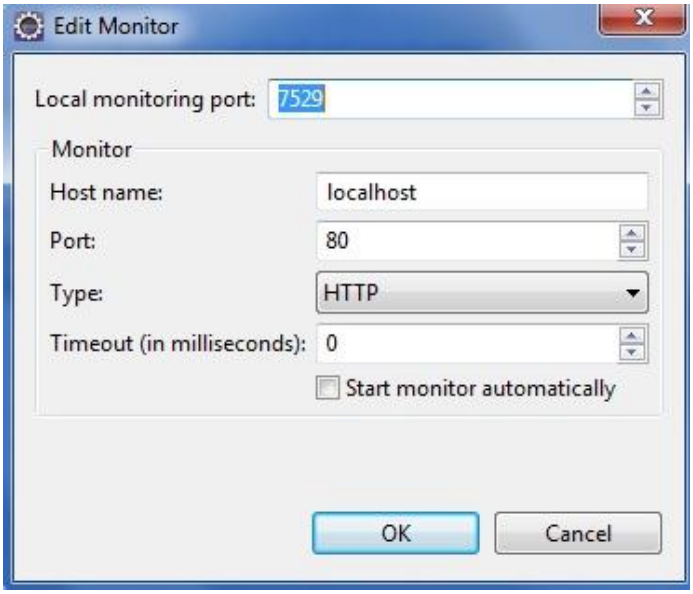
Was genau bei der Kommunikation zwischen Web Service Client (hier ***CalculatorClient***) und Web Server (hier Eclipse-Tomcat-Plugin) geschieht, kann mit dem TCP/IP-Monitor überwacht werden. Dabei fungiert der Monitor als ein Proxy-Server.



Es können also Client Anfragen auf dem einen Port (z.B. localhost:7529) gestellt werden und der Proxy mappt die Anfragen dann auf den Zielport (z.B. localhost:80). Die dabei verarbeiteten Protokollnachrichten (vom Typ HTTP) werden angezeigt.

Da bei der Erstellung des Web Service ***Calculator*** auch das Häkchen **Monitor** mit ausgewählt wurde, ist eine entsprechende TCP/IP-Monitor Konfiguration bereits erstellt wurden.

Die erstellte Konfiguration kann über **Window -> Preferences -> Run/Debug -> TCP/IP Monitor -> Edit** eingesehen und bearbeitet werden.



Der **Local monitoring port: 7529** ist zufällig gewählt und hat sonst keine weitere Bedeutung. Werden die SOAP-Nachrichten nicht (korrekt) angezeigt, empfiehlt sich die Einstellung **Type: TCP/IP**.

Um den Monitor in späteren Tests des Web Service ***Calculator*** wieder zu verwenden, sind zwei Dinge zu beachten.

Der Monitor in der Liste **TCP/IP Monitors** muss gestartet sein.
Der Parameter `endpoint` muss beim Aufruf des Testclient angegeben werden.
http://localhost/CalculatorClient/sampleCalculatorProxy/TestClient.jsp?endpoint=http://localhost:7529/Calculator/services/Calculator

Die Ansicht **TCP/IP-Monitor** selbst kann mit **Window -> Show View -> Other… -> Debug -> TCP/IP-Monitor** angezeigt werden.

An dieser Stelle sei noch einmal darauf hingewiesen, dass die Kommunikation zwischen dem SOAP-Client-Proxy und dem Web Service ***Calculator*** angezeigt wird und nicht die Kommunikation zwischen der ***TestClient.jsp*** und dem Web Service. Mehr Informationen gibt es dazu im Tutorial ***JSP Web Service Client erstellen***.

1.3 Calculator Web Service testen

1.3.1 Testen mit TestClient.jsp

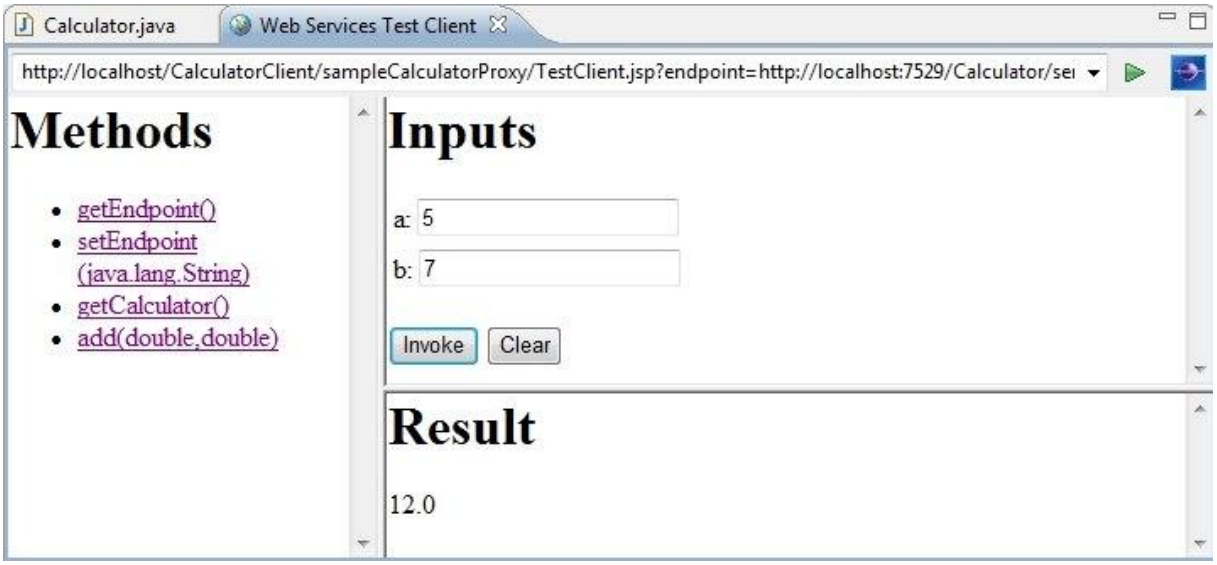
Eine Möglichkeit den Web Service zu testen haben wir ja bereits kennen gelernt – die ***TestClient.jsp***.

Die ***TestClient.jsp*** ist eine der JSPs, die im Rahmen des automatisch erstellten ***CalculatorClient*** erstellt wurde. Wie jede JSP kann auch diese direkt im Browser aufgerufen werden.

http://localhost/CalculatorClient/sampleCalculatorProxy/TestClient.jsp

Wenn ein Monitoring stattfinden soll, muss entsprechen der Endpoint hinzugefügt werden.

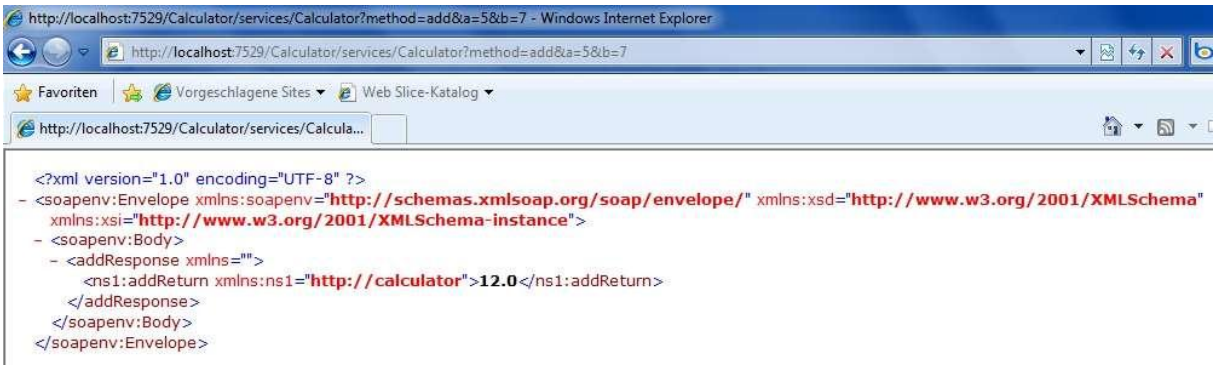
Das Ergebnis haben wir uns ebenfalls schon angesehen.



1.3.2 Direkter Web Service Aufruf (Browser)

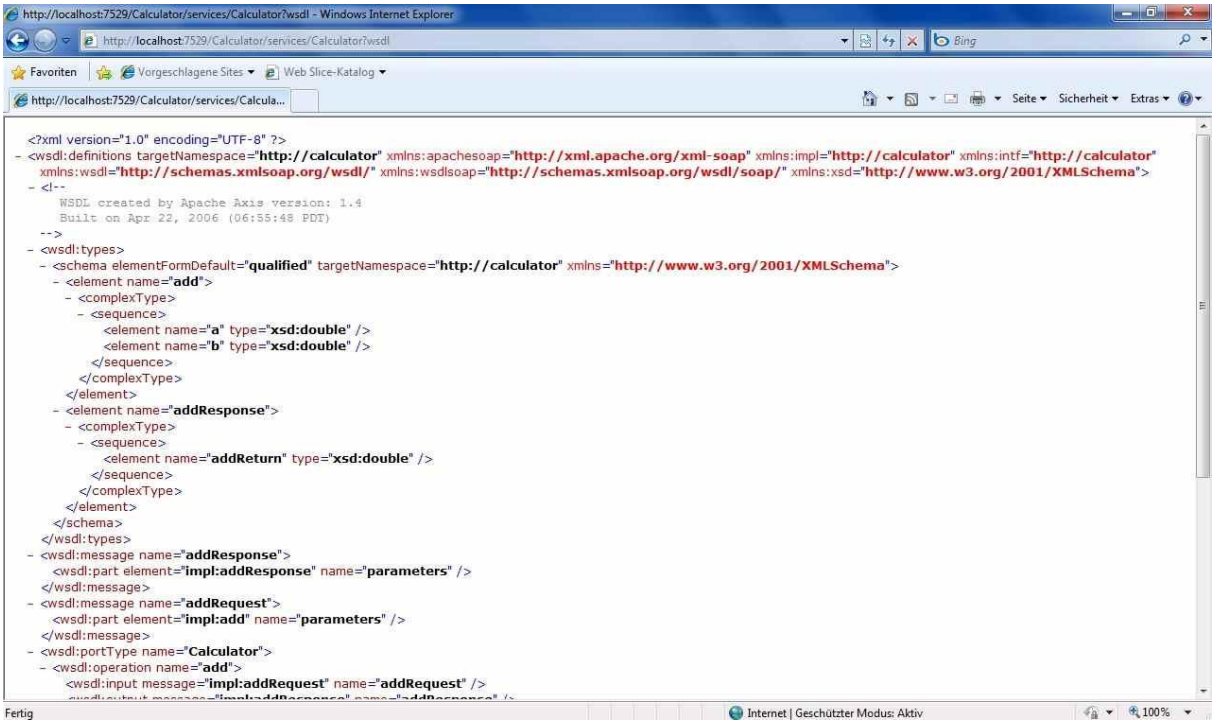
Außer dem Test mit dem automatisch erstellten Web Service Client kann der Web Service auch direkt mit dem Browser getestet werden (ohne Client-Projekt – also ohne SOAP-Client-Proxy).

Hierfür geben wir die URL für den Web Service ***Calculator*** und die notwendigen Parameter `method`, `a` und `b`, sowie die entsprechenden Werte `5` und `7` an.



Auch hier können wir unter Angabe des Monitoring-Ports die Kommunikation überwachen.

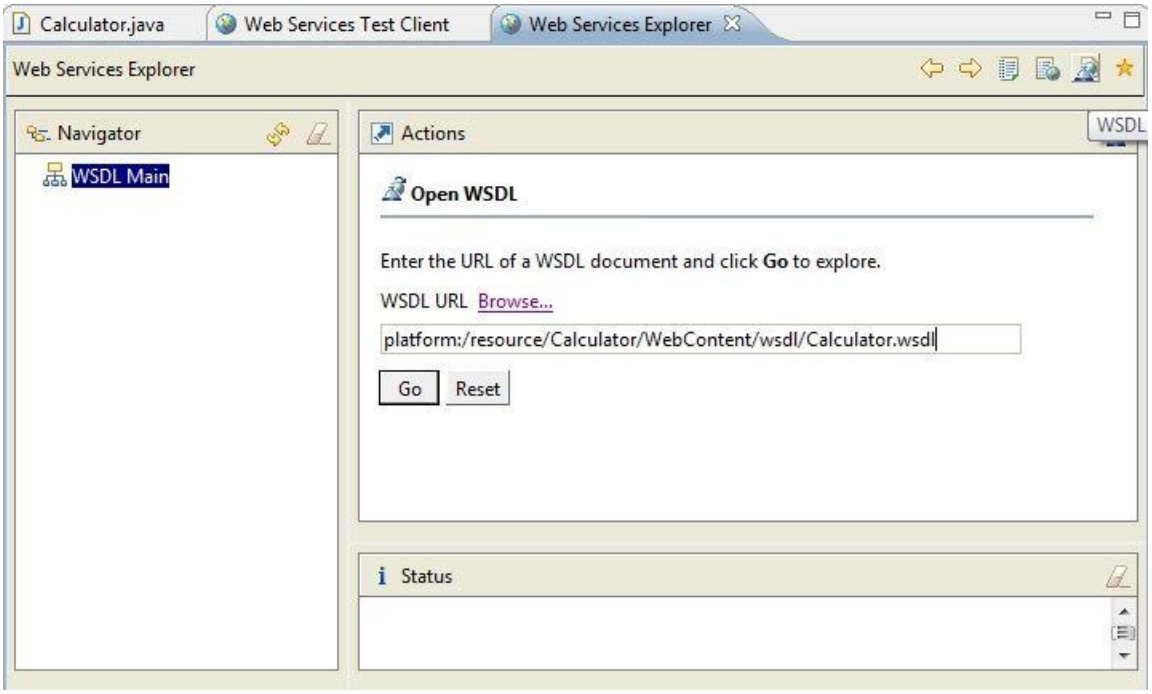
Im Browser kann man sich auch die WSDL-Beschreibung des Services ansehen.

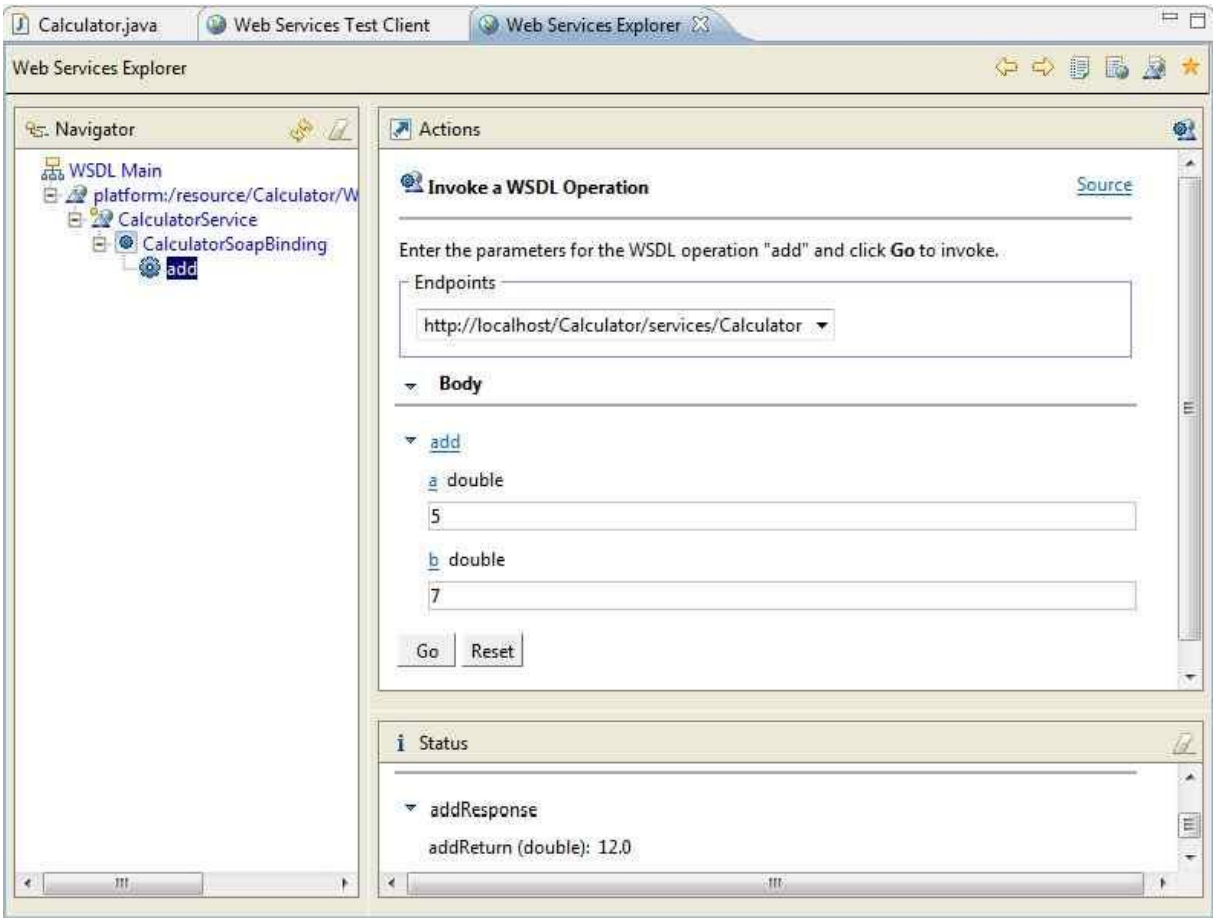


1.3.3 Web Service Explorer

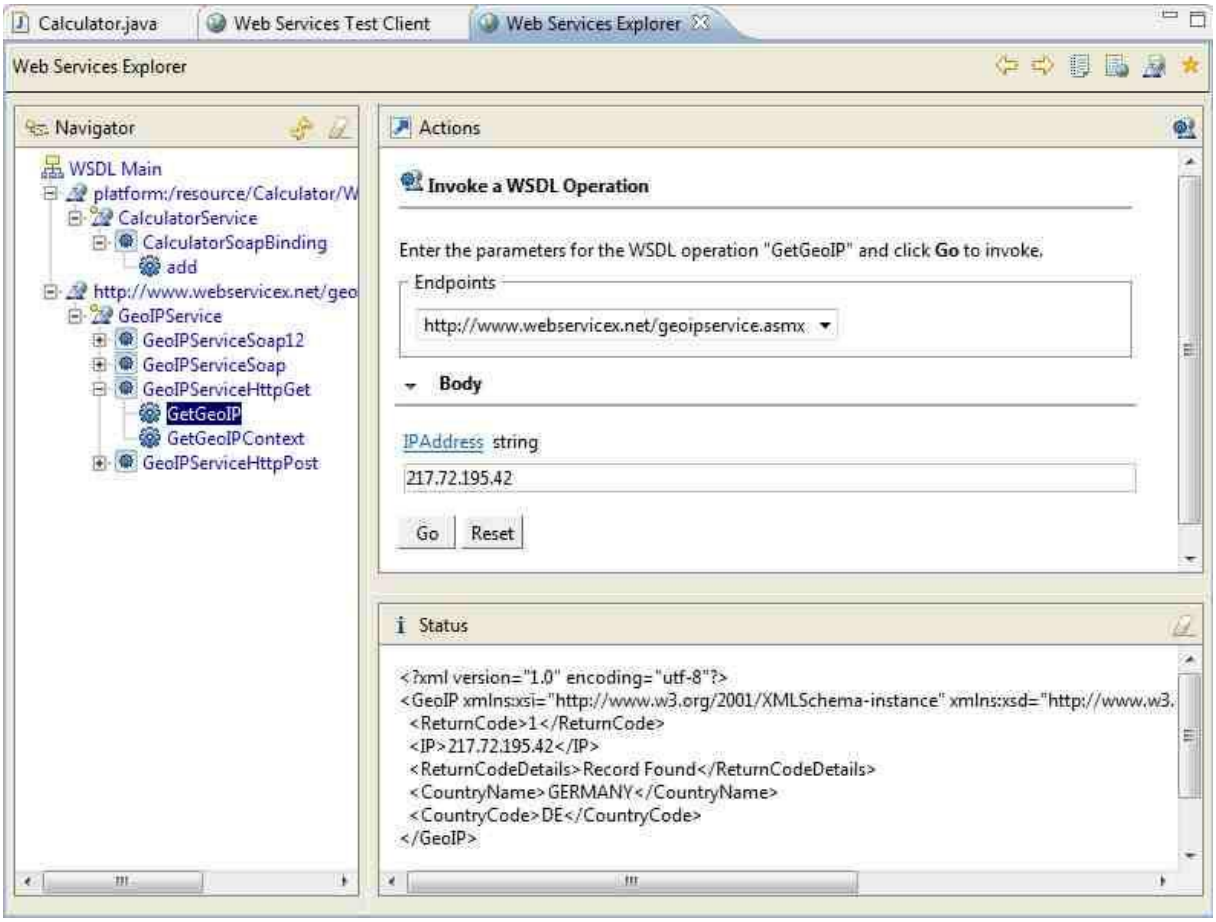
Mit dem Web Service Explorer können Web Services über ihre WSDL-Beschreibung getestet werden. Voraussetzung ist, dass der Ort der WSDL-Beschreibung bekannt ist.

Mit **Run -> Launch the Web Service Explorer** wird der Web Service Explorer gestartet. Nach dem Start kann mit Klick auf das **WSDL-Page**-Icon (oben rechts) und Klick auf **WSDL-Main** die WSDL-Beschreibung (lokale Datei oder Web URL) ausgewählt werden.





Statt der WSDL-Beschreibung des Web Service ***Calculator*** könnte man auch eine WSDL-Beschreibung eines im Internet frei verfügbaren Service nehmen (z. B. http://www.webservicex.net/geoipservice.asmx?WSDL) und würde dann folgendes Ergebnis erhalten:



Lassen wir's mal dabei. Aber wozu eigentlich das ganze mit den WSDL-Files? Wie man sieht, kann mit der WSDL-Beschreibung festgestellt werden, wie auf einen Service zugegriffen werden muss.

Besonders interessant ist, dass mit Hilfe einer WSDL-Datei ein Web-Service-Grundgerüst (Skeleton) erstellt werden kann (zum Beispiel in Java). Das Vereinfacht das Programmieren – vorausgesetzt, die WSDL-Beschreibung existiert bereits.

2 Autokauf Web Service

Das Ziel dieses Kapitels ist die Erstellung eines Web Service, über den Autos gekauft (und verkauft) werden können. Dafür werden drei Methoden implementiert: `alleAutosAnzeigen` (liefert eine Liste aller verfügbaren Autos), `kaufeAuto` (setzt das Attribut `gekauft` auf `true`) und `verkaufeAuto` (setzt das Attribut `gekauft` auf `false`). Ein Auto hat demnach folgende Attribute: `autoID` (muss für jedes Auto eindeutig sein), `farbe`, `anzahlSitze` und `gekauft`. Die genaue Implementierung erfolgt weiter unten im Tutorial. Man erkennt hier schon, dass alles recht einfach gehalten wird – uns geht es ja mehr um das Prinzip.

Wie am Ende des letzten Kapitels angedeutet, werden wir nun den umgekehrten Weg gehen, also top-down. D.h. wir erstellen das Java Webservice-Grundgerüst anhand der WSDL-Beschreibung. Das ist ein realistisches Szenario, da oftmals die Schnittstellenbeschreibung bekannt ist und implementiert werden muss.

Die WSDL-Datei könnte z.B. vom Auftraggeber vorgegeben sein. In dem Fall ist sie es nicht und wir müssen sie selbst erstellen. ‚Halt', wird der eine oder andere jetzt denken, ‚bis ich die 100 Zeilen XML-Code geschrieben habe, hab ich auch die paar Klassen implementiert'. Das ist richtig. Die bottom-up Variante ist meist schneller. Dafür hat man nur geringen Einfluss auf die erstellte WSDL-Datei. Diese Datei ist jedoch die Grundlage für alle anderen Gruppenmitglieder und sollte deshalb übersichtlich sein. D.h. zum einen sollte der Quellcode (XML-Code) korrekt und eindeutig sein und zum anderen sollten grafische Editoren die gewünschte Web Service Struktur anschaulich darstellen. Und gerade bei dem letzten Punkt, der korrekten Darstellung in grafischen WSDL-Editoren, scheitern die automatisch generierten WSDL-Dateien häufig.

2.1 Erstellen der WSDL-Datei

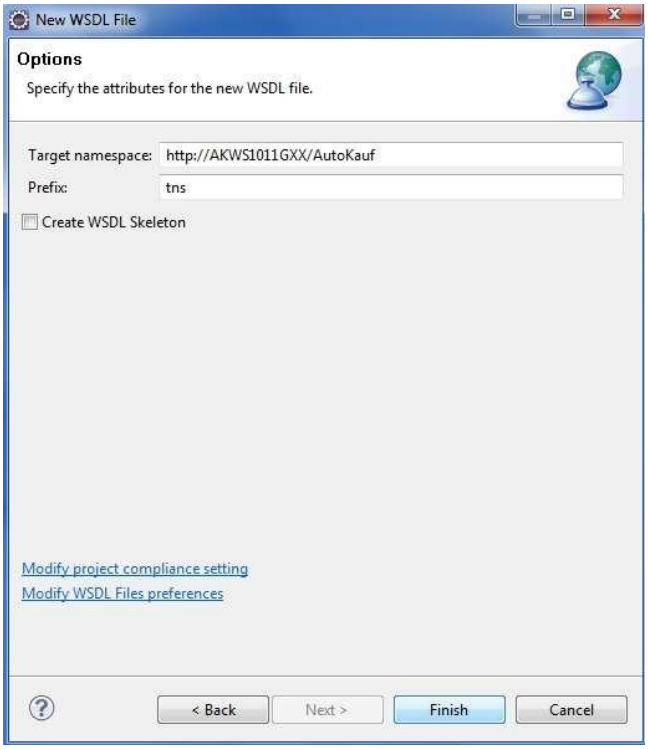
Wenn die WSDL-Datei am Ende in einem grafischen WSDL-Editor anschaulich dargestellt werden soll, liegt es nahe diese Datei auch mit einem grafischen WSDL-Editor zu erstellen.

In Eclipse-WTP gibt es dafür den **WSDL-Editor**. In der hier verwendeten Version ist dieser zwar recht instabil, die Verwendung bietet sich jedoch an, da der Editor besonders gut auf den integrierten Wizard zum erstellen des Java Web Service Skeletts abgestimmt ist. Im Bezug auf die Stabilität bedeutet das: öfter mal speichern!

Alternativ kann auch ein anderer (grafischer) WSDL- bzw. XML-Editor verwendet werden, z.B. XMLSpy. In XMLSpy ist ebenfalls ein grafischer WSDL-Editor enthalten. Wie auch immer, dieses Tutorial beschreibt die Vorgehensweise mit dem in Eclipse WTP integrierten WSDL-Editor.

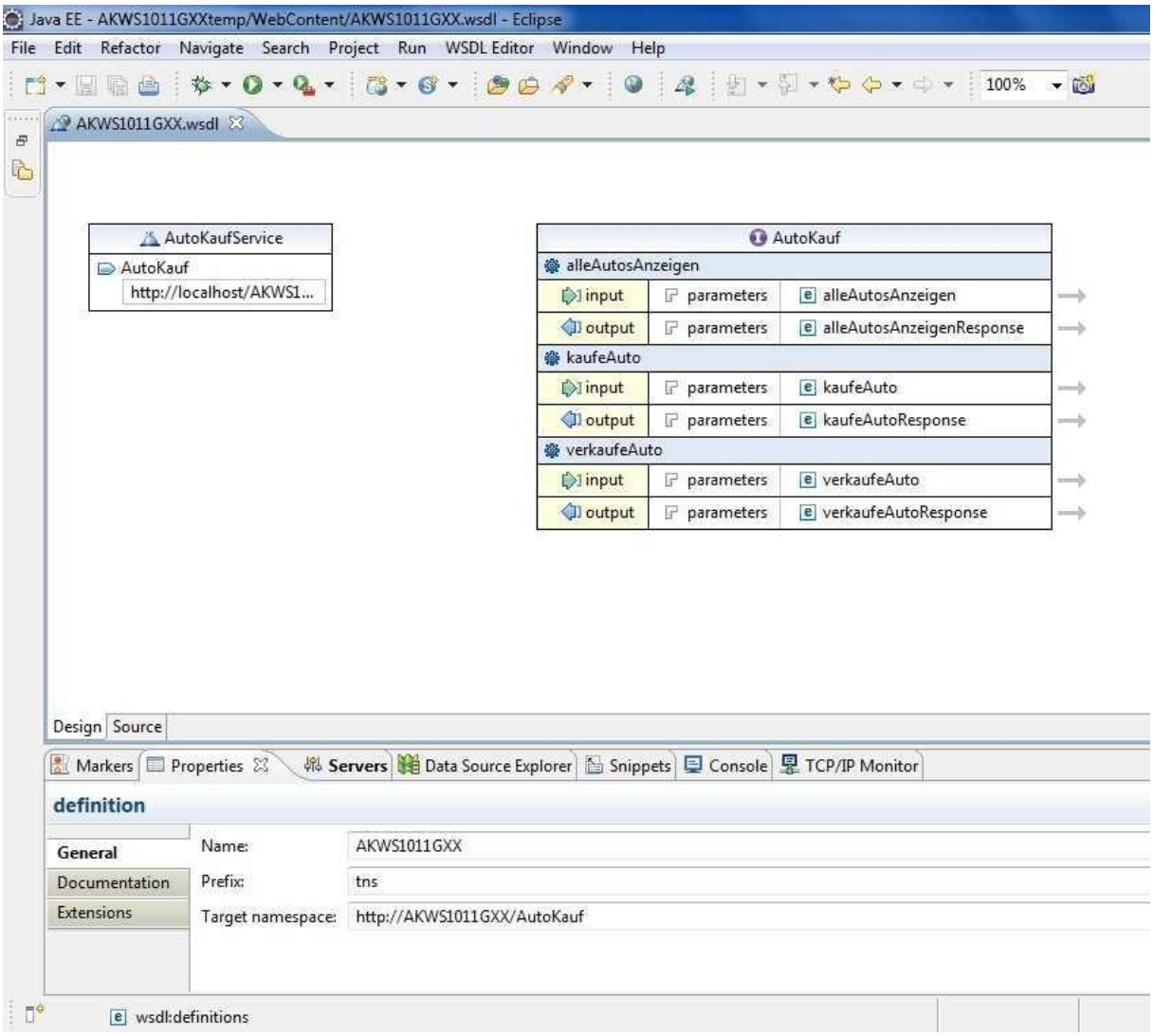
In Eclipse erstellen wir zunächst ein neues **Dynamic Web Project** namens ***AKWS1011GXXtemp***. Aus Bugtechnischen Gründen müssen wir zunächst ein temporäres Projekt erstellen – Erläuterungen dazu erfolgen weiter unten im Tutorial.

Mit Rechtsklick auf den Ordner `WebContent` **New -> Other -> Web Services -> WSDL** erstellen wir eine neue WSDL-Datei ***AKWS1011GXX.wsdl***



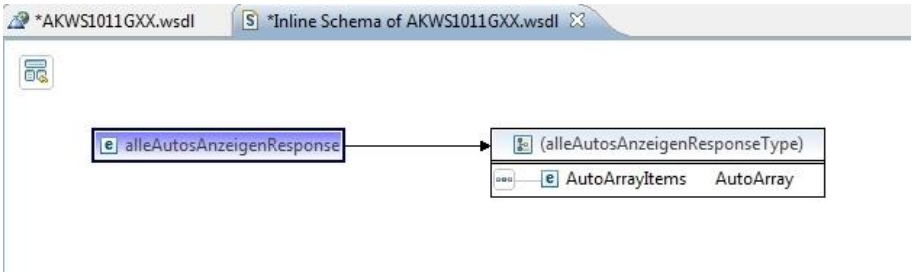
Mit Rechtsklick in der leeren Design-Ansicht des Editors erstellen wir nun einen neuen **Service** `AutoKaufService` und einen zugehörigen **Port** `AutoKauf` mit der Adresse http://localhost/AKWS1011GXX/services/AutoKauf. Unter dieser URL können wir den Service später erreichen.

Als nächstes erstellen wir einen neuen **PortType** `AutoKauf` und drei neue **Operation**s (Methoden) `alleAutosAnzeigen`, `kaufeAuto` und `verkaufeAuto`. Die Input- und Output-Elemente werden automatisch generiert.

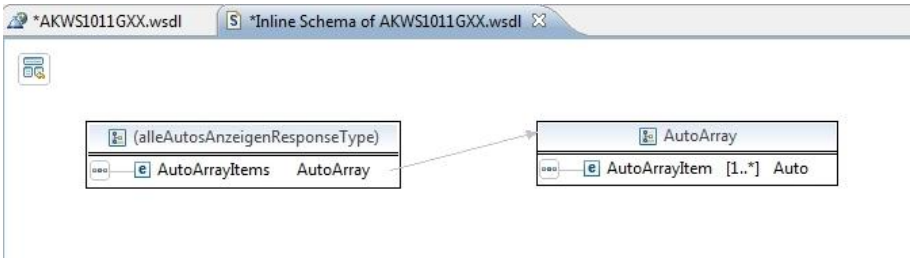


Mit Klick auf den Pfeil neben einem Element können wir das Element im Inline-XML-Schema näher spezifizieren.

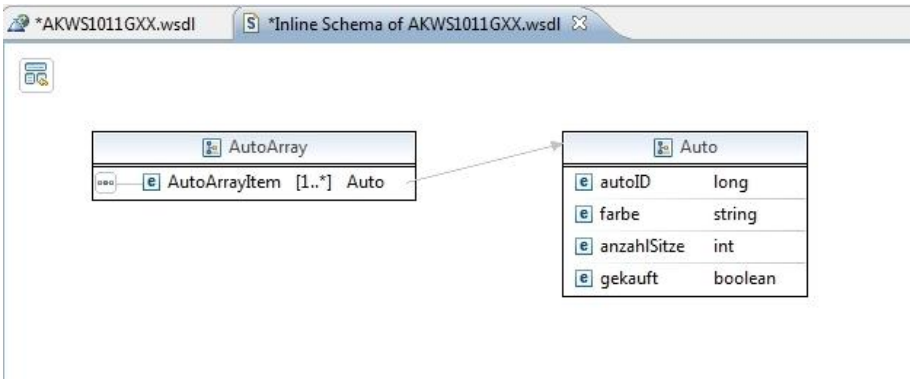
Das interessanteste ist das Outputelement `alleAutosAnzeigenResponse`. Dieses besitzt ein Unterelement `AutoArrayItems` . Als dessen Typ legen wir einen neuen Komplexen Datentyp `AutoArray` an.



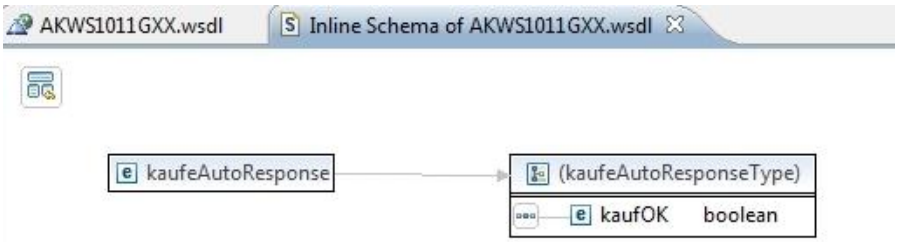
Mit Doppelklick auf die Typbezeichnung (Kopfzeile) können wir den Typ `AutoArray` spezifizieren. Ein `AutoArray` besteht demnach aus dem Element `AutoArrayItem` vom neuen Komplexen Typ `Auto`. Mit **Set Multiplicity -> 1..*** signalisieren wir, dass es sich um ein Array handelt.

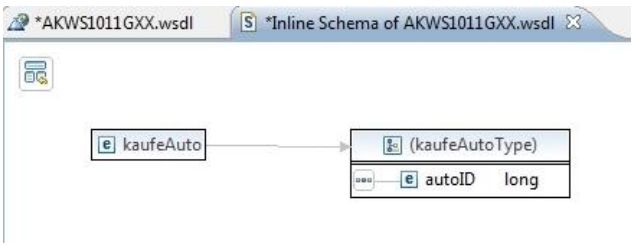


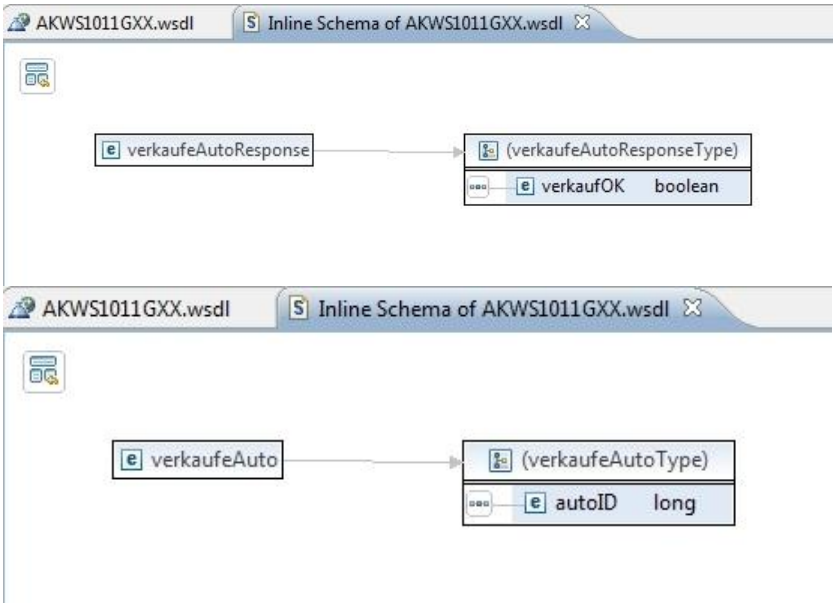
Der Typ `Auto` besitzt die Elemente `autoID` (`xsd:long`), `farbe` (`xsd:string`), `anzahlSitze` (`xsd:int`) und `gekauft` (`xsd:boolean`).



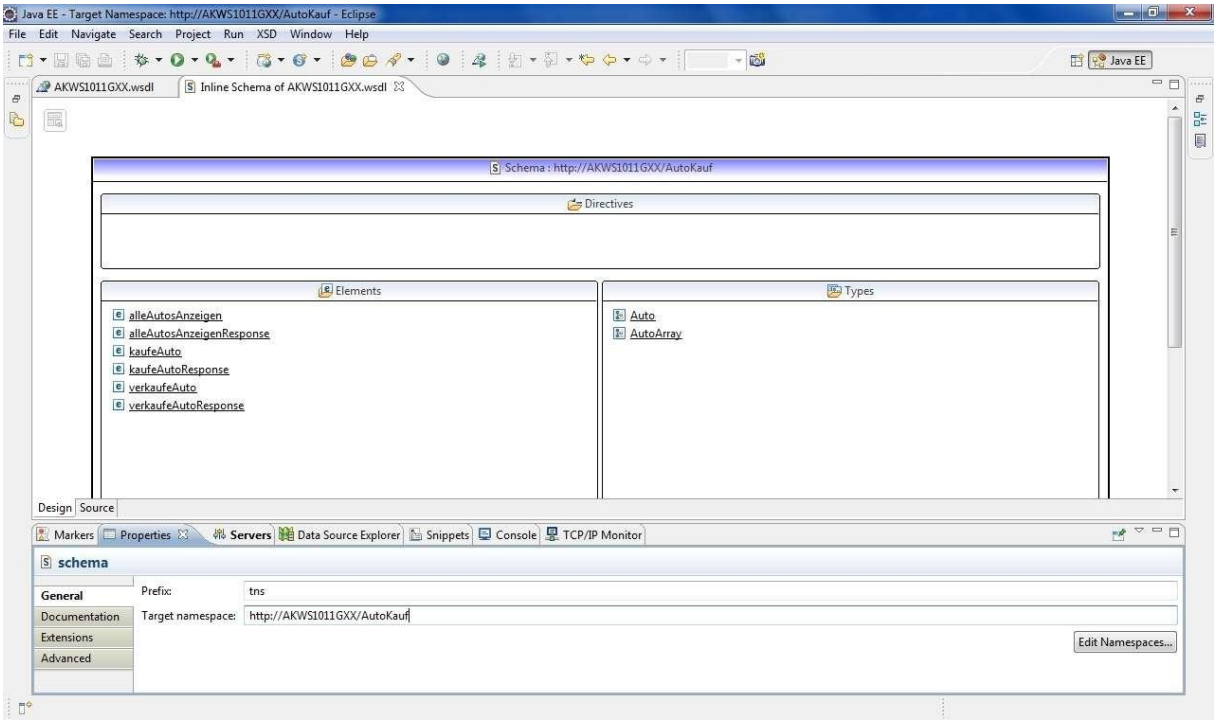
Nun müssen wir noch die anderen Übergabe- bzw. Rückgabe-Elemente spezifizieren.







In der Übersicht der neu erstellten Elemente und Typen, sehen wir auch den aktuellen Target-Namespace (tns) für das Inline-Schema.



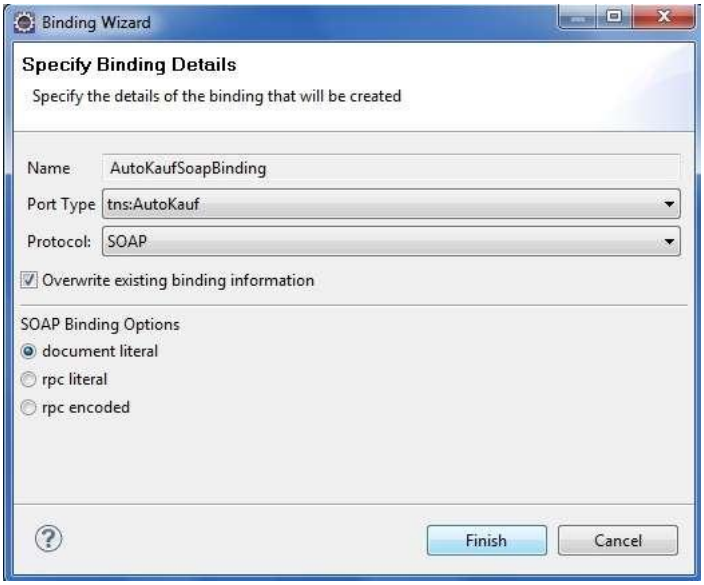
Der tns sorgt dafür, dass die von uns erstellten Elemente und Datentypen weltweit (zumindest dokumentenweit) eindeutig sind. Im Moment sind der tns des Inline-Schemas und der tns des WSDL-Dokumentes noch identisch (`http://AKWS1011GXX/AutoKauf`). Bei der späteren Klassengeneration würden so alle Klassen in einem Package erstellt werden (`AKWS1011GXX.AutoKauf`). Das wird recht unübersichtlich. Den tns in der Design-Ansicht (also an dieser Stelle) zu ändern funktioniert nicht. Deshalb müssen wir den tns des Inline-Schemas später direkt im XML-Quellcode des WSDL-Dokumentes ändern.

Vorerst können wir die Inline-Schema Ansicht speichern und schließen.

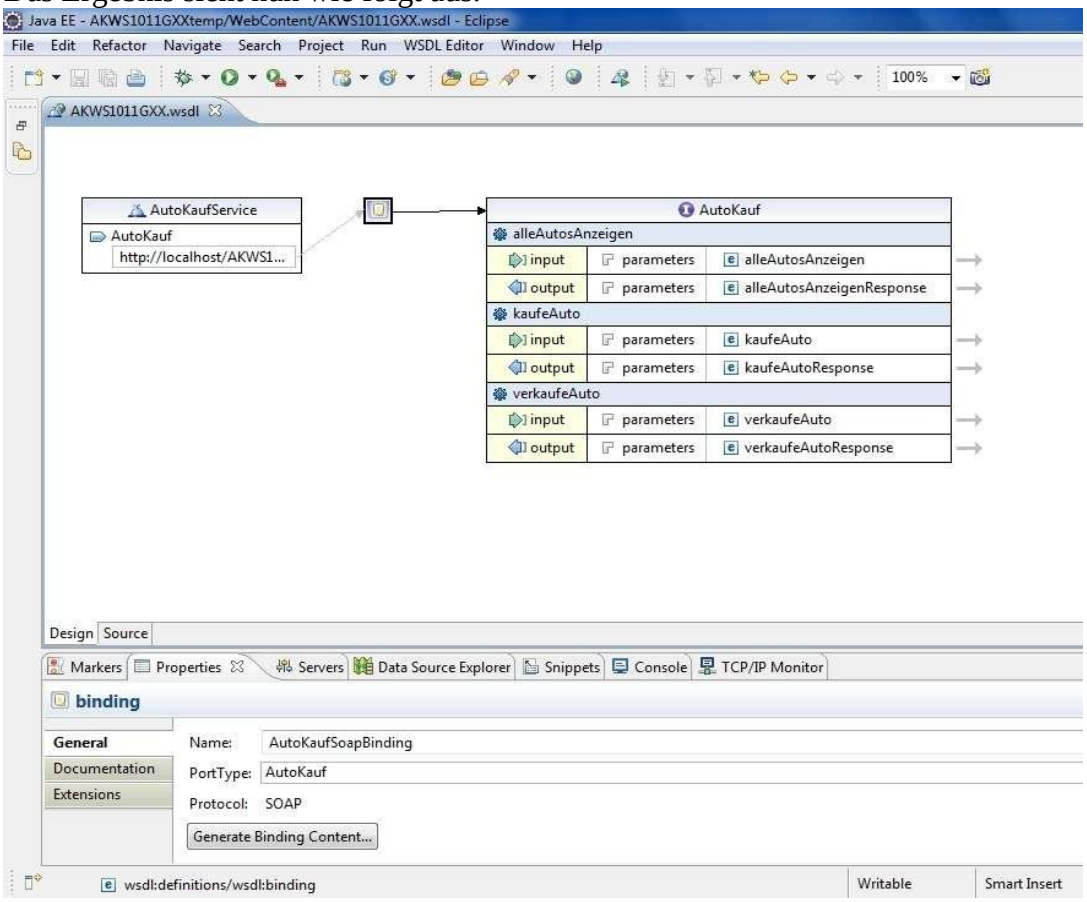
Zurück in der WSDL-Datei-Ansicht erstellen wir eine neue Bindung `AutoKaufSoapBinding`. Dieser Bindung ordnen wir den Porttyp `AutoKauf` zu.

Anschließend ordnen wir dem Service-Port `AutoKauf` die Bindung `AutoKaufSoapBinding` zu.

Abschließend generieren wir in der Bindung einen neuen **Binding Content**.



Das Ergebnis sieht nun wie folgt aus.



In der **Source**-Ansicht des WSDL-Editors können wir den XML-Code der Datei noch einmal begutachten. Sollte die Formatierung schlecht sein, kann mit Rechtsklick auf den Quellcode

Format -> Document die Pretty-Print-Funktion des Eclipse XML-Editors angewendet werden.

In der Quellcode-Ansicht müssen wir nun noch den tns des Inline-Schemas ändern. Dazu erstellen wir im `Definitions`-Tag der WSDL-Datei einen zusätzlichen Namespace.

```
xmlns:typesns="http://AKWS1011GXX/AutoKauf/Types"
```

Im (Inline-) `Schema`-Element ändern wir das Attribut `targetNamespace` zu `"http://AKWS1011GXX/AutoKauf/Types"` und ändern anschließend die Präfixe der betroffenen Element- bzw. Typaufrufe von `tns` zu `typesns`. Durch Speichern des Dokumentes werden die betroffenen Stellen mit einem roten Kreuz markiert. Die fertige WSDL-Datei kann mit der im WAR-Archiv verglichen werden.

2.2 Web Service aus AKWS1011GXX.wsdl erstellen

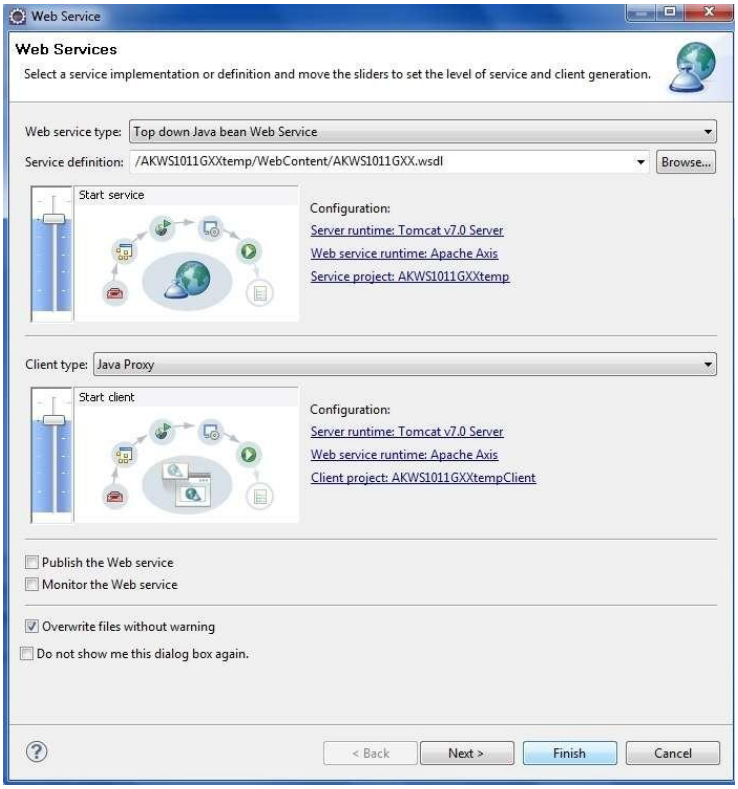
Wenn es keine Fehler gibt in der erstellten WSDL-Datei, können wir die Web Service Klassen automatisch generieren lassen.

Die generierten Methoden sind natürlich zunächst noch leer. Die Implementierung wird weiter unten im Tutorial beschrieben.

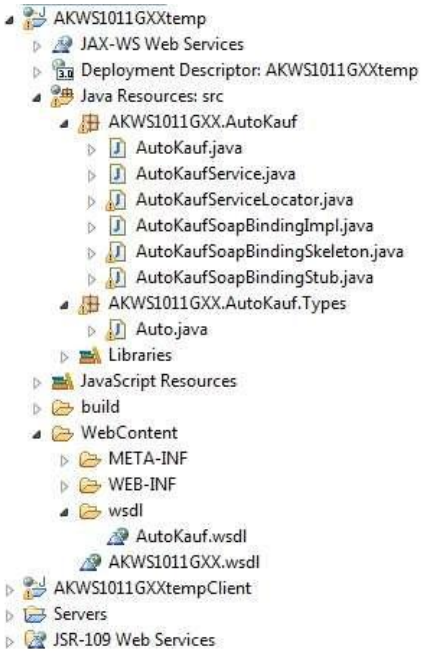
Vorsicht! Wird dieser Assistent später erneut aufgerufen, werden die implementierten Methoden überschrieben. Die vorherige Implementierung wird zwar als bak-Datei gespeichert. Wird (z.B. im Fehlerfall) der Assistent noch einmal ausgeführt, ist auch die bak-Datei überschrieben (→ gelegentlich eine Sicherung des Projektes erstellen).

Mit Rechtsklick auf die WSDL-Datei ***AKWS1011GXX*** im Ordner ***WebContent:*** **Web Services -> Generate Java bean skeleton** starten wir den Wizard zum Erstellen des Grundgerüsts für den Web Service. Es ist wichtig, dass an dieser Stelle auch ein Client mit erstellt wird obwohl wir den Client eigentlich nicht benötigen.

Den folgenden Dialog bestätigen wir mit **Finish**.

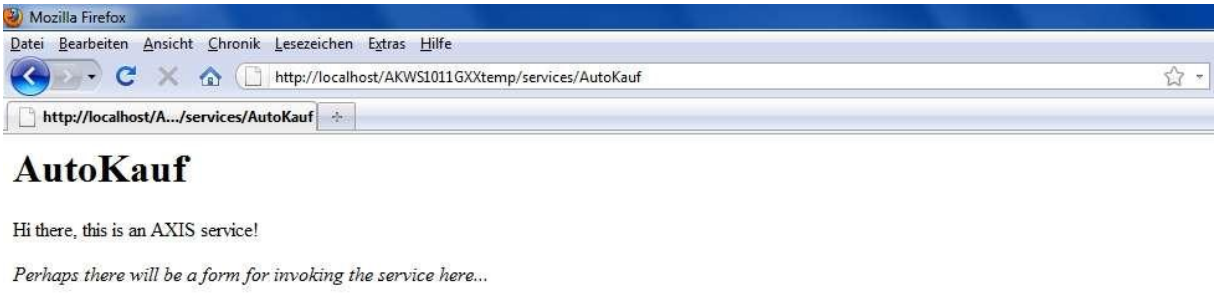

Web Service
Web Services
Select a service implementation or definition and move the sliders to set the level of service and client generation.
Web service type:
Top down Java bean Web Service
Service definition:
/AKWS1011GXXtemp/WebContent/AKWS1011GXX.wsdl
Browse...
Start service
Configuration:
Server runtime: Tomcat v7.0 Server
Web service runtime: Apache Axis
Service project: AKWS1011GXXtemp
Client type:
Java Proxy
Start client
Configuration:
Server runtime: Tomcat v7.0 Server
Web service runtime: Apache Axis
Client project: AKWS1011GXXtempClient
Publish the Web service
Monitor the Web service
Overwrite files without warning
Do not show me this dialog box again.
< Back
Next >
Finish
Cancel

Folgende Klassen-Struktur sollte erstellt wurden sein.

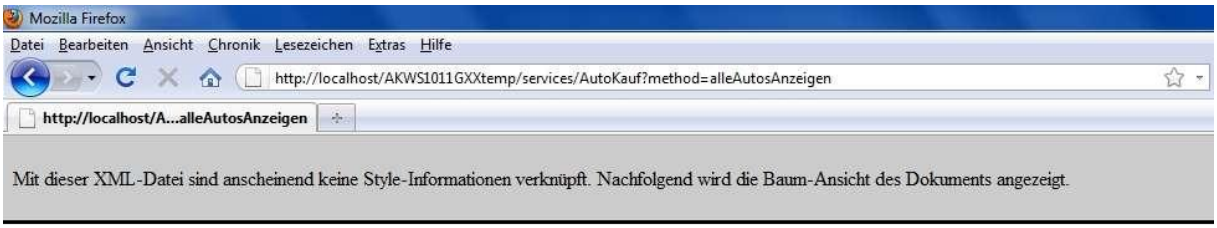


Es fällt auf, dass eine weitere WSDL-Datei erstellt wurden ist. Beim Vergleich beider Dateien in der **Source**-Ansicht des WSDL-Editors sollte bis auf die Reihenfolge einiger Attribute alles identisch sein.

Den erstellten Web Service können wir nun testen - z. B. mit dem Browser. Zuerst prüfen wir unter http://localhost/AKWS1011GXXtemp/services/AutoKauf, ob der Service aktiv ist.

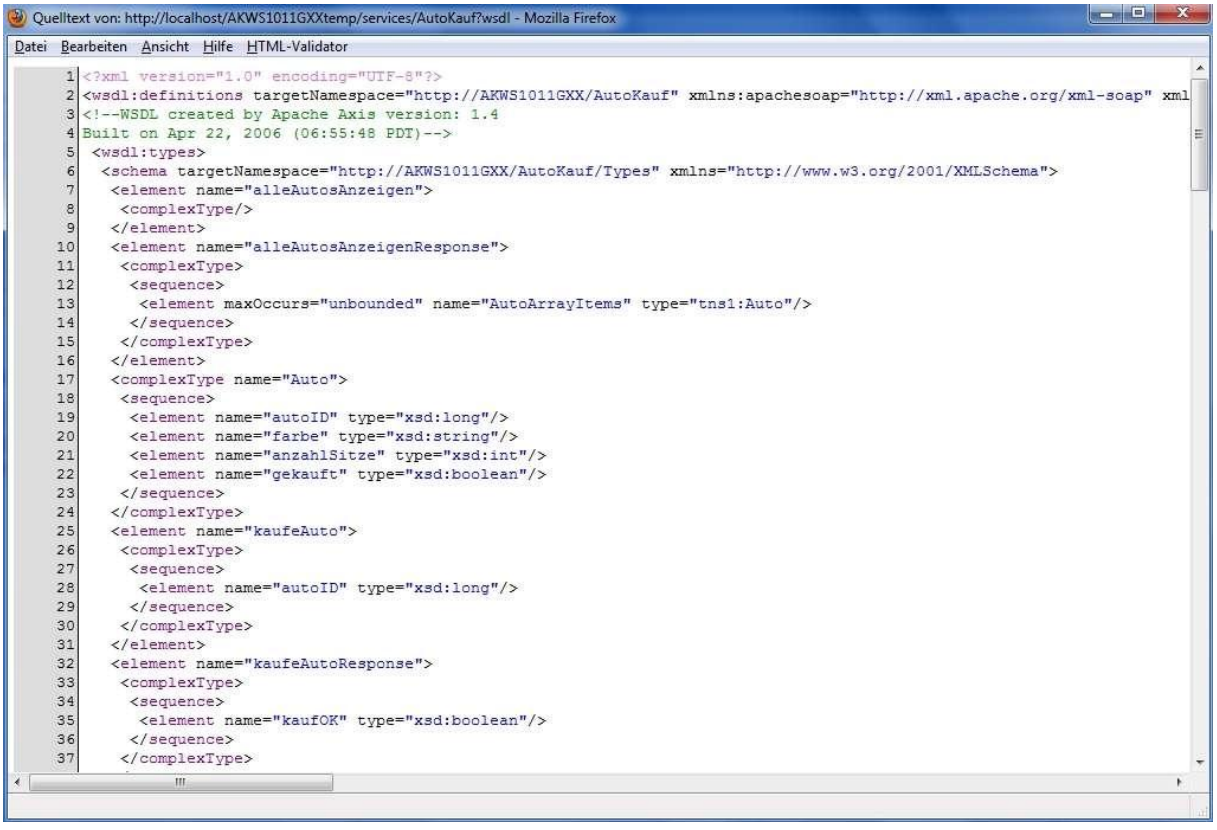


Nun testen wir die Methode `alleAutosAnzeigen`. Das Ergebnis ist natürlich leer (`nil` oder auch `null`), da wir im Java-Skeleton noch keine Funktionalität hinter die Methode `alleAutosAnzeigen` gelegt haben.




```
−<soapenv:Envelope>
  −<soapenv:Body>
    −<alleAutosAnzeigenResponse>
        <AutoArrayItems xsi:nil="true"/>
      </alleAutosAnzeigenResponse>
    </soapenv:Body>
  </soapenv:Envelope>
```

Mit der URL http://localhost/AKWS1011GXXtemp/services/AutoKauf?wsdl können wir die WSDL-Beschreibung anzeigen lassen.




```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <wsdl:definitions targetNamespace="http://AKWS1011GXX/AutoKauf" xmlns:apachesoap="http://xml.apache.org/xml-soap" xml
3  <!--WSDL created by Apache Axis version: 1.4
4  Built on Apr 22, 2006 (06:55:48 PDT)-->
5   <wsdl:types>
6    <schema targetNamespace="http://AKWS1011GXX/AutoKauf/Types" xmlns="http://www.w3.org/2001/XMLSchema">
7     <element name="alleAutosAnzeigen">
8      <complexType/>
9     </element>
10    <element name="alleAutosAnzeigenResponse">
11     <complexType>
12      <sequence>
13       <element maxOccurs="unbounded" name="AutoArrayItems" type="tns1:Auto"/>
14      </sequence>
15     </complexType>
16    </element>
17    <complexType name="Auto">
18     <sequence>
19      <element name="autoID" type="xsd:long"/>
20      <element name="farbe" type="xsd:string"/>
21      <element name="anzahlSitze" type="xsd:int"/>
22      <element name="gekauft" type="xsd:boolean"/>
23     </sequence>
24    </complexType>
25    <element name="kaufeAuto">
26     <complexType>
27      <sequence>
28       <element name="autoID" type="xsd:long"/>
29      </sequence>
30     </complexType>
31    </element>
32    <element name="kaufeAutoResponse">
33     <complexType>
34      <sequence>
35       <element name="kaufOK" type="xsd:boolean"/>
36      </sequence>
37     </complexType>
```

Beim Vergleich der beiden WSDL-Referenzen

<ECLIPSE_WORKSPACE>\AKWS1011GXXtemp\WebContent\wsdl\AutoKauf.wsdl

und

http://localhost/AKWS1011GXXtemp/services/AutoKauf?wsdl (**View Page Source**)

fällt auf, dass die Referenzen nicht identisch sind. Wo ist zum Beispiel unser Namespace-Präfix typesns. Das ist verwunderlich, da die von uns erstellte lokale WSDL-Datei doch zumindest nach der W3C-Spezifikation korrekt ist und auch die Klassengenerierung fehlerfrei durchlief. Der Grund ist, das Axis die Web WSDL Referenz dynamisch anhand der Klassenstruktur erstellt und dabei der ursprünglichen WSDL-Datei keine Beachtung schenkt. Und diese Web WSDL Referenz ist in diesem Fall schlichtweg falsch. So führt diese Abweichung der Web WSDL Referenz dazu, dass diese für die Client-Erstellung (z. B. in den weiteren Tutorials) praktisch nicht verwendet werden kann. Nun könnte man sagen: ‚Gut nehmen wir halt die (selbst erstellte) lokale WSDL-Datei ***AKWS1011GXX.wsdl***!' Und in der Tat mit dieser Datei gebe es keine Probleme. Aber bei Web Services ist das nicht der richtige Weg. Erinnern wir uns: Web Service Clients suchen ggf. im UDDI nach Service Beschreibungen und binden diese dynamisch ein. Bei diesem Vorgang wird immer die Web WSDL Referenz genommen – diese sollte also funktionieren.

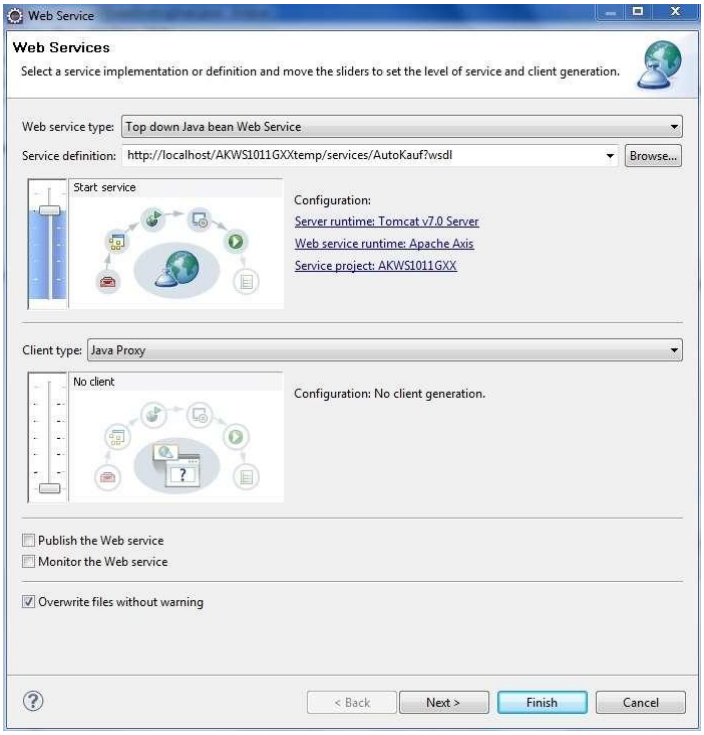
Für jetzt bedeutet das, dass wir uns etwas einfallen lassen müssen – für später können wir nur hoffen, dass dieser Bug behoben wird.

Und nun kommt das ‚temporär' ins Spiel. Wir haben ja bisher nur ein temporäres Projekt erstellt. Mit dessen Hilfe erstellen wir nun das Grundgerüst für unseren richtigen Web Service. Statt der lokalen WSDL-Datei verwenden wir diesmal die Web WSDL Referenz des ***AKWS1011GXXtemp*** Web Service.

Dazu öffnen wir mit **File -> New -> Other… -> Web Services -> Web Service** den Wizard zum Erstellen eines Web Service. Als **Web Service Typ** nehmen wir Top-Down (Generierung aus der WSDL-Referenz). Als **Service Definition** nehmen wir die Web WSDL Referenz unseres Projektes ***AKWS1011GXXtemp***:

http://localhost/AKWS1011GXXtemp/services/AutoKauf?wsdl

Als **Service Project** legen wir ***AKWS1011GXX*** fest und erstellen nun das Web Service Grundgerüst.



Auch hier gilt: Die generierten Methoden sind natürlich zunächst noch leer. Die Implementierung wird weiter unten im Tutorial beschrieben und **wird dieser Assistent später erneut aufgerufen, werden die implementierten Methoden überschrieben.**

Achtung! Die temporären Projekte können wir bestehen lassen. Falls es jedoch gelöscht wird sollte auf jeden Fall die von uns erstellte WSDL-Datei ***AKWS1011GXX.wsdl*** gesichert werden.

Die für dieses Projekt automatisch generierte WSDL-Datei schauen wir uns lieber nicht an. Auch in den weiteren Tutorials wird immer die manuell erstellte WSDL-Datei ***AKWS1011GXX.wsdl*** als Bezug genommen.

2.3 Implementieren der Web Service Methoden

Wir hinterlegen nun noch die Methoden mit entsprechender Funktionalität – vorerst noch beispielhaft. Das Parsen der XML-Datei (***Autos.xml***) folgt im Tutorial ***XML-Datei für Web Service auslesen und schreiben***.

Fangen wir an mit der Operation `alleAutosAnzeigen`. Hier erstellen wir einfach ein Array von `Autos`, das bei einer Anfrage als Response zurückgeben wird.

Die anderen beiden Methoden sind sichtlich einfacher aufgebaut. Auch hier erstellen wir eine kleine Beispielanwendung, um die Web Service Operationen zu testen.

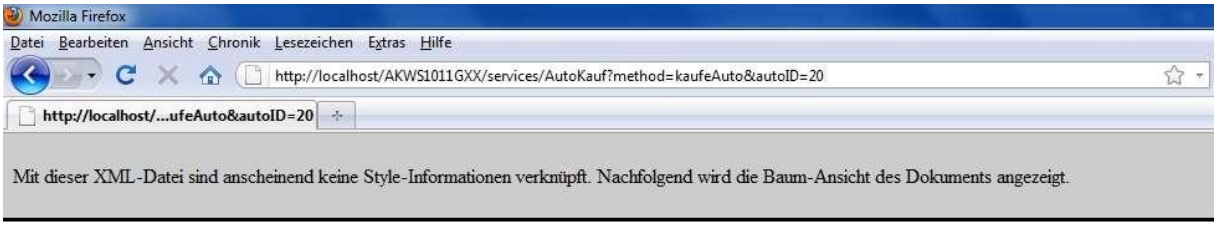
Das folgende Listing zeigt den Quellcode der Service-Implementation. Zum Testen reicht das Speichern der Klasse. Den Rest (Deployen) erledigt das Tomcat-Plugin im Hintergrund.

```
package AKWS1011GXX.AutoKauf;

import AKWS1011GXX.AutoKauf.Types.*;

public class AutoKaufSoapBindingImpl implements AKWS1011GXX.AutoKauf.AutoKauf{
    public AKWS1011GXX.AutoKauf.Types.Auto[] alleAutosAnzeigen() throws java.rmi.RemoteException {
        return new AKWS1011GXX.AutoKauf.Types.Auto[] {
                                    new Auto(1,"rot",3,false),
                                    new Auto(2,"blau",5,true)};
    }

    public boolean kaufeAuto(long autoID) throws java.rmi.RemoteException {
        //gib true bei ungerader ID zurück
        return (autoID % 2 == 0) ? false : true;
    }

    public boolean verkaufeAuto(long autoID) throws java.rmi.RemoteException {
        //gib true bei gerader ID zurück
        return (autoID % 2 == 0) ? true : false;
    }

}
```

Nun kann der Service mit seinen Operationen ausführlich getestet werden. Wird die Methode `kaufeAuto` z. B. mit der geraden `autoID=20` aufgerufen, liefert sie `false` zurück.

http://localhost/AKWS1011GXX/services/AutoKauf?method=kaufeAuto&autoID=20

Schauen wir uns das Ergebnis im Browser an.




```
−<soapenv:Envelope>
  −<soapenv:Body>
    −<kaufeAutoResponse>
        <kaufOK>false</kaufOK>
      </kaufeAutoResponse>
    </soapenv:Body>
  </soapenv:Envelope>
```

Diesen Service hätte man sicher auch (schneller) bottom-up entwickeln können – ein, zwei Klassen und fertig. Das Problem ist dann allerdings, dass man nur geringen Einfluss auf die daraus generierte WSDL-Datei hat. Das wir hier den Umweg mit dem temporären Projekt gehen müssen ist schade. Dennoch ist der Top-Down-Weg für die Web Service Erstellung der bessere. Stellen wir uns vor, wir hätten für eine korrekte Klassenstruktur alle Klassen selbst erstellen und vor allem korrekt implementieren müssen. Ein weiterer Vorteil ist, dass wenn die WSDL-Datei erst einmal verfügbar ist, kann daraus ein Web Service in jeder beliebigen (unterstützten) Programmiersprache erstellt werden, vorausgesetzt es gibt einen entsprechenden Code Generator.

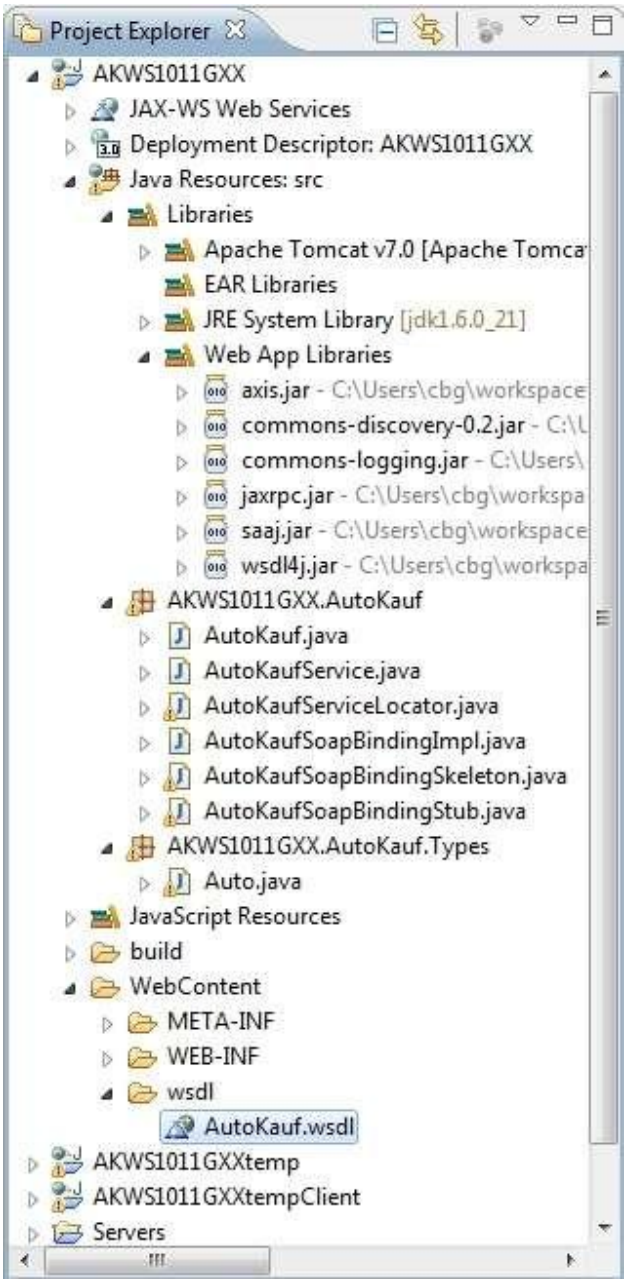
Außerdem ist dies eine gute Möglichkeit sich mit dem Thema WSDL auseinander zu setzten, denn ich gehe davon aus, dass sich der eine oder andere trotz dieser Anleitung noch durch den XML-Code wühlen musste.

Für die Zukunft müssen wir auf stabile WSDL-Editoren und ein besseres Zusammenspiel mit den Code Generatoren hoffen.

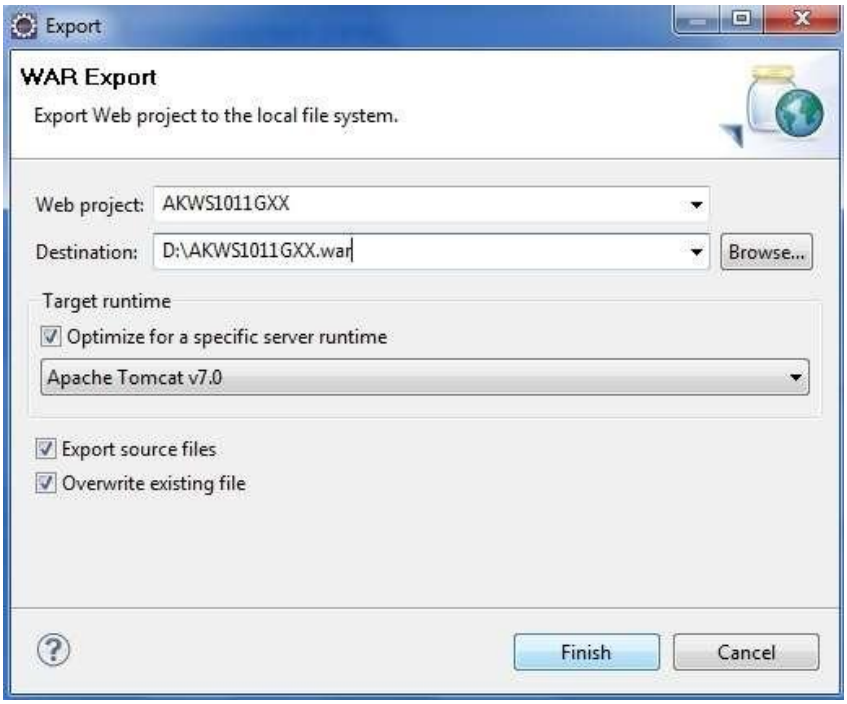
3 Erstellen des Archivs AKWS1011GXX.war

Die Erstellung eines WAR-Archivs hat viele Vorteile. Vorausgesetzt der Quellcode ist mit eingebunden, kann ein anderes Gruppenmitglied das Archiv als Eclipse Projekt importieren und weiterbearbeiten. Ein WAR-Archiv kann auch in den ***WebApps***-Ordner eines ‚standalone' Tomcat abgelegt werden. Nach (Neu-) Start des Tomcat wird das Archiv automatisch deployed. Der Web Service steht dann auch ohne Entwicklungsumgebung zu Verfügung.

Bevor wir abschließend das WAR-Archiv erstellen, schauen wir uns zum Vergleich im Package-Explorer die gesamte Dateistruktur noch einmal an.



Mit Rechtsklick auf das Projekt ***AKWS1011GXX*** **Export -> WAR file** erstellen wir das entsprechende WAR-Archiv.



Zum Testen des Archivs können wir Eclipse beenden, die Datei ***AKWS1011GXX.war*** in den Tomcat-WebApps-Ordner (<Tomcat root>\WebApps\) kopieren und Tomcat als Applikation (bzw. Dienst) starten. Im Browser können wir uns nun das Ergebnis ansehen.

Entsprechend dieser Anleitung kann auch das Projekt ***Calculator*** als WAR-Archiv exportiert werden, da dieses ebenfalls in den folgenden Tutorials verwendet wird.