Georg-August-Universität Göttingen
Fakultät für Forstwissenschaften und Waldökologie
Abteilung Ökoinformatik, Biometrie und Waldwachstum

Einsatz von Webservices für forstliche Simulationsmodelle

Use of Web Services for forest simulation models

Masterarbeit vorgelegt von:

Jonas Müller
Deisterstraße 8, 37081 Göttingen
E-Mail: jonas_mueller@hotmail.de

im Schwerpunkt Waldökosystemanalyse und Informationsverarbeitung
des Studiengangs Forstwissenschaften und Waldökologie.

1. Gutachter: Prof. Dr. Winfried Kurth
2. Gutachter: Prof. Dr. Joachim Saborowski
Betreuung: Jan C. Thiele

Datum der Abgabe: 16.03.2012

Zusammenfassung

Um der Forderung nach Nachhaltigkeit in der Forstwirtschaft gerecht zu werden, wird gerade im Angesicht des Klimawandels und der damit einhergehenden Risiken, die Planung und Entscheidungsfindung zunehmend wichtiger (Pretzsch et al. 2008). Zur Unterstützung der forstlichen Planung werden Simulationsmodelle eingesetzt, weil damit u.a. Zukunftsszenarien analysiert und die Entscheidungsfindung erleichtert werden kann. Auch im BEST-Projekt, in dem „regional angepasste Konzepte und innovative Systemlösungen zur Produktion von Biomasse“ (FZW 2012) entwickelt und bewertet werden, sollen Simulationsmodelle als Beratungswerkzeuge zum Einsatz kommen. Das Teilprojekt UP 3 hat innerhalb des BEST-Projektes u.a. die Aufgabe, eine Auswahl dieser Simulationsmodelle über Webservices verfügbar zu machen (Kurth et al. 2010). Im Rahmen der vorliegenden Arbeit wird diese Zielsetzung aufgegriffen und die Eignung verschiedener Webservice-Ansätze für die Bereitstellung von forstlichen Simulationsmodellen untersucht.

Ausgehend von einer Einführung in das Thema Webservices, werden vier verschiedene Ansätze zur Erstellung von Webservices vorgestellt. Bei diesen vier Ansätzen handelt es sich um XML-RPC, Webservices, die auf den beiden Standards SOAP und WSDL basieren, REST-Webservices und Web Processing Services (WPS). Diese Webservice-Ansätze werden auf die Eignung für den Einsatz mit forstlichen Simulationsmodellen untersucht und anschließend hinsichtlich der Eignung bewertet und miteinander verglichen. Ziel der Arbeit ist es, eine Empfehlung für einen Webservice-Ansatz zu geben und deren Vor- und Nachteile für den Einsatz von forstlichen Simulationsmodellen aufzuzeigen.

Die Bewertung der Ansätze zeigte, dass sich Web Processing Services und auf SOAP und WSDL basierte Webservices am besten für die Anbindung von Simulationsmodellen eignen. Abhängig ist der am besten geeignete Ansatz dabei aber auch von der Komplexität und den Eigenschaften des anzubindenden Modells sowie dem Vorwissen des Entwicklers bezüglich des gewählten Webservice-Ansatzes. Eher weniger für die Anbindung mit Webservices geeignet sind Simulationsmodelle, die große Datenmengen verarbeiten oder zurückgeben. Außerdem fällt durch den Einsatz von Webservices ein erhöhter Entwicklungsaufwand an, der bei der Überlegung, einen Webservice für die Anbindung eines Simulationsmodells einzusetzen, zu berücksichtigen ist. Der erhöhte Aufwand kann sich aber vor allem dadurch auszahlen, dass die Möglichkeit der Wiederverwendbarkeit des Modells erhöht wird und dass die Integration des Modells in bestehende Programme erleichtert wird.

Abstract

To meet the demand for sustainable forest management, especially in terms of climate change and subsequent risks, planning and decision-making becomes increasingly important (Pretzsch et al. 2008). Simulation models support forest planning since they allow studying future scenarios thus facilitating the decision-making process. Within the scope of the BEST-Project, where regionally adapted approaches and innovative system solutions for producing biomass are developed and evaluated (FZW 2012), simulation models serve as supporting tools. One of the tasks of the sub-project UP3, which forms part of the BEST-Project, is making a selection of these simulation models available via Web Services (Kurth et al. 2010). In the present thesis this aim is adopted and the suitability of different Web Service approaches for providing forest simulation models is analysed.

Starting with an introduction to Web Services four different approaches for developing Web Services are presented. The four approaches are XML-RPC, Web Services based on SOAP and WSDL, REST Web Services and Web Processing Services (WPS). These approaches are analysed, evaluated and compared against each other regarding their suitability for forest simulation models. The present thesis' aim is to recommend a Web Service approach and to point out advantages and disadvantages of the use of Web Services for forest simulation models.

The evaluation showed that WPS and Web Services based on SOAP and WSDL are the best Web Service approaches for forest simulation models. But which approach proves to be the best depends on complexity and features of the model to be deployed and on the prior knowledge of the developer regarding the chosen Web Service approach, too. Simulation models, which work with a huge amount of Input or Output data, are less suitable for the deployment with Web Services. Furthermore, the use of Web Services increases the implementation effort, which has to be taken into account, when developing Web Services for simulation models. The high implementation effort can be worthwhile though since reuse of models and integration into existing programs is facilitated.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1. Einleitung

Simulationsmodelle sind in der Forstwirtschaft u.a. deswegen von großer Bedeutung, weil die langen Produktionszeiträume eine Beantwortung vieler Fragen mit Versuchen in vielen Fällen nicht zeitnah zulassen (Nagel und Sprauer 2009). Deswegen werden Modelle vor allem dort eingesetzt, wo Planungen und Zukunftsprognosen eine Rolle spielen. Die Komplexität forstlicher Simulationsmodelle reicht von einfachen empirischen Wachstumsmodellen (Ertragstafeln) bis hin zu computergestützten Entscheidungsunterstützungssystemen, bei denen viele verschiedene Modelle miteinander kombiniert werden. Diese Kombination einzelner Modelle zu einem Modellsystem ist besonders dann mit Schwierigkeiten verbunden, wenn die zu verbindenden Modellimplementierungen durch unterschiedliche Entwickler programmiert werden, weil die Modelle inhaltlich und technisch gut aufeinander abgestimmt werden müssen und dadurch ein großer Koordinationsaufwand entsteht. Um diese technischen Schwierigkeiten und den Aufwand zu reduzieren, können Webservices für forstliche Simulationsmodelle eingesetzt werden.

Eine derartige Situation, nämlich dass verschiedene Entwickler Simulationsmodelle bereitstellen, die unabhängig voneinander entwickelt werden, aber dennoch die Möglichkeit besitzen sollen, miteinander zu interagieren, ist im BMBF-Verbundprojekt „Bioenergie-Regionen stärken (BEST)“ gegeben.

Die Verwendung von Webservices bietet sich hier aus unterschiedlichen Gründen an. Wenn die Modelle über Webservices angebunden werden, verfügen sie über einheitlich definierte Schnittstellen. Dadurch könnten die Modelle unabhängig voneinander entwickelt, getestet und angewendet werden und eine anschließende Verbindung der Modelle wird einfacher. Daneben kann ein Austausch der Modelle sowie das Einbinden von externen Modellen erleichtert werden.

1.1 Das Forschungsprojekt

Das BEST-Verbundprojekt besteht aus 31 Teilprojekten, die von unterschiedlichen Institutionen bzw. Abteilungen bearbeitet werden (FZW 2012). Ziel des Projektes ist es, „regional angepasste Konzepte und innovative Systemlösungen zur Produktion von Biomasse zu entwickeln und im Hinblick auf ökologische und ökonomische Auswirkungen zu bewerten“ (FZW 2012). Es werden verschiedene Verfahren zur Biomassenerzeugung, wie beispielsweise Kurzumtriebsplantagen, Agroforstsysteme oder Nutzung von Grünlandbiomasse hinsichtlich unterschiedlicher Aspekte, wie z.B. ökologische Folgen, Wirtschaftlichkeit, Nachhaltigkeit oder Wechselwirkungen mit dem Klimawandel untersucht. Daraus werden verschiedene Hand-

lungsempfehlungen und Nutzungskonzepte hinsichtlich der Landnutzungsform für unterschiedliche Regionen abgeleitet. Die gewonnenen Erkenntnisse sollen zum Teil in Simulationsmodellen formalisiert werden und mit Hilfe von Zukunftsszenarien, z.B. zur Klimaentwicklung, eine Basis für die Entscheidungsfindung in der Praxis schaffen. Die Anbindung dieser Simulationsmodelle und die Integration in ein web-basiertes Beratungswerkzeug gehören zu den Aufgaben des Teilprojektes UP 3 (Kurth et al. 2010), in dessen Rahmen diese Arbeit entstanden ist.

1.2 Die Ziele

In dieser Arbeit soll untersucht werden, ob und inwieweit Webservices geeignet sind, um forstliche Simulationsmodelle bereitzustellen. Zuerst soll ein Überblick über die verschiedenen Ansätze aus dem Bereich Webservice gegeben werden, als Basis für ein grundlegendes Verständnis von Webservices. Auch zu diesem Zweck und um zu untersuchen, welche der Ansätze sich für den Einsatz mit forstlichen Simulationsmodellen eignen, werden beispielhaft Webservices mit den unterschiedlichen Ansätzen implementiert. Angebunden werden sollen dabei verschiedene Simulationsmodelle. Danach soll eine Bewertung erfolgen, um eine Empfehlung abgeben zu können, welcher der Ansätze sich am besten für die Anbindung von forstlichen Simulationsmodellen eignet. Ein weiteres Ziel der Arbeit ist es, die Möglichkeiten und die Grenzen bzw. die Vor- und Nachteile des Einsatzes von Webservices für forstliche Simulationsmodelle aufzuzeigen. Wichtige Aspekte dabei sind auch, welche Eigenschaften die Simulationsmodelle besitzen sollten, damit eine Anbindung mit Webservices über das Internet, unabhängig von der gewählten Technologie, sinnvoll ist und welche Art von Modellen für Webservices möglicherweise weniger geeignet sind.

1.3 Aufbau der Arbeit

Ausgehend von einer allgemeinen Erklärung von Webservices werden in Kapitel 2 (Einführung in Webservices) vier verschiedene Ansätze für Webservices aufgezeigt und beschrieben. Anschließend werden in Kapitel 3 (Implementierungen) die Implementierungen von Webservices mit diesen vier Ansätzen für verschiedene Beispiel-Modelle vorgestellt. Auf die Implementierungen folgt in Kapitel 4 (Bewertung der Webservice-Ansätze) eine Bewertung der Ansätze hinsichtlich unterschiedlicher Aspekte, die für die Erstellung von Webservices für forstliche Simulationsmodelle wichtig sind. In Kapitel 5 (Diskussion) werden dann die Vor- und Nachteile von Webservices und außerdem Sicherheits- und Performance-Aspekte diskutiert. Des Weiteren werden in der Diskussion die Fragen aufgegriffen, welche Arten von Simulationsmodellen sich am besten für den Einsatz von Webservices eignen und welche der

Webservice-Ansätze sich am besten für forstliche Simulationsmodelle eignen. Abschließend werden in Kapitel 6 (Schlussfolgerungen) die wichtigsten Erkenntnisse aus der Arbeit zusammengefasst dargestellt und ein Ausblick gegeben.

2. Einführung in Webservices

2.1 Webservices

Für Webservices gibt es verschiedene Erklärungsansätze bzw. Auffassungen der Thematik, es existiert jedoch keine verbindliche Definition (Melzer 2008). Nach (Richardson und Ruby 2007) ist alles, was sich im Web befindet, ein Webservice. Sie bezeichnen eine Webseite, bzw. deren Bereitstellung bereits als einen Webservice. Eine konkretere Beschreibung ist bei Gillett et al. (2002) von Forrester Research zu finden: „Software designed to be used by other software via Internet protocols and formats“. Demnach handelt es sich bei Webservices um Programme oder Daten, die mit Hilfe von Internetprotokollen und -formaten aus anderen Programmen heraus aufgerufen werden können. Hierin liegt ein wichtiger Aspekt für die angestrebte Modellanbindung, nämlich der Aufruf von Funktionen auf entfernten Computern (RPC – Remote Procedure Call). Jedoch ist die Definition von Forrester Research noch relativ allgemein gehalten und enthält keine spezifischen Angaben über die eingesetzten Technologien. In der folgenden Definition des W3C (Booth et al. 2004) werden die eingesetzten Protokolle und Formate genannt: „A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards”. Nach dieser Definition werden Webservices mit SOAP und WSDL in Verbindung gebracht. Sie bezieht sich direkt auf die Art von Webservices, die in dieser Arbeit als SOAP-Webservices bezeichnet werden. Außerdem geht aus der Definition hervor, dass Webservices Interoperabilität ermöglichen, also Kommunikation zwischen Maschinen unabhängig von der Plattform und der Programmiersprache.

Diese erste Definition sagt aus, dass es sich bei Webservices um Anwendungen oder Dienste handelt, die über das Internet angesprochen werden können und die den Aufruf von Funktionalitäten und RPCs über das Internet ermöglichen. Die zweite Definition erweitert diese Aussage, indem ein konkreter Ansatz für Webservices aufgezeigt wird, nämlich SOAP-Webservices bzw. SOAP, WSDL und HTTP, mit dem Webservices realisiert werden können.

Neben diesem finden sich weitere Ansätze für Webservices, nämlich REST und XML-RPC. Hierbei handelt es sich um zwei SOAP verwandte Technologien (Melzer 2008), mit denen ebenfalls Webservices realisiert werden können. Neben diesen drei Ansätzen, die ausgewählt wurden, weil diese die am weitesten verbreiteten Ansätze für Webservices sind (vgl. Melzer 2008, Burbiel 2007), wird in dieser Arbeit der OGC-Standard WPS behandelt, weil darauf basierende Webservices im Bereich der Geodatenverarbeitung vielfach eingesetzt werden. Der Standard beschreibt eine standardisierte Schnittstelle, welche die Veröffentlichung von räumlichen Prozessen vereinfachen soll (Open Geospatial Consortium 2007). Er gibt unter anderem vor, welche Funktionalitäten OGC-konforme Webservices anbieten sollen und welche Technologien und Web-Standards für diese Webservices eingesetzt werden müssen. Das Konzept hinter WPS ist aber nicht nur für Geografische Informationssysteme, sondern auch für andere Bereiche einsetzbar (Heimann et al. 2010). Diese vier genannten Ansätze für Webservices, nämlich XML-RPC, SOAP-Webservices, REST-Webservices sowie der Standard WPS werden in den nächsten Abschnitten vorgestellt.

2.2 XML-RPC

XML-RPC ist einer der einfachsten Webservice-Ansätze, der es Computern ermöglicht, Prozeduren auf anderen Computern bzw. Servern über ein Netzwerk aufzurufen (Laurent et al. 2001). Zum einen stellt er einen möglichen Lösungsansatz für die Anbindung von Simulationsmodellen mit Webservices dar, zum anderen lässt sich anhand dieses einfachen Webservices die Funktionsweise von SOAP-Webservices besser verstehen. Dies ist dadurch zu erklären, dass XML-RPC vergleichbar ist „mit einer vorläufigen, inoffiziellen SOAP Version“ (Melzer 2008). XML-RPC ist nicht vom W3C standardisiert, sondern basiert auf einer siebenseitigen Spezifikation des Entwicklers Dave Winer (vgl. Winer 1999).

Für einen XML-RPC werden die Auszeichnungssprache XML und das HTTP-Protokoll verwendet. Es greift also auf weit verbreitete Techniken zurück, die einen Einsatz auf praktisch allen Plattformen zulassen und XML-RPC damit unabhängig vom Betriebssystem und der Programmiersprache machen, solange von dem Rechner, auf dem sich die Funktion befindet, HTTP und XML verstanden werden.

Der Ablauf eines XML-RPC kann folgendermaßen beschrieben werden (Laurent et al. 2001): Der Prozeduraufruf wird von einem Client-Programm gestartet, indem es einen XML-RPC-Client aufruft. Dafür müssen diesem der Methodenname, die Eingabeparameter und die Adresse des Ziel-Servers übermittelt werden. Aus dem Methodennamen und den Parametern erstellt der XML-RPC-Client ein XML-Dokument, um es anschließend mit einer HTTP-POST-

Anfrage an den Server zu übermitteln. Abbildung 1 zeigt den beispielhaften Aufbau einer Anfrage für einen XML-RPC.

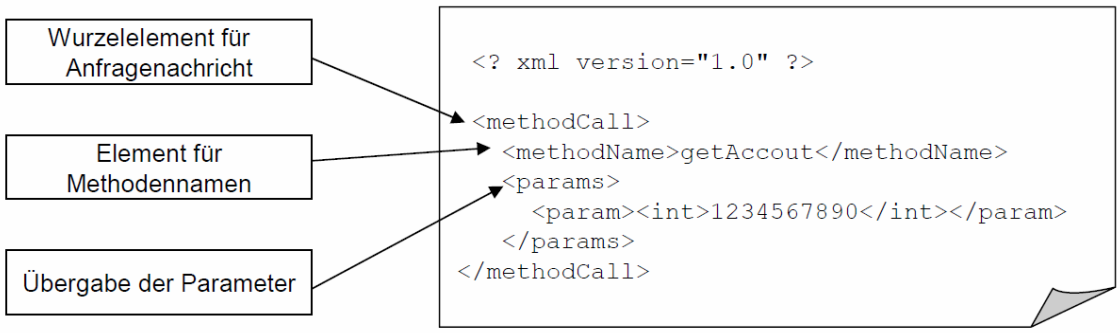

Abbildung 1: Beispielhafte XML-RPC-Anfrage. Quelle: Burghardt und Hagenhoff (2003)

Der Server empfängt dieses XML-Dokument und übermittelt es an einen XML-RPC Listener. Dieser liest aus dem Dokument den Methodennamen und die Parameter aus und ruft dann die entsprechende Methode mit den Eingabeparametern auf. Die Ergebnisse des Methodenaufrufs werden nun wieder an den XML-RPC Prozess übermittelt, der sie in ein XML-Dokument verpackt. Dieses Dokument wird als Antwort auf die HTTP-POST-Anfrage zurückgeliefert. Aus dieser Antwort bzw. dem zugeschickten XML-Dokument zieht der XML-Client schließlich die Rückgabewerte heraus und übergibt sie an das Client-Programm, welches den RPC gestartet hat und nun mit diesen Werten weiter rechnen kann.

XML-RPC-Anfragen sind immer zustandslos und synchronisiert, da nur das HTTP-Protokoll, das diese beiden Eigenschaften besitzt, für den XML-RPC verwendet wird. Das bedeutet, dass jede Anfrage eines Benutzers unabhängig von vorhergehenden und nachfolgenden Anfragen behandelt wird. Der Server speichert keine Informationen über die Anfragen und kann somit nicht feststellen, ob eine Folge von Anfragen vom gleichen Benutzer stammt. Die Zustandslosigkeit kann, wenn nötig, aber aufgelöst werden, indem beispielsweise Informationen über den Zustand mit der HTTP-POST-Anfrage mitgeschickt werden.

Weil die Anfragen synchronisiert sind, folgt auf eine Anfrage immer eine Antwort und somit pausiert der Client-Prozess nach einer Anfrage, bis er eine Antwort bekommen hat.

XML-RPC kann acht verschiedene Datentypen verwenden, darunter die sechs einfachen Datentypen Integer, Double, Boolean, String, Date-Time-Variablen und Binärobjekte. Daneben kann auch mit den beiden komplexeren Datentypen Array und assoziatives Array gearbeitet werden.

Weil es keine Schnittstellenbeschreibung für XML-RPC gibt, müssen die erforderlichen Programme von Hand erstellt werden (Burbiel 2007). Dafür sind in den meisten Programmiersprachen Implementierungen von XML-RPC verfügbar (Melzer 2008). Dazu zählen beispielsweise Apache XML-RPC (für die Programmiersprache Java) (Apache Software Founda-

tion 2010), SimpleXMLRPCServer (für Python) (Python Software Foundation 2010) oder Libiqxmlrpc (für C++) (Dedov 2012). Damit lassen sich Client- und Server-Programme in der jeweiligen Programmiersprache relativ einfach schreiben, welche die nötigen Funktionalitäten wie Erstellung, Auswertung, Übertragung und Empfang der Anfragen und Antworten übernehmen.

Vorteile von XML-RPC sind die relativ einfache Erlern- und Nutzbarkeit, und dass dieses Protokoll in vielen Programmiersprachen umgesetzt werden kann (Burbiel 2007). Ein Nachteil an XML-RPC ist, dass die Spezifikation keine Schnittstellenbeschreibung für dieses Protokoll vorsieht. Dadurch ist es nicht möglich, Client-Programme automatisch zu generieren, und auch eine manuelle Anbindung ist durch das Fehlen einer eindeutigen Schnittstellenbeschreibung unter Umständen schwieriger. Diese fehlende Schnittstellenbeschreibung ist für SOAP-Webservices vorhanden, welche im nächsten Kapitel erläutert werden.

2.3 SOAP-Webservices

SOAP-Webservices sind eine Weiterentwicklung von XML-RPC (Burbiel 2007). Im Gegensatz zu XML-RPC nutzen SOAP-Webservices nicht einfache XML-Nachrichten, sondern SOAP Nachrichten für die Kommunikation, und sie verwenden WSDL, um den Dienst und die Schnittstelle zu beschreiben (Eine nähere Beschreibung dieser beiden Komponenten ist in den nachfolgenden Kapiteln zu finden). SOAP-Webservices werden nicht als Ganzes mit einem eigenen Standard beschrieben, sondern die für diese Art von Webservice eingesetzten Komponenten (SOAP und WSDL) sind standardisiert. Somit ist für SOAP-Webservices auch nicht festgelegt, welches Transportprotokoll für die Nachrichtenübertragung eingesetzt werden muss. SOAP-Webservices sind also nicht wie XML-RPC an das HTTP-Protokoll gebunden, dennoch wird HTTP üblicherweise für SOAP-Webservices verwendet (Löwenstein und Kraeft 2011).

Ein weiterer Standard, der neben dem SOAP- und dem WSDL-Standard im Zusammenhang mit SOAP-Webservices oft genannt wird, ist UDDI (Universal Description, Discovery and Integration). Die Idee von UDDI ist, eine zentrale Sammlung von Webservices zur Verfügung zu stellen, in der Anwender passende Webservices für eigene Aufgabenstellungen finden und Anbieter ihre Webservices einstellen können. 2006 haben jedoch Microsoft und IBM ihre öffentlichen UDDI Registrierungen abgeschaltet. Auch sonst konnten sich öffentliche UDDI Registrierungen nicht durchsetzen (Richardson und Ruby 2007). Aus diesem Grund und weil es für die Bereitstellung von forstlichen Modellen nur eine sehr untergeordnete Rolle spielt, wird UDDI in dieser Arbeit nicht weiter behandelt.

Ein beispielhafter Kommunikationsablauf für einen SOAP-Webservice kann in folgende Schritte untergliedert werden (Goldhammer 2010): Innerhalb einer Anwendung erzeugt ein SOAP-Client eine SOAP-Nachricht mit den Nutzdaten. Diese Nachricht wird mittels eines Transportprotokolls übertragen. Wird HTTP genutzt, dann erzeugt der SOAP-Client einen HTTP-Client, welcher die Nachricht an den Server überträgt. Nachdem der Server die Nachricht empfangen hat, überträgt er sie an den Webservice. Der Webservice extrahiert die übertragenen Informationen aus der SOAP-Nachricht und ruft die entsprechende Funktion auf. Nachdem die Bearbeitung erfolgt ist, werden die Ergebnisse in eine SOAP-Nachricht verpackt und, im Falle der Nutzung von HTTP als Transportprotokoll, über die noch vorhandene HTTP-Sitzung als Antwort zurückgesendet. Mit dieser Antwort kann die Anwendung, die den Webservice aufgerufen hat, nun weiterarbeiten.

In diesem einfachen Fall ähnelt der Ablauf stark dem XML-RPC, es wird ebenfalls HTTP als Transportprotokoll verwendet. SOAP-Webservices bieten aber, unter anderem durch die Unabhängigkeit vom Transportprotokoll, eine höhere Flexibilität als der XML-RPC. Beispielsweise kann in einer SOAP-Nachricht auch eine Route definiert werden, für die angegeben wird, welche Zwischenstationen eine SOAP-Nachricht passieren muss, bevor sie zum Endpunkt gelangt. Die Webservices können also von dem einfachen Anfrage-Antwort Prinzip des XML-RPCs abweichen. Anders als XML-RPC ist SOAP auch nicht auf wenige einfache Datentypen beschränkt, sondern bietet über 40 einfache Datentypen an (Melzer 2008).

Zur Implementierung von SOAP-Webservices werden in verschiedenen Programmiersprachen Bibliotheken angeboten, um Clients und Server zu programmieren. Außerdem gibt es auch Toolkits, mit denen Webservices bzw. Client- und Serverprogramme erstellt werden können, wie z.B. Apache Axis (Apache Software Foundation 2011) oder das .NET Framework (Microsoft 2012).

Ein Vorteil von SOAP ist die Flexibilität, die u.a. durch die Unabhängigkeit vom Transportprotokoll und die große Auswahl an verfügbaren Datentypen erzielt wird. Daneben bringen verschiedene verfügbare Tools den Vorteil, Webservices automatisch und ohne viel Programmieraufwand erstellen zu können (Richardson und Ruby 2007). Durch WSDL wird es erleichtert, Clients für bestehende Webservices zu implementieren, und WSDL macht auch die automatische Generierung von Clients für bestehende Webservices möglich.

Nachteil von SOAP-Webservices ist besonders die Komplexität (Richardson und Ruby 2007), die auch durch die hohe Flexibilität entsteht. Die Erstellung von Clients und Servern ohne die genannten Tools erfordert zumeist gute Programmierkenntnisse (Melzer 2008). Außerdem erzeugen SOAP-Webservices beim Methodenaufruf einen großen Overhead (Löwenstein und Kraeft 2011), der sich negativ auf die Geschwindigkeit auswirken kann.

Die beiden Kernelemente von SOAP-Webservices, nämlich das SOAP-Protokoll und die Schnittstellenbeschreibung WSDL, werden in den beiden nächsten Kapiteln erläutert.

2.3.1 SOAP

SOAP ist ein Netzwerkprotokoll, welches vom W3C standardisiert worden ist. Die erste SOAP-Version wurde 1999 veröffentlicht, die derzeit neuste SOAP-Version 1.2 wurde 2007 als W3C-Standard anerkannt. Bei einer SOAP-Nachricht handelt es sich um ein XML-Dokument, das, gemäß dem Standard, verschiedene Elemente enthalten kann bzw. enthalten muss. **Fehler! Verweisquelle konnte nicht gefunden werden.** und Listing 1 zeigen den schematischen und den konkreten Aufbau eines SOAP-Dokumentes im Vergleich.

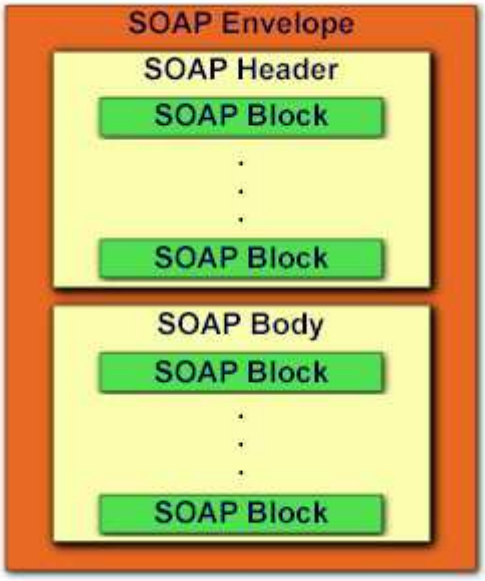

Abbildung 2: Schematischer Aufbau eines SOAP-Dokumentes. Quelle: www.w3c.org

```
<env:Envelope
xmlns:env="http://www.w3.org/2003/05/soap-envelope">
<env:Body>
  <html>
    <head>
      <meta name="Author" content="Ingo">
    </head>
    <body>
      Willkommen bei http://www.soa-buch.de/
    </body>
  </html>
</env:Body>
</env:Envelope>
```

Listing 1: Beispiel einer SOAP-Nachricht. Quelle: Melzer (2008)

Jede SOAP-Nachricht besitzt den sogenannten SOAP-Envelope. Dies ist das Wurzelelement der Nachricht. Darin enthalten sind der SOAP-Header und der SOAP-Body. Der SOAP-Header ist nicht verpflichtend und enthält hauptsächlich Metainformationen, also Informationen über zugrunde liegende Daten (Burbiel 2007). Dazu gehören beispielsweise Sicherheitsinformationen, Transaktionsdaten und Routinginformationen (Zeppenfeld und Finger 2009). „Der SOAP-Body enthält die zu übertragenden Informationen, das heißt die eigentlichen Nutzdaten […] der Nachricht“ (Melzer 2008). Dazu zählen Informationen über die aufzurufende Methode, eventuelle Parameter, die von der Methode verarbeitet werden sollen, oder die Ergebnisse des Funktionsaufrufs, wenn es sich bei der Nachricht um eine Antwort des Webservices handelt.

2.3.2 WSDL

Ein weiterer wichtiger Bestandteil von SOAP-Webservices ist WSDL (Web Service Description Language). Diese vom W3C standardisierte Metasprache basiert ebenso wie SOAP auf der Auszeichnungssprache XML. Mit Hilfe dieses Dokumentes kann ein Client feststellen, welche Funktionalitäten der Webservice anbietet, und diese Funktionalitäten aufrufen. 2001 wurde die WSDL-Version 1.1 (Christensen et al. 2001) veröffentlicht, die derzeit neueste WSDL-Version, die Version 2.0 (Chinnici et al. 2007), wurde 2007 veröffentlicht. Die grundlegende Struktur eines WSDL-Dokuments mit den sechs Hauptelementen ist in Abbildung 3 dargestellt:

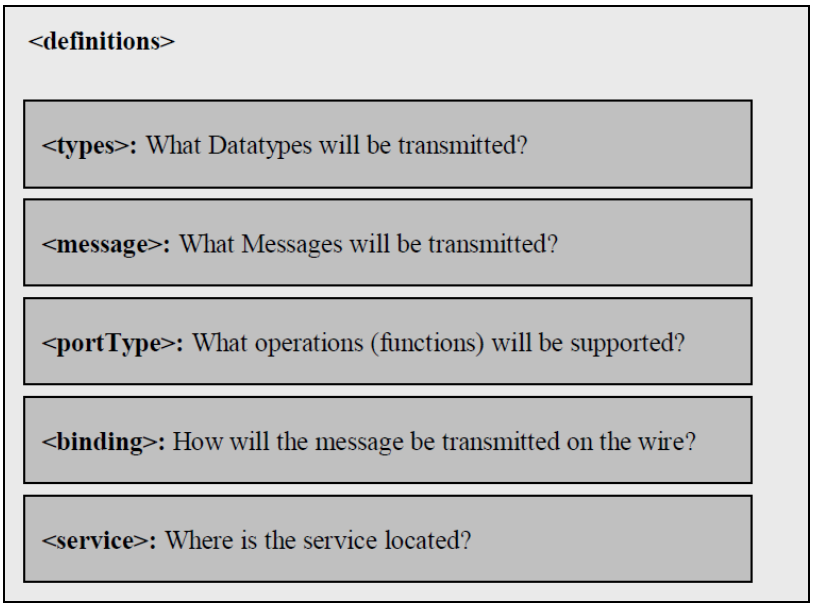

Abbildung 3: Aufbau eines WSDL-Dokumentes. Quelle: Cerami (2002)

- Die Datentypen, die später im Webservice verwendet werden, werden im types-Element angegeben.
- Im message-Element werden die einzelnen Nachrichten definiert, die zwischen Client und Server ausgetauscht werden.
- Die eigentliche Art der Kommunikation mit dem Webservice wird mit dem portType-Element (ab WSDL 2.0: interface) beschrieben. Hier wird angegeben, welche Operationen der Webservice anbietet und welche Nachrichten bei der jeweiligen Operation ausgetauscht werden.
- Das verwendete Protokoll für das Übertragen der Daten wird im binding-Element spezifiziert. Angegeben werden könnte beispielsweise HTTP (GET oder POST) oder SOAP.

- Wo der Webservice erreicht werden kann, wird im service-Element angegeben. Dieses Element enthält ein oder mehrere port-Elemente (ab WSDL 2.0: endpoint), in denen eine URL (Uniform Resource Locator) des Webservices angegeben wird.

WSDL dient nicht nur dazu, dem menschlichen Nutzer zu beschreiben, wie der Webservice aufgerufen werden kann, sondern es ist gerade für die automatische Erstellung von Clients für Webservices interessant. Die Erstellung eines WSDL-Dokumentes kann ebenfalls automatisch durch Programme bereitgestellt werden.

2.4 REST-konforme Webservices

REST-konforme Webservices basieren auf dem Architekturstil „REpresentational State Transfer“ (REST), der aus der Dissertation von Roy Fielding stammt (Fielding 2000). Fielding gibt Vorschläge, wie Software-Architekturen aufgebaut werden sollen, und stellt dabei unter anderem einige Prinzipien auf, beispielsweise was beim Einsatz von Internet-Protokollen wie HTTP beachtet werden sollte. Ziel ist es dabei u.a., die Implementierung und Nutzung von verteilten, webbasierten Systemen zu vereinfachen und zu verbessern (Fielding 2000). Es handelt sich dabei aber nicht um die Definition eines Standards, sondern eher um ein Verfahren zum Bewerten von Architekturen (Richardson und Ruby 2007), es existiert keine Sammlung von genau festgelegten Richtlinien für REST-Architekturen (Burbiel 2007). Deswegen kann man sich nur auf die in der Dissertation vorgestellten Prinzipien oder auf die Interpretationen dieser Prinzipien durch verschiedene Autoren oder Entwickler beziehen (Richardson und Ruby 2007). Dadurch ist die Einordnung, ob ein Webservice REST-konform ist oder nicht, nicht immer ganz einfach und auch abhängig von der Interpretation der REST-Prinzipien von Seiten des Betrachters.

„Das zentrale Konzept von REST sind Ressourcen“ (Tilkov 2011). Eine Ressource ist alles, was wichtig genug ist, um als eigenständige Einheit referenziert zu werden. Die Referenzierung einer Ressource geschieht über einen eindeutigen URI (Uniform Resource Identifier) bzw. eine URL. So sind etwa ein Dokument, eine Zeile einer Datenbank oder das Ergebnis eines Algorithmus eine Ressource (Richardson und Ruby 2007). REST-konforme Webservices umschließen eine Ressourcenklasse und stellen eine einfache Schnittstelle bereit, über die man den Zustand der Ressourcen abfragen bzw. manipulieren kann (Löwenstein und Kraeft 2011). Bei REST-konformen Webservices werden dafür ausschließlich die Standard HTTP-Methoden eingesetzt, von denen GET, POST, PUT und DELETE die wichtigsten sind. GET wird für das Ausliefern einer Ressource verwendet, POST für das Anlegen einer neuen Ressource, PUT, um eine Ressource zu verändern und DELETE, um eine Ressource zu löschen.

„Es wird nicht für jeden Service eine eigene Schnittstelle entworfen“ (Tilkov 2011), sondern die anwendungsspezifische Funktionalität wird auf die einheitliche Schnittstelle abgebildet bzw. nur mittels der HTTP-Methoden erreicht (Tilkov 2011). Neben diesen beiden Prinzipien, Ressourcen mit eindeutiger Identifikation und der einheitlichen Schnittstelle, gibt es noch drei weitere wichtige Kernprinzipien von REST: Hypermedia, unterschiedliche Repräsentationen und statuslose Kommunikation (Tilkov 2011).

Durch Hypermedia kann der Anwendungszustand des Webservices gesteuert werden, indem der Server dem Client über Links mitteilt, welche Aktionen als nächstes möglich sind (Tilkov 2011). Außerdem meint das Prinzip Hypermedia, dass Dokumente, bzw. Ressourcen sinnvolle Verbindungen (Links) zu anderen Ressourcen enthalten sollen.

Dargestellt werden Ressourcen mit Repräsentationen. Ressourcen sind nicht die Daten an sich, sondern nur die Idee des Servicedesigns (Richardson und Ruby 2007). Fragt man eine bestimmte Ressource ab, so erhält man die Repräsentation einer Ressource, beispielsweise ein XML- und/oder HTML-Dokument. Das Prinzip der unterschiedlichen Repräsentationen besagt, dass von einer Ressource mehrere unterschiedliche Repräsentationen bereitgestellt werden können bzw. sollten. Handelt es sich bei einer Ressource beispielsweise um einen Zeitungsartikel, dann wäre eine mögliche Repräsentation dieser Ressource eine HTML-Seite. Eine andere Repräsentation des gleichen Artikels könnte z.B. ein einfaches Textdokument sein. Es wäre auch möglich, dieses Textdokument in verschiedenen Sprachen bereitzustellen. Dann wäre jede einzelne Darstellung der Ressource (des Artikels) in einer anderen Sprache eine unterschiedliche Repräsentation der Ressource.

Das letztgenannte Prinzip der statuslosen Kommunikation empfiehlt, dass die Kommunikation zwischen Client und Server grundsätzlich zustandslos erfolgt. „[A]lle Informationen, die der Server zur Verarbeitung einer Anfrage benötigt, [müssen auch] in dieser Anfrage enthalten sein“ (Tilkov und Ghadir 2006).

REST-konforme Webservices unterscheiden sich grundlegend von den anderen drei Webservice-Ansätzen, die in dieser Arbeit behandelt werden. Diese drei Ansätze sind Webservices im RPC-Stil. In einem einfachen Fall hat ein solcher Webservice eine einzige Internetadresse, unter der er zu erreichen ist. Aufgerufen wird er, indem an diese Adresse beispielsweise ein XML- oder ein SOAP-Dokument gesendet wird. In diesem Dokument sind die aufzurufende Operation und die Eingangsparameter für die Operation enthalten. Nach dem Durchlauf der Operation sendet der Webservice ein entsprechendes Antwort-Dokument an den Client zurück.

REST-konforme Webservices sind hingegen niemals nur über eine einzige Internetadresse erreichbar, sondern jede Ressource des Webservices besitzt eine eigene URL. „Genau ge-

nommen gibt es gar keine REST-Services. Es gibt nur Ressourcen, die angeboten werden“ (Bayer und Sohn 2007). Die Interaktion zwischen dem Webservice und dem Client geschieht über diese Ressourcen. Indem eine der HTTP-Methoden auf eine solche URL bzw. auf eine Ressource angewendet wird, wird der Webservice genutzt. Im Prinzip können lediglich vier Dinge mit einer Ressource gemacht werden: Sie wird angefordert, aktualisiert, gelöscht oder es wird eine neue Ressource an der URL erstellt, auf die die HTTP-Methode angewendet wird. Die gesamte Funktionalität des Webservices wird über die Interaktionen mit den Ressourcen mittels der HTTP-Methoden erreicht.

REST-konforme Webservices können theoretisch mit jeder Programmiersprache umgesetzt werden, da alle Sprachen Unterstützung für die Entwicklung von Webservern und -clients, HTTP und URLs bieten (Tilkov 2011). Daneben gibt es viele Frameworks für REST-Webservices, die die Erstellung einfach machen (Richardson und Ruby 2007). Dazu zählen beispielsweise Ruby on Rails (für die Programmiersprache Ruby) (Rails-Core-Team 2012), Django (für Python) (Django Software Foundation 2012) und Jersey (für Java) (Oracle 2012).

Der Aufbau von REST-konformen Webservices bringt bei der Einhaltung der Prinzipien einige Vorteile mit sich. Sie sind zum einen einfach und leichtgewichtig, weil der Nutzer nicht für jeden einzelnen Service besondere Spezifikationen oder Anleitungen lesen muss (Foerster et al. 2011). REST-konforme Webservices verfügen über die uniforme Schnittstelle und damit geschieht die Nutzung von jedem Service prinzipiell auf die gleiche Art. Die Bezeichnungen der einzelnen angebotenen Operationen müssen nicht bekannt sein, weil die HTTP-Methoden in Verbindung mit den Ressourcen eingesetzt werden. Weil also der Webservice aus den Ressourcen aufgebaut ist, sind auch einzelne Komponenten leicht austauschbar und der Webservice kann leicht erweitert werden. Daneben gelten REST-Webservices als gut skalierbar und sie bieten eine gute Performance. Dies wird vor allem durch die zustandslose Kommunikation erreicht, weil aufeinanderfolgende Anfragen dadurch nicht unbedingt vom gleichen System beantwortet werden müssen (Tilkov 2011). Außerdem kann eine große Anzahl von Anfragen aus dem Cache-Speicher beantwortet werden (Tilkov 2011).

Ein Nachteil von REST-konformen Webservices ist, dass kein Standard für die Webservices vorliegt. Dadurch wird u.a. die Parameterübergabe und -rückgabe erschwert, da es hier keine Vorgaben gibt, und der fehlende Standard führt dazu, dass REST oft falsch verstanden wird. Daneben ist es mit REST aufwendiger, bereits bestehende Anwendungen über das Web verfügbar zu machen, weil die Anwendungen vor der Bereitstellung mit REST erst entsprechend umgebaut werden müssen. Es muss eine sinnvolle Ressourcenstruktur geschaffen werden, und die Funktionalität muss auf die uniforme Schnittstelle abgebildet werden. Bei Webservices im

RPC-Stil muss eine bestehende Anwendung selbst kaum noch überarbeitet werden, sondern es wird eine Schnittstelle für die Anwendung entworfen.

2.5 Web Processing Services

Der Web Processing Service ist ein Standard vom OGC, dessen Version 1.0.0 im Jahr 2007 publiziert wurde (vgl. Open Geospatial Consortium 2007). WPS beschreibt eine Schnittstelle, die die Veröffentlichung, das Auffinden und die Anbindung von Geo-Prozessen für den Client vereinfachen soll (Open Geospatial Consortium 2007). In diesem Standard wird definiert, welche Anforderungen ein Webservice erfüllen und wie er aufgebaut sein muss, um als WPS im Sinne vom OGC gültig zu sein. Vorgegeben werden beispielsweise mögliche Datentypen, einzusetzende Techniken und Standards oder die zu unterstützenden Operationen. Bei letzteren handelt es sich um GetCapabilities, DescribeProcess und Execute. Diese drei Operationen liefern die Funktionalitäten von Web Processing Services und stellen ihre zentralen Elemente dar.

Die GetCapabilities-Operation liefert dem Client eine grundlegende Beschreibung bzw. Metadaten über alle Prozesse, die unter der Internetadresse des WPS erreichbar sind, denn es ist möglich, unter einer URL mehrere Prozesse bereitzustellen. Diese GetCapabilities-Operation muss über HTTP-GET implementiert werden, unter Verwendung von KVP (Key-Value pair) für die Parameterübergabe in der URL, und sie kann zusätzlich dazu auch mit HTTP-POST als XML kodiert implementiert werden (Open Geospatial Consortium 2007). Damit soll SOAP unterstützt werden (Open Geospatial Consortium 2007). Hier liegt eine wichtige Eigenschaft des WPS. Die Operationen werden alle auf eine genau vorgeschriebene Art angeboten, aber es ist dennoch möglich, zusätzlich weitere Techniken einzusetzen. Dadurch wird die Flexibilität von WPS erhöht, ohne dass der Nutzer des Webservices von der zusätzlichen Technik abhängig gemacht wird.

Die DescribeProcess-Operation kann für jeden der angebotenen Prozesse einzeln ausgeführt werden und gibt dem Client eine komplette Beschreibung des jeweiligen Prozesses. Dazu gehören Informationen zu Eingaben, Ausgaben und deren Datenformaten. Es sind drei verschiedene Datenstrukturen für die Parameter definiert: LiteralData, ComplexData und BoundingBoxData. LiteralData umfasst einfache Datentypen wie beispielsweise Integer, Float, String oder Boolean. ComplexData steht für komplexe Datenstrukturen wie z.B. GML-Dokumente (GML - Geography Markup Language), Bilder oder Binärdaten. BoundingBoxData gibt eine bestimmte Bounding Box in einem räumlichen Bezugssystem an. DescribeProcess muss, wie bei GetCapabilities, über HTTP-GET mit KVP implementiert werden und kann zusätzlich mit

HTTP-POST und XML-Kodierung der Parameter implementiert sein (Open Geospatial Consortium 2007).

Diese beiden Operationen, bzw. die Antworten auf Anfragen an die beiden eben beschriebenen Prozesse können zur automatischen Erstellung eines Clients benutzt werden, erfüllen also ähnliche Zwecke wie ein WSDL-Dokument bei SOAP-Webservices. Zusätzlich ist es möglich, ein WSDL-Dokument für die angebotenen Prozesse bereitzustellen und dessen URL in der DescribeProcess-Antwort anzugeben (Open Geospatial Consortium 2007). Hier wird wieder die Möglichkeit gegeben, WPS auch mit SOAP kompatibel zu machen bzw. umzusetzen.

Die Execute-Operation ist schließlich dafür da, einen der Prozesse des WPS auszuführen. Im Gegensatz zu den anderen beiden Operationen ist für Execute eine Implementierung mit HTTP-POST und einer Codierung der Eingangs- und Ausgabedaten in XML vorgeschrieben (Open Geospatial Consortium 2007). Somit ist sichergestellt, dass größere Eingabedateien möglich sind, da mit HTTP-GET nur eine begrenzte Menge an Zeichen zum Server geschickt werden können. Zusätzlich ist aber eine Implementation mit HTTP-GET und KVP möglich (Open Geospatial Consortium 2007). So kann eine Execute-Operation auch über die Adresszeile eines Web-Browsers ausgeführt werden.

Die Ein- und Ausgabedaten für die Execute-Operation können auch mit einer URL-Referenz angegeben werden. Dadurch kann der WPS benötigte Eingangsdaten selbstständig an der angegebenen URL abholen, ohne dass ihm diese Eingabedateien zugeschickt werden müssen, und der WPS kann Ausgabedaten eines Prozesses an einer URL hinterlegen. Dadurch sind auch asynchrone Anfragen des WPS möglich, der User muss nicht unbedingt auf die Beendigung des Prozesses warten, sondern kann die Ergebnisse des Prozesses auch zu einem späteren Zeitpunkt abholen. „Sinnvoll ist dies bei Prozessen, die lange Prozessierungszeiten benötigen und deshalb bei synchroner Anfrage anfällig sind für time outs auf der Clientseite“ (Brauner 2008).

Zur Verdeutlichung ist in Abbildung 4 der typische Ablauf bei der Nutzung eines WPS grafisch aufbereitet zu sehen:

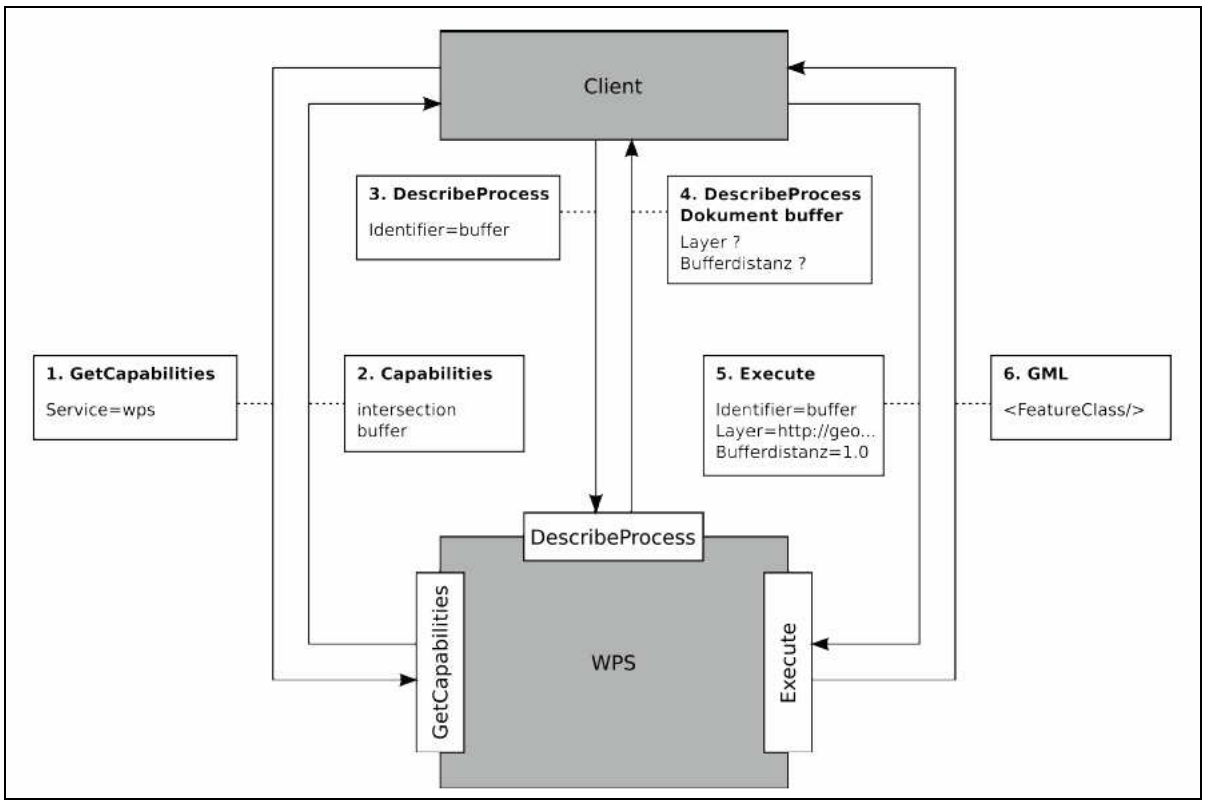

Abbildung 4: Typischer Ablauf bei der Nutzung eines WPS. Quelle: Brauner (2008)

Der WPS beginnt normalerweise, indem der Client eine GetCapabilities Anfrage an den WPS-Server sendet. Dadurch erhält er eine Übersicht aller möglichen Prozesse. In diesem Beispiel werden die beiden Prozesse „intersection“ und „buffer“ angeboten. Als nächstes fordert der Client eine genauere Beschreibung des „buffer“-Prozesses mit der Operation DescribeProcess an. Die Anfrage wird an dieselbe URL gesendet, enthalten ist hier die Information, welcher Prozess beschrieben werden soll. In der Antwort auf die DescribeProcess-Operation sind alle Angaben enthalten, um den Prozess ausführen zu können. In diesem Fall sind die nötigen Eingabeparameter für die Execute-Anfrage aufgelistet, nämlich der zu bearbeitende Layer und die Bufferdistanz. Die dafür gewünschten Eingaben werden schließlich in der Execute-Operation angegeben und zum WPS-Server gesendet. Zusätzlich ist in dem Beispiel wieder angegeben, für welchen Prozess die Execute-Operation ausgeführt werden soll. Abgeschlossen wird der gesamte Prozess durch das Senden der Antwort des WPS-Servers. In diesem Fall erhält der Client eine GML-Datei.

Für die Umsetzung von Web Processing Services bieten sich besonders Frameworks an. Hiermit ist es möglich, WPS-Prozesse gemäß dem Standard zu programmieren. Die Frameworks bieten Unterstützung bei der Erstellung aller drei WPS-Operationen und erlauben mit wenig Aufwand, vorhandenen Programm-Code mit einem WPS anzubieten. Für die Erstellung von WPS können beispielsweise die Frameworks PyWPS (für die Programmiersprache

Python) (Cepicky 2011) oder das Zoo-Projekt (für C#, Fortran, Java, Python, PHP, Perl, JavaScript) (ZOO Project 2012) genutzt werden.

Vorteile von WPS sind zum einen die Unterstützung von räumlichen Daten und Prozessen. Räumliche Daten werden auch bei forstlichen Modellen eingesetzt, und WPS bietet u.a. die Möglichkeit einfach und direkt an ein GIS-System angebunden zu werden. Daneben kann WPS durch die Unterstützung von HTTP-GET einfach aus dem Web-Browser heraus genutzt werden.

Ein Nachteil von WPS könnte unter Umständen die Abhängigkeit von den Frameworks sein. Ohne die Frameworks wird die Umsetzung von WPS wesentlich komplizierter, denn sie nehmen dem Nutzer viel Arbeit ab, z.B. hinsichtlich der Einhaltung des Standards oder der Erstellung der Schnittstellenbeschreibung. Hinsichtlich der Frameworks ist der User aber unter Umständen in Bezug auf die Programmiersprache beschränkt. Es werden zwar die gängigsten Programmiersprachen unterstützt, jedoch ist ein Framework für eine weniger gängige Sprache u.U. nicht verfügbar.

3. Implementierung

In diesem Kapitel werden die vorgestellten Ansätze für Webservices praktisch angewendet. Dies soll dazu beitragen, das Verständnis anhand von praktischen Beispielen zu vertiefen und die Eignung der Ansätze für die Modellanbindung zu beleuchten. Alle hier gezeigten Ansätze wurden auf der Linux-Distribution OpenSUSE 11.4 (Novell 2011) erstellt. Für die Implementierungen der Webservices wurde zum einen die Programmiersprache Java verwendet, weil es sich hierbei um eine sehr weit verbreitete Programmiersprache handelt. Nach Tiobe Software (2012) ist Java die in den letzten Jahren am meisten eingesetzte Programmiersprache. Zum anderen wurde Python für die Programmierung verwendet, da es sich hier um eine Skriptsprache handelt, die in großem Umfang im Web-Umfeld eingesetzt wird, gut mit Linux zusammen verwendet werden kann und nach Tiobe Software (2012) derzeit eine der beiden am weitesten verbreiteten Skriptsprachen ist. Auf den Einsatz von weiteren Programmiersprachen wurde verzichtet, da dies über den Umfang der Arbeit hinausgehen würde.

Im Folgenden werden die Implementierungen von einfachen Beispielmodellen mit den vier Webservice-Ansätzen vorgestellt. Dabei werden für jeden der Ansätze verschiedene Modelle zur Anwendung gebracht, um verschiedene Aspekte bei der Webservice-Programmierung zeigen zu können. Um die verschiedenen Ansätze besser miteinander vergleichen zu können, wurde mit jedem der vier Ansätze das Beispielmodell (Oberhöhenmodell) aus dem folgenden Kapitel angebunden. Die dafür erstellten Programmcodes für den XML-RPC sind in Kapitel

3.1 zu finden, für die anderen Ansätze sind sie im Anhang enthalten (SOAP-Webservice: Kapitel A, REST-Webservice: Kapitel C, WPS: Kapitel E).

3.1 Modellanbindung mit XML-RPC

Um einen RPC mit XML-RPC zu realisieren, müssen ein Server und ein Client programmiert werden. Der Server stellt die durchführbaren Operationen bereit, im Rahmen dieser Arbeit also den Aufruf des Simulationsmodells. Der Client wird benötigt, um das Modell über das Internet auf dem Server aufrufen zu können. Dabei müssen dem Server die Eingabeparameter und die aufzurufende Funktion übermittelt werden. Die Implementierungen werden in Python und in Java durchgeführt, um zeigen zu können, dass XML-RPC mit in verschiedenen Sprachen programmierten Client und Server möglich ist.

Als Beispielmodell für den XML-RPC soll ein einfaches Höhenwachstumsmodell verwendet werden, welches die Oberhöhe eines gleichaltrigen Fichtenbestandes aus dem Alter und der Oberhöhenbonität mit der folgenden Modellgleichung nach von Gadow (2004) schätzt.

$$H = SI \cdot \left(\frac{1 - \exp(-0.0006 \cdot SI \cdot A)}{1 - \exp(-0.0006 \cdot 100 \cdot A)} \right)^{1.507}$$

H = Oberhöhe

SI = Oberhöhenbonität

A = Alter

Gleichung 1: Wuchsmodell für die Oberhöhe eines gleichaltrigen Fichtenbestandes.

3.1.1 XML-RPC mit Python

Für die Implementierung des XML-RPC in Python wird die Python-Version 2.7 (Python Software Foundation 2011) verwendet. Das hier aufgezeigte Beispiel ist angelehnt an Kapitel 20.7 von Kaiser and Ernesti (2007), die eine einfache Möglichkeit zur Nutzung von XML-RPC aufzeigen. Für die Erstellung des Servers wird das Python-Modul *SimpleXMLRPCServer* (Python Software Foundation 2010) eingesetzt, welches eine einfache Serverumgebung in Python zur Verfügung stellt (Kaiser und Ernesti 2007). Das folgende Programm in Listing 2 zeigt die Umsetzung des Servers für das angesprochene Beispielmodell:

```
import math
from SimpleXMLRPCServer import SimpleXMLRPCServer

def oberhoehe(A, SI):
     alpha = -0.0006
     delta = 1.507
     A = float(A)
     SI = float(SI)
     oberhoehe = SI * math.pow(((1 - math.exp(alpha * SI * A)) /
          (1 - math.exp(alpha * 100 * A))), delta)
     return oberhoehe

server = SimpleXMLRPCServer(('localhost', 8080))
server.register_function(oberhoehe)
server.serve_forever()
```

Listing 2: Programmcode XML-RPC-Server in Python

Nach dem Import der notwendigen Module wird die Modellfunktion definiert, welche der Server später anbieten soll. Anschließend wird eine Instanz der Klasse *SimpleXMLRPCServer* erzeugt, die alle notwendigen Funktionen für einen lauffähigen XML-RPC-Server bereitstellt. Zur Instanziierung sind die IP-Adresse und der Port, an den der Server gebunden werden soll, die einzigen zwingenden Parameter (Kaiser und Ernesti 2007). Danach findet die Registrierung der Modellfunktion mit Hilfe der *register_function* Methode für den RPC-Aufruf statt. Abschließend wird die Methode *serve_forever* aufgerufen, um den Server für die Beantwortung einer unbestimmten Anzahl an Anfragen zu starten (Kaiser und Ernesti 2007).

Die Klasse *SimpleXMLRPCServer* übernimmt sämtliche Aufgaben auf der Serverseite, die im Theoriekapitel zum XML-RPC genannt wurden, ohne dass dafür zusätzlicher Programmieraufwand nötig ist: Nach dem Starten ist der Server bereit, an die angegebene Adresse geschickte HTTP-POST-Anfragen für einen XML-RPC entgegenzunehmen und das enthaltene XML-Dokument an den XML-RPC Listener weiterzureichen. Die Implementierung des XML-RPC Listeners wird hier ebenfalls von der Klasse *SimpleXMLServer* übernommen, ohne dass der Programmierer zusätzliche Arbeit leisten muss. Alle Aufgaben, die der XML-RPC Listener ausführt, nämlich das Extrahieren der Informationen aus dem XML-Dokument und das Aufrufen der gewünschten Operation mit den übermittelten Parametern, bleiben bei der Programmierung im Verborgenen. Abschließend sorgt die Klasse *SimpleXMLRPCServer* dafür, dass die Rückgabewerte des Funktionsaufrufs in ein XML-Dokument geschrieben werden und als Antwort auf die HTTP-POST-Anfrage zurückgeschickt werden.

Der Server ist also mit wenigen Programmzeilen bereit für den Einsatz, und fast alle wichtigen Schritte des XML-RPCs auf der Serverseite werden automatisch ausgeführt. Um ihn nutzen zu können, wird noch der Client benötigt. Die Implementierung eines Clients, der die serverseitige Modellfunktion aufruft, besteht aus folgenden drei Zeilen (Listing 3)**Fehler! Verweisquelle konnte nicht gefunden werden.**:

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy('http://localhost:8080')
print proxy.oberhoehe(100, 39.5)
```

Listing 3: Programmcode XML-RPC-Client in Python

Als erstes wird das nötige Modul *xmlrpclib* aus der Standardbibliothek (Kaiser und Ernesti 2007) eingebunden. Dieses Modul enthält die Klasse *ServerProxy*, welche für den Client eingesetzt wird. In der zweiten Zeile wird eine Instanz dieser Klasse erzeugt, der Adresse und Port des XML-RPC-Servers übergeben werden müssen. Nun können die verfügbaren Funktionen aufgerufen werden. In diesem Beispiel wird die Funktion *oberhoehe* mit zwei Eingabeparametern ausgeführt und das Ergebnis ausgegeben.

Das Client-Programm führt alle Aufgaben auf der Clientseite aus, die im Theoriekapitel zum XML-RPC aufgezählt werden: Die erzeugte Instanz der Klasse *ServerProxy* startet den XML-RPC, indem eine HTTP-POST-Anfrage an die angegebene Serveradresse gesendet wird. Diese Anfrage enthält ein XML-Dokument mit den Eingabeparametern und dem Funktionsnamen, wie es in Listing 4**Fehler! Verweisquelle konnte nicht gefunden werden.** dargestellt ist.

```
<?xml version="1.0"?>
<methodCall>
  <methodName>oberhoehe</methodName>
    <params>
      <param>
          <value><int>50</int></value>
      </param>
      <param>
          <value><double>39.5</double></value>
      </param>
    </params>
</methodCall>
```

Listing 4: Inhalt der HTTP-POST-Anfrage eines XML-RPC

Hat der Server die Anfrage bearbeitet, empfängt die Instanz der Klasse *ServerProxy* das XML-Dokument aus der HTTP-POST-Antwort und liefert die extrahierten Rückgabewerte zurück. Ein Beispiel für diesen Aufruf zeigt Abbildung 5:

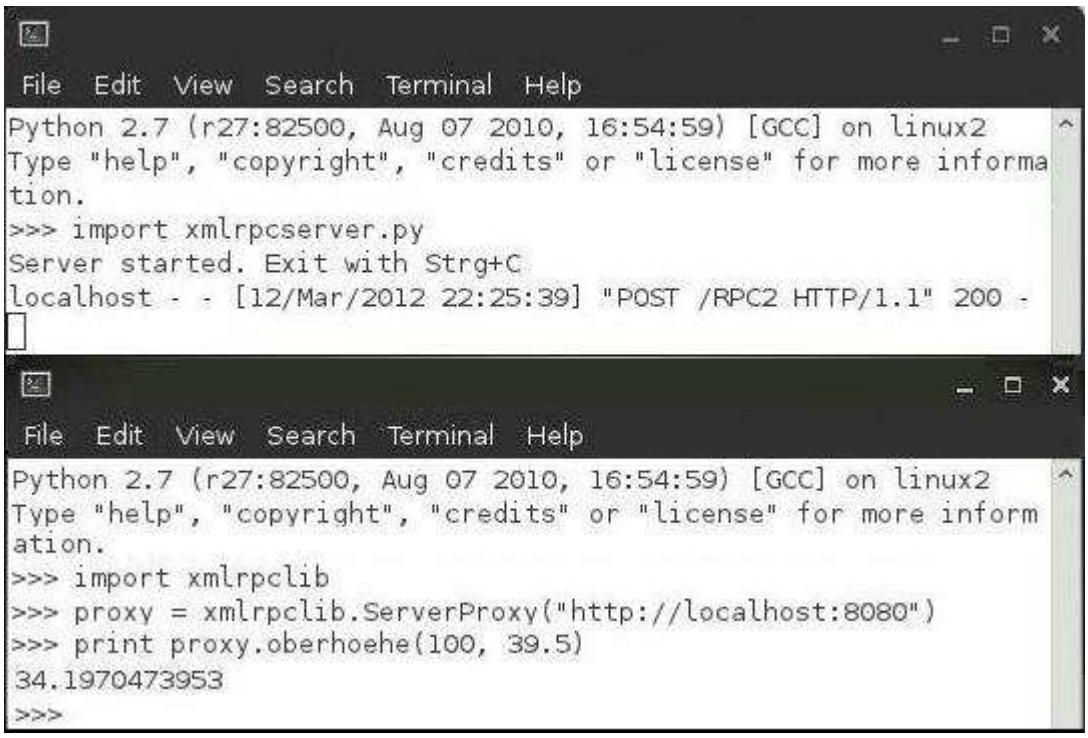

Abbildung 5: Ausführung des XML-RPC

Im oberen Fenster wurde der XML-RPC-Server aus Listing 2 gestartet. Das untere Fenster zeigt den XML-RPC-Client, welcher den XML-RPC angestoßen und das Ergebnis des XML-RPC in der Kommandozeile ausgegeben hat. Der Server hat diese Anfrage dokumentiert, was in der letzten Zeile des oberen Fensters angezeigt wird. Darin enthalten sind die HTTP-Methode, welche den Server erreicht hat (HTTP-POST), und der HTTP-Statuscode der Anfrage (200 - Anfrage erfolgreich bearbeitet).

Neben der hier vorgestellten Möglichkeit können Client und Server auch in unterschiedlichen Sprachen programmiert werden, was im folgenden Kapitel gezeigt wird.

3.1.2 XML-RPC Aufruf mit einem Java-Client

Die eingesetzte Software für einen XML-RPC in Java sind das Java JDK 6 (Oracle 2011) und Apache XML-RPC Version 3 (Apache Software Foundation 2010). Hierbei handelt es sich um eine Implementierung des XML-RPC für die Programmiersprache Java. Die hier vorgestellte Umsetzung des XML-RPC mit Java orientiert sich an Kapitel 6 von Abts (2010). Um den Python-XML-RPC-Server aufzurufen, der im vorhergehenden Kapitel beschriebenen wurde, soll ein Java-Client implementiert werden. Dafür werden die beiden Klassen *XmlRpcClient* und *XmlRpcClientConfigImpl* der Apache XML-RPC-Implementierung eingesetzt. Zusätzlich wird im Programmcode das Packet Java.net eingebunden, um eine URL verwenden zu können. Der folgende Programmcode in Listing 5**Fehler! Verweisquelle konnte nicht gefunden werden.** zeigt den Java-Client, mit dem der Python XML-RPC-Server aufgerufen werden kann:

```
import org.apache.xmlrpc.client.XmlRpcClient;
import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;
import java.net.URL;


public class JavaClient {
  public static void main(String args[]) throws Exception {
    XmlRpcClient client = new XmlRpcClient();
    XmlRpcClientConfigImpl confi = new XmlRpcClientConfigImpl();
    confi.setServerURL(new URL("http://localhost:8080"));
    client.setConfig(confi);
    Object[] params = {100, 36.5};
    System.out.println(client.execute("oberhoehe", params));
  }
}
```

Listing 5: Programmcode XML-RPC-Client in Java

In diesem Beispiel wird als erstes eine Instanz der Klasse *XmlRpcClient* erstellt. Von dieser Klasse wird später die *execute* Methode benötigt, um den XML-RPC zu starten. Vorher muss die erzeugte Instanz noch konfiguriert werden. Diese Konfiguration besteht darin, die URL und den Port des XML-RPC-Servers zu übermitteln. Um die *execute*-Methode aufrufen zu können, müssen neben dem Funktionsnamen noch die Parameter übergeben werden. Dafür wird hier ein Objekt-Array erzeugt, in dem alle zu übergebenden Parameter enthalten sind. Diese Art der Parameterübergabe ist durch die genutzte XML-RPC Apache Implementierung vorgegeben. Ist der Python-Server, wie im vorhergehenden Kapitel beschrieben, gestartet worden, kann nun mit der Ausführung des Java-Clients auf die Funktionen des Servers zugegriffen werden.

Die Programmierung des Java-Clients unterscheidet sich nur in wenigen Punkten vom zuvor gezeigten Python-Client. Auch hier werden die gesamten Schritte des XML-RPC von einer Klasse übernommen und bleiben für den Programmierer unsichtbar. Der Ablauf, der durch das Aufrufen des Java-Clients gestartet wird, ist grundsätzlich derselbe wie bei dem Python-Client. Das Beispiel zeigt, dass der XML-RPC Interoperabilität zwischen verschiedenen Programmiersprachen ermöglicht, indem die auszutauschenden Daten mit XML übermittelt werden.

3.1.3 XML-RPC-Server mit Java

Abschließend soll ein Java XML-RPC-Server mit einem Python-Client aufgerufen werden. In der folgenden Klasse (Listing 6) ist analog zu dem Python-Server die Oberhöhenfunktion verfügbar:

```
public class Oberhoehe {
  public double getOberhoehe(int A, double SI) {
    double alpha = -0.0006;
    double delta = 1.507;
    double oberhoehe = SI * Math.pow(((1 - Math.exp(alpha * SI
        *(double)A)) / (1 - Math.exp(alpha * 100 * (double)A))),
        delta);
    return oberhoehe;
  }
}
```

Listing 6: Programmcode für das Modell Oberhöhe in Java

Um die Methoden der Klasse *Oberhoehe* für einen XML-RPC verfügbar zu machen, wird die ausführbare Java-Klasse *ServerModell* gemäß Listing 7 erstellt:

```
import org.apache.xmlrpc.server.PropertyHandlerMapping;
import org.apache.xmlrpc.server.XmlRpcServer;
import org.apache.xmlrpc.webserver.WebServer;


public class ServerModell {
  public static void main(String[] args) throws Exception {
    PropertyHandlerMapping phm = new PropertyHandlerMapping();
    phm.addHandler("oberhoehe", Oberhoehe.class);
    WebServer webServer = new WebServer(8080);
    XmlRpcServer server = webServer.getXmlRpcServer();
    server.setHandlerMapping(phm);
    webServer.start();
  }
}
```

Listing 7: Programmcode XML-RPC-Server in Java

Für den XML-RPC-Server werden hier die Klassen *WebServer* und *XmlRpcServer* aus dem Apache XML-RPC-Paket eingesetzt. *WebServer* ist ein einfacher HTTP-Server, der speziell für XML-RPC-Anfragen geeignet ist, die Klasse *XmlRpcServer* verarbeitet die XML-RPC-Anfragen (Abts 2010). Zuvor wird mit der Methode *addHandler* der Klasse *PropertyHandlerMapping* ein Handler hinzugefügt. Bei einem Handler handelt es sich um eine entfernt aufrufbare Methode, welche anschließend mit der Methode *setHandlerMapping* für die *XmlRpcServer*-Instanz registriert wird (Abts 2010).

Im Gegensatz zu dem vorher erzeugten Python-Server muss ein Handler aus der aufzurufenden Methode erzeugt werden. Dies geschieht bei Python intern in der für den XML-RPC-Server eingesetzten Klasse. Ansonsten ähneln sich die Programme stark - in beiden Programmiersprachen sorgen bereitgestellte Klassen für eine einfache Erstellung des Servers. Nach dem Starten des Servers ist die bereitgestellte Funktion verfügbar. Aufgerufen werden soll sie nun mit dem schon bekannten Python-Client, der nur leicht abgeändert werden muss (vgl. Listing 8):

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy('http://localhost:8080/')
print proxy.oberhoehe.getOberhoehe(100, 36.5)
```

Listing 8: Programmcode XML-RPC-Client in Python für den Aufruf des Java XML-RPC-Servers

3.1.4 Fazit zur Implementierung von XML-RPC

In dem hier gezeigten Beispiel wird lediglich ein sehr einfaches forstliches Modell mit einem XML-RPC verfügbar gemacht. Die Anbindung komplexerer Modelle würde im Grunde genommen aber auf die gleiche Weise verlaufen. Im Unterschied zu dem hier genutzten Modell könnten aber Schwierigkeiten im Zusammenhang mit den nutzbaren Datentypen auftreten. Bei den Eingabeparametern handelt es sich in diesem Beispiel um die beiden einfachen Datentypen Integer und Double. Modelle mit Ein- und Ausgaben mit anderen als denen für den XML-RPC möglichen Datentypen müssten ggf. vor dem Einsatz mit XML-RPC an die verfügbaren Datentypen angepasst werden. Außerdem ist mit XML-RPC nur ein Rückgabewert möglich. Bei Modellen mit mehr als einem Rückgabewert besteht noch die Möglichkeit, mehrere Rückgabewerte in Arrays zu speichern und zurückzugeben. Die Rückgabe von ganzen Text- oder XML-Dokumenten ist mit einem XML-RPC nicht möglich.

3.2 Modellanbindung mit SOAP

In diesem Kapitel wird die Anbindung des Modells Treegross (Tree Growth Open Source Software), ein Modell zur Analyse und Vorhersage der Entwicklung von Waldbeständen, mit einem SOAP-Webservice vorgestellt. Treegross ist ein bekanntes und frei verfügbares forstliches Wachstumsmodell, welches wesentlich komplexer ist als das zuvor gezeigte Oberhöhenmodell. Damit soll gezeigt werden, dass mit Webservices auch komplexere Modelle einfach angebunden werden können. Da es sich bei Treegross um ein Java-basiertes Softwarepakt handelt, wird die Implementierung des Webservices mit der Programmiersprache Java durchgeführt.

Treegross besteht aus unterschiedlichen Paketen, welche für unterschiedliche Bereiche der Waldwachstumsmodellierung eingesetzt werden können, wie z.B. ein Paket für die Bestandesbehandlung oder die Holzernte. Jedes der Pakete beinhaltet verschiedene Java-Klassen, welche die eigentliche Funktionalität von Treegross darstellen. Einige dieser Klassen können zum einen einzeln genutzt werden, um bestimmte Simulationen durchzuführen, wie z.B. Berechnung von Höhenkurven oder zur Volumenberechnung, und zum anderen können die von Treegross bereitgestellten Klassen zu einem Modell zusammengefügt werden (Nagel 2003). Das hier verwendete Modell setzt sich aus den von Treegross bereitgestellten Komponenten zur Modellierung eines gleichaltrigen Waldbestandes mit Bestandesgenerierung, -wachstum

und -behandlung zusammen. Das Bestandeswachstum erfolgt in Schritten von jeweils fünf Jahren, mit einer Durchforstung zwischen zwei Wachstumsperioden. Eingangsparameter sind die Baumart und die Anzahl der zu simulierenden Perioden sowie einige Bestandeskennzahlen zu Beginn der Wachstumssimulation: Alter, Grundfläche, Mittelhöhe und Mitteldurchmesser. Als Ergebnis liefert das Modell nach Ablauf der Bestandessimulation die neu berechneten Bestandeskennzahlen zurück. Einen genaueren Einblick in den Modellablauf liefert der Modellcode im Anhang (Kapitel B). Das erstellte TreeGross-Modell wird mit der Methode „runTG“ aus der Klasse „TG_scheduler“ ausgeführt. Die für das Modell eingesetzten Treegross-Pakete können heruntergeladen werden bei Treegross Development (2003).

3.2.1 SOAP-Webservice mit Apache Axis 2

Für die Erstellung des Webservices wird die Java Version 6 in der Entwicklungsumgebung Eclipse Indigo (Eclipse Foundation 2010) zusammen mit dem Webservice Toolkit Apache Axis 2 (Apache Software Foundation 2011) verwendet. Als Server für den Webservice wird Apache Tomcat Version 6 (Apache Software Foundation 2012a) genutzt. Apache Axis 2 ist eine SOAP-Engine zum Erstellen von SOAP-Servern und -Clients, welche in C++ und Java implementiert ist.

Bevor der Webservice erstellt werden kann, muss Apache Axis 2 und Apache Tomcat in der Entwicklungsumgebung Eclipse eingerichtet werden. Dies ist für die Erstellung des SOAP-Webservices wichtig, weil diese größtenteils automatisch von Axis 2 aus Eclipse heraus durchgeführt wird. Hilfreich dabei und bei der Erstellung des Webservices ist der Eintrag „Creating Web services with the Apache Axis2 runtime environments“ der Eclipse Dokumentation (Eclipse Foundation 2011). Nach dem Einrichten von Axis 2 in Eclipse ist es möglich, auf alle Funktionalitäten, die Axis 2 liefert, aus Eclipse heraus zuzugreifen. Durch diese Kombination ist die Erstellung von SOAP-Webservices ohne ein tiefergehendes Verständnis des verwendeten Toolkits und der Konzepte von SOAP-Webservices möglich. Im Folgenden sollen nun die Grundzüge der Webservice-Erstellung mit dem Toolkit dargestellt werden.

Die vom Webservice angebotenen Operation bzw. der Server des Webservices besteht aus der Methode *runws* der Klasse *TreeGrossWS* aus Listing 9**Fehler! Verweisquelle konnte nicht gefunden werden.**:

```
package treegrossws;

public class TreeGrossWS {

  public double[] runws(int inAlter, int inArt, double inHG,
      double inDG, double inG, int inPerioden){

    TG_scheduler tgm = new TG_scheduler();
    return tgm.runTG(inalter, inart, inhg, indg, ing, inperioden);
  }
}
```

Listing 9: Programmcode der mit dem SOAP-Webservice angebotenen Methode

Aus dieser Klasse kann von Eclipse und Axis 2 der Webservice automatisch erzeugt werden, wenn sie in ein Dynamisches Webprojekt in Eclipse eingebunden wird. Das eigentliche Modell wird aus dieser Methode heraus aufgerufen, um eine bessere Übersichtlichkeit zu gewährleisten. Der Name der hier vorgestellten Klasse *TreeGrossWS* stellt später die Bezeichnung des Webservices dar, und die angebotene Operation des Webservices erhält den Namen der Methode (*runTG*). Die Ein- und Ausgabedaten der Methode werden ebenfalls von Eclipse und Axis 2 für den Webservice aus dieser Klasse übernommen. Die Eingangsdaten des Webservices bestehen also aus den drei Ganzzahlen (Integer) *inAlter* (Bestandesalter), *inArt* (Baumart) und *inPerioden* (Anzahl simulierter Perioden) und den drei Gleitkommazahlen (Double) *inHG* (Mittelhöhe des Bestandes), *inDG* (Mitteldurchmesser des Bestandes) und *inG* (Bestandesgrundfläche).

Für die Erzeugung des Webservices in Eclipse ist es lediglich nötig, den Menüpunkt für die Erstellung eines neuen Webservices auszuwählen und anschließend diese Klasse (*TreegrossWS*) anzugeben. Danach wird der Webservice automatisch auf dem Webserver Apache Tomcat eingerichtet sowie gestartet und ist damit bereit, Anfragen entgegenzunehmen. Bei dem eingesetzten Transportprotokoll handelt es sich um HTTP. Weiterhin muss sich der Nutzer nicht um die SOAP-Nachrichten und das WSDL-Dokument an sich kümmern, da auch dies alles voll automatisch erstellt wird.

3.2.2 Clienterstellung mit Apache Axis 2

Für die Clienterstellung in Eclipse existiert ebenfalls ein Menüpunkt, der für den Webservice-Client genutzt werden kann. Um den Client zu erstellen, muss die URL des WSDL-Dokumentes des zu nutzenden Webservices angegeben werden. Anschließend wird die Clienterstellung automatisch durchgeführt. Als Ergebnis erstellen Axis 2 und Eclipse die zwei Java Klassen *TreegrossSOAPCallbackHandler* und *TreegrossSOAPStub*. Die Klasse für den Callback-Handler kann verwendet bzw. erweitert werden, um asynchrone Aufrufe des Webservices zu ermöglichen. In diesem Beispiel ist sie nicht relevant. Die Klasse *TreegrossSO-*

APStub liefert die für den Client nötigen Funktionalitäten und ist damit das eigentliche Kernelement des erstellten Clients. Diese Funktionalitäten werden in der Klasse *TreegrossClient* (Listing 10) verwendet, um den zuvor erstellten Webservice zu nutzen.

```
package treegrossws;

import java.rmi.RemoteException;
import treegrossws.TreeGrossWSStub.Runws;
import treegrossws.TreeGrossWSStub.RunwsResponse;

public class TreegrossClient {

  public static void main(String[] args) throws RemoteException {

    TreeGrossWSStub stub = new TreeGrossWSStub();
    Runws run = new Runws();
    run.setInalter(60);
    run.setInart(110);
    run.setIndg(17.5);
    run.setIng(18.6);
    run.setInhg(22.7);
    run.setInperioden(2);
    RunwsResponse res = stub.runws(run);
    double[] result = new double[4];
    result = res.get_return();

    for(int i=0; i<4; i++){
      System.out.println(result[i]);
    }
  }
}
```

Listing 10: Programmcode des Clients für den SOAP-Webservice in Java

Zuerst wird ein Stub für den Webservice instanziiert. Der Stub ist der Programmcode oder Programmteil, welcher für die Kommunikation mit dem Programmcode des Webservices zuständig ist. Anschließend wird eine Instanz der Klasse *Runws*, deren Name sich von der durch den Webservice angebotenen Operation ableitet, erzeugt und für die Parameterübergabe genutzt. Mit Hilfe dieser Instanz werden die Eingangsparameter für den Webservice zusammengefasst. Nun wird noch eine Instanz der Klasse *RunwsResponse* benötigt, welche die Antwort des Webservices entgegennehmen kann. Ausgeführt wird der Webservice, indem mittels des Stubs die Methode *runws* mit den Eingangsparametern ausgeführt wird, welche die Ergebnisse des Webservices an die Instanz der Klasse *RunwsResponse* übergibt.

Wie bei dem zuvor erstellten Server verläuft also auch die Erstellung des Client-Programms fast vollständig automatisiert. Die Handhabung der eigentlichen Bestandteile des Webservices, die Verwendung des SOAP-Protokolls und des WSDL-Dokuments, wird dem Programmierer des Webservice-Clients komplett abgenommen.

3.2.3 Erstellung eines Python-Clients

Um das Kapitel zur Erstellung eines SOAP-Webservices abzuschließen, soll an dieser Stelle noch die Erstellung eines SOAP-Webservice-Clients mit der Python-Bibliothek Suds (Ortel et al. 2012) vorgestellt werden. Hierbei handelt es sich um eine Bibliothek, mit dessen Hilfe Clients für Webservices direkt aus einem WSDL-Dokument erzeugt werden können. In Listing 11 ist der dafür eingesetzte Programmcode dargestellt.

```
from suds.client import Client

def usews(inAlter, inArt, inHg, inDg, inG, inPerioden):
  url = 'http://localhost:8080/SOAPTreegross/services/TreegrossWS?wsdl'
  client = Client(url)
  print client
  result = client.service.runws(inAlter, inArt, inHG, inDG, inG,
      inPerioden)
  return result

print usews(60, 110, 18.6, 22.7, 17.5, 1);
```

Listing 11: Programmcode des Python-Clients für den SOAP-Webservice

Aus der Suds-Bibliothek wird die Klasse *Client* importiert, der innerhalb der hier gezeigten Funktion *usews* die URL des WSDL-Dokumentes des Treegross-Webservices übergeben wird. Anschließend kann mit dem Client die Methode *runws* des Webservices ausgeführt werden.

Dieses Beispiel zeigt, dass die Client-Erstellung für SOAP-Webservices unabhängig von der Programmiersprache durchgeführt werden kann. Ermöglicht wird dies durch das WSDL-Dokument des zu nutzenden Webservices. In diesem Beispiel müssen zusätzlich zur URL des WSDL-Dokumentes der Name der angebotenen Operation sowie deren Ein- und Ausgaben bekannt sein. Dafür kann die Textrepräsentation der erzeugten Instanz der Klasse *Client* herangezogen werden, die diese Informationen enthält und mit dem Befehl *print client* in der Konsole ausgegeben werden kann.

3.2.4 Fazit zur Implementierung von SOAP-Webservices

Die Implementierung des SOAP-Webservices mit Apache Axis 2 in Eclipse kann fast vollständig automatisch durchgeführt werden. Es werden kaum Programmierkenntnisse und Kenntnisse über SOAP-Webservices vorausgesetzt. Auch der Programmcode des erzeugten Webservices bleibt dabei im Verborgenen und muss nicht zusätzlich modifiziert werden. Aus diesen Gründen ist die Implementierung und Nutzung eines Webservices relativ einfach möglich. Dadurch kann die Implementierung aber kaum zu einem besseren Verständnis des zugrundeliegenden Konzeptes und des internen Ablaufes von SOAP-Webservices beitragen.

3.3 Modellanbindung mit REST

Um die Bereitstellung eines Modells mit einem REST-Webservice zu demonstrieren, wird in diesem Kapitel ein Webservice für die Anpassung einer einfachen linearen Regression erstellt. Dieses Regressionsmodell unterscheidet sich von dem Oberhöhenmodell, für das ebenfalls ein REST-Webservice erstellt wurde (Anhang: Kapitel C). Durch den größeren Modellumfang können anhand des Regressionsmodells die Prinzipien von REST-Webservices besser erläutert werden. Die Datengrundlage der Regression bilden der BHD und das Alter einer beliebigen Anzahl von Bäumen. An diese Daten soll dann das lineares Modell $y = \beta_0 + \beta_1 \cdot x$ mit dem BHD als abhängige und dem Baumalter als unabhängige Variable angepasst werden. Die Eingangsdaten können von dem Nutzer des Webservices vorgegeben werden. Er erhält die Möglichkeit, Beobachtungen, in diesem Fall Bäume mit einem bestimmten Alter und BHD, in einer Liste zusammenzustellen. Anschließend soll der Nutzer dann auf Grundlage dieser Eingangsdaten eine Modellanpassung durchführen können. Die für den hier vorgestellten REST-Webservice erstellten Quellcodes sind im Anhang (Kapitel D) enthalten.

3.3.1 Struktur des REST-Webservices

REST-konforme Webservices sind eine Zusammenstellung von Ressourcen. Bei der Erstellung eines Webservices müssen deswegen zuerst sinnvolle Ressourcen für den Webservice identifiziert und mit einer eindeutigen URL versehen werden. Die Ressourcen sind die Interaktionspunkte des Webservices für den Client bzw. den Nutzer. Den schematischen Aufbau des Webservices für die lineare Regression mit seinen Ressourcen zeigt Abbildung 6. Die Pfeile in der Abbildung zeigen Links zwischen den Ressourcen an.

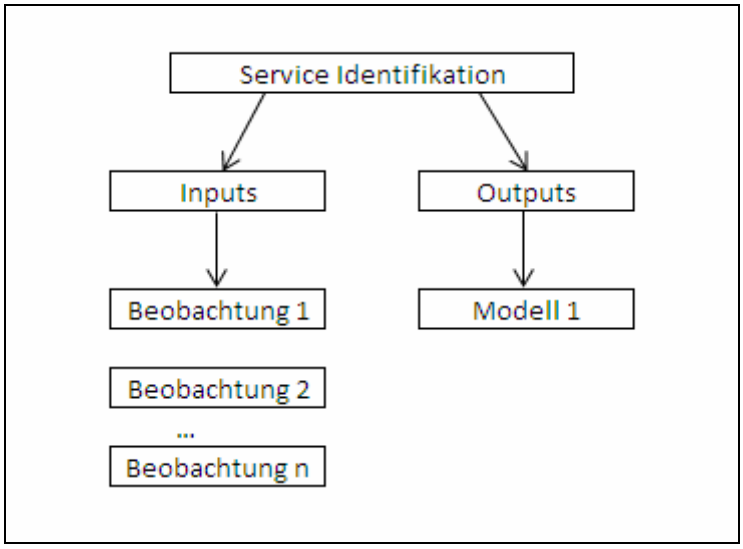

Abbildung 6: Aufbau des REST-Webservices für eine lineare Regression.

Der REST-Webservice besteht aus fünf verschiedenen Ressourcen-Klassen: Dem Einstiegspunkt des Webservices, den beiden Ressourcen-Klassen für die Eingaben (Inputs) und die Ausgaben (Outputs) und den einzelnen Beobachtungs- und Modell-Ressourcen.

Der Einstiegspunkt des Webservices ist die erste Ressource, die über die URL des Webservices erreichbar ist. Wie bei allen anderen Ressourcen des Webservices kann mit HTTP-GET eine Repräsentation der Ressource abgefragt werden. Eine GET-Anfrage mit dem Browser liefert die folgende Repräsentation der Ressource, dargestellt in Abbildung 7:

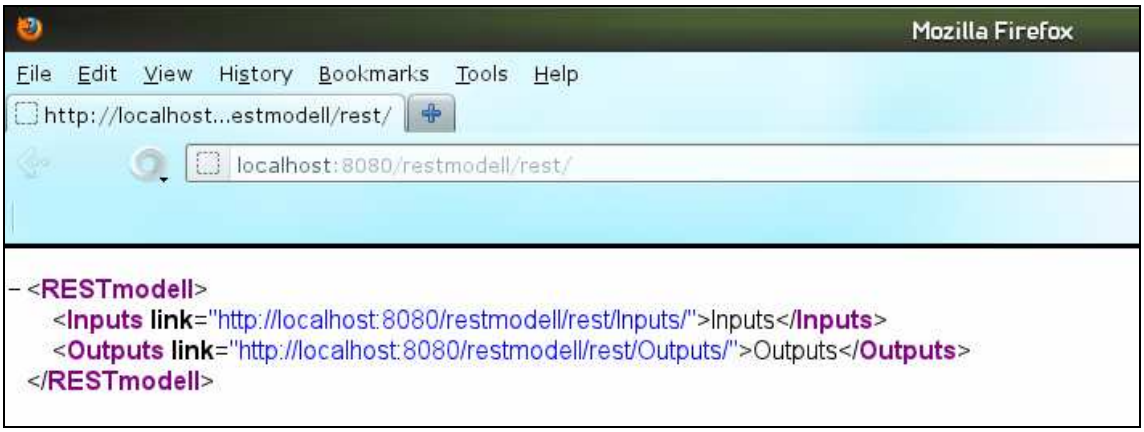

Abbildung 7: Eintrittspunkt des REST-Webservices

Von dieser Ressource aus sind zum einen die Eingaben (die Beobachtungen) für die Modellanpassungen über einen Link erreichbar und zum anderen eine Übersicht über die Modelle, die daran angepasst werden können.

Folgt man dem Link für die Modellinputs, bzw. führt eine HTTP-GET-Anfrage auf die URL des Links aus, erhält man eine Übersicht über alle Beobachtungen, an die später die lineare Regression angepasst werden kann. Alle diese Beobachtungen zusammengenommen stellen eine identifizierbare Ressource dar. Die XML-Repräsentation dieser Ressource ist in Abbildung 8 dargestellt:

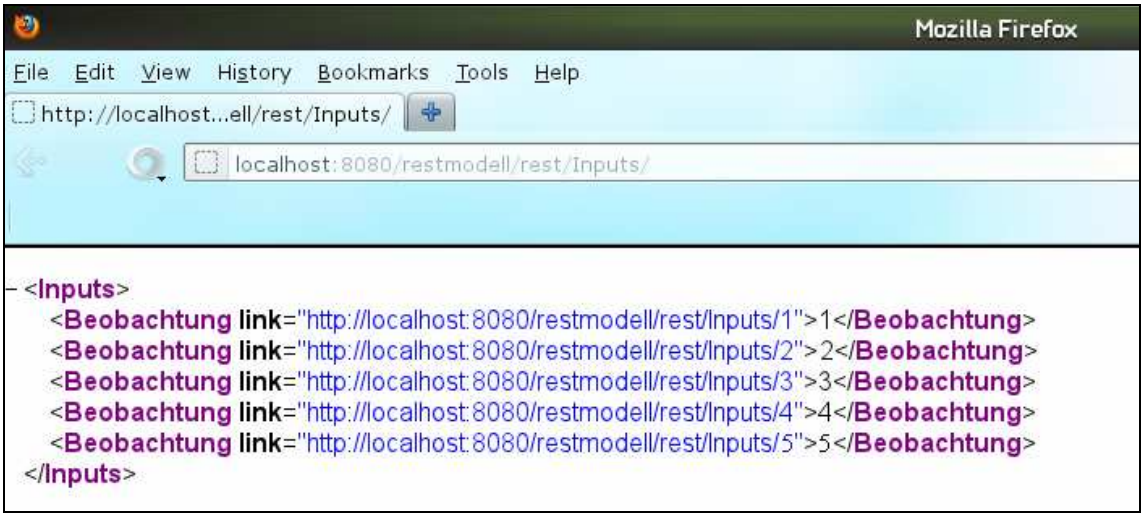

Abbildung 8: Übersicht über die Input-Ressourcen für die Modellanpassung des REST-Webservices

Von dieser Ressource aus sind über Links die einzelnen Beobachtungen erreichbar, welche jede für sich wiederum eine eigene Ressource darstellen. Eine XML-Repräsentation einer der Beobachtungs-Ressourcen erhält man, indem eine HTTP-GET-Anfrage auf eine der dargestellten URL ausgeführt wird. Abbildung 9 zeigt die Antwort auf eine solche Anfrage bzw. eine Beobachtungs-Ressource an:

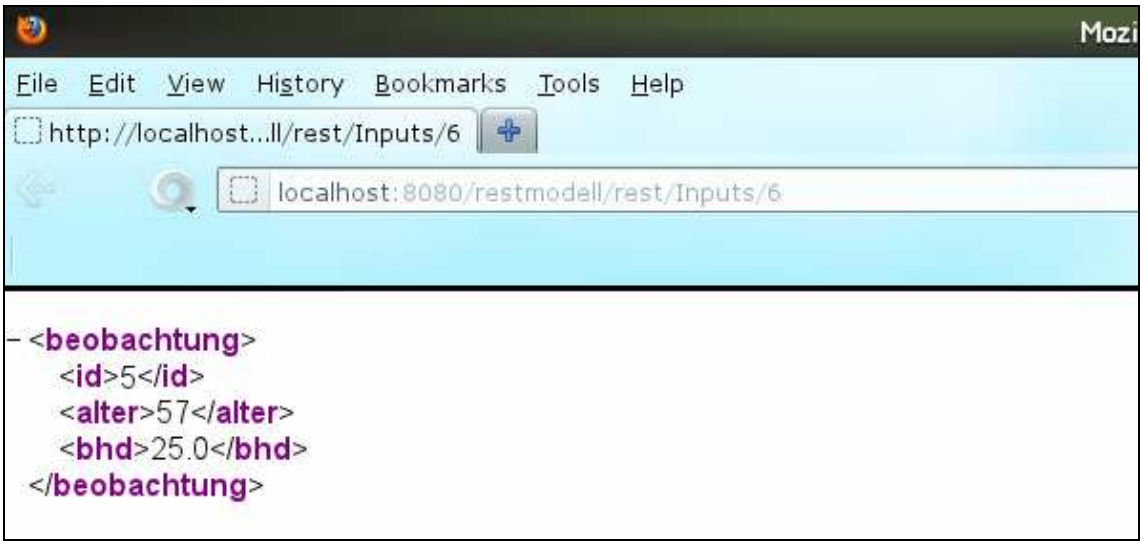

Abbildung 9: Einzelne Input-Ressource des REST-Webservices

Diese einzelnen Beobachtungs-Ressourcen können vom Nutzer des Webservices verändert werden. Dazu stehen die HTTP-Methoden DELETE, POST und PUT zur Verfügung. Über eine HTTP-DELETE-Anfrage an die URL einer Beobachtung erhält der Nutzer die Möglichkeit, eine Ressource zu löschen. Mit einer HTTP-PUT-Anfrage an die URL einer Bobachtung kann der Nutzer eine Beobachtung aktualisieren, also beispielsweise den BHD der Beobachtung ändern. Über HTTP-POST kann eine weitere Beobachtung, mit gewünschtem Alter und BHD, hinzugefügt werden. Auf diese Weise können die Eingangsdaten für die Anpassung des linearen Modelles verändert werden.

Um schließlich das lineare Modell an die Daten anzupassen, muss die Ressource für das lineare Regressionsmodell des Webservices manipuliert werden. Über den Einstiegspunkt des Webservices kann der Nutzer die Ausgabe-Ressource (Outputs) des Webservices erreichen (vgl. **Fehler! Verweisquelle konnte nicht gefunden werden.**), welche ebenfalls aus einer Liste besteht, die alle verfügbaren Modelle zusammenfasst und auf diese mit Links verweist. Der hier implementierte Webservice besitzt lediglich die Möglichkeit, eine lineare Regression an die Daten anzupassen, weshalb die Liste über die verfügbaren Modelle auf nur eine Ressource verweist. Dennoch wird hier die Übersicht über die verfügbaren Modellanpassungen als eigene Ressource bezeichnet, weil es denkbar wäre, den Webservice zu erweitern, damit auch andere Modelle als nur eine lineare Regression an die Daten angepasst werden können.

Um nun die lineare Regression auszuführen, hat der Nutzer die Möglichkeit, diese Ressource (vgl. Abbildung 10), das einfache lineare Modell, mit HTTP-PUT zu aktualisieren.

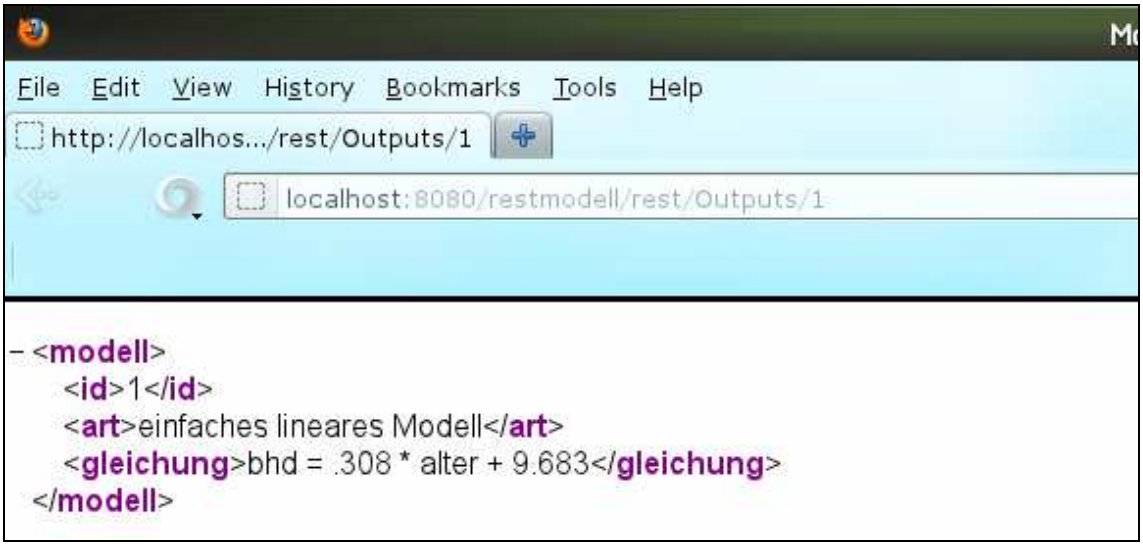

Abbildung 10: Ressource für das lineare Regressionsmodell des REST-Webservices

Wird eine HTTP-PUT-Anfrage an die URL des linearen Modells gesendet, führt der Webservice intern eine lineare Regression anhand der Beobachtungs-Ressourcen durch und ändert anschließend die Modellgleichung entsprechend ab. Um das Ergebnis zu erhalten, muss eine Repräsentation dieser Ressource mit einer HTTP-GET-Anfrage an der gleichen URL abgeholt werden.

3.3.2 Programmierung des REST-Webservices

Für die Umsetzung des gezeigten REST-Webservices wird die JAX-RS (Java API for RESTful Web Services) Referenzimplementierung Jersey in der Version 1.11 (Oracle 2012) mit Eclipse Indigo und Java Version 6 eingesetzt. JAX-RS ist ein Framework, um die Entwicklung von REST-konformen Webservices zu erleichtern.

Die Programmierung eines REST-Webservices mit Jersey ist aufwendiger und erfordert auch mehr Programmierkenntnisse als die vorher gezeigten Implementierungen des XML-RPC oder des SOAP-Webservices und kann deswegen hier nicht im Detail beschrieben werden. Alle für diesen REST-Webservice erstellten Quellcodes sind aber im Anhang (Kapitel D) enthalten. An dieser Stelle soll lediglich das grundlegende Konzept der Programmierung eines REST-Webservices mit Jersey erläutert werden. Dieses Konzept besteht aus einfachen Java-Objekten (POJOs) und Annotationen, um Metadaten in den Quelltext einzubinden. Für jede der mit dem Webservice angebotenen Ressourcen-Klassen wird eine Java-Klasse benötigt. Diese Klassen werden mit JAX-RS Annotationen versehen, um die Ressourcen und die Aktionen, die mit den Ressourcen ausgeführt werden können, zu identifizieren. Die Java-Klasse für die Eingabe-Ressourcen (Inputs) in Listing 12 besitzt verschiedene solcher Annotationen:

```
package com.sun.jersey.samples.helloworld.resources;

import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;

...

// The Java-class is hosted at path "/Inputs"
@Path("/Inputs")
public class InputResource {
  // Java-method processes HTTP GET requests
  @GET
  // Java-method produces content with MIME Media type
  // "application/xml"
  @Produces(MediaType.APPLICATION_XML)
  public File HTMLInputs() {
    ...
    return XMLFile;
  }
...
}
```

Listing 12: Programmcode-Auszug für die Ressourcen-Klasse Inputs des REST-Webservices

Mit der @PATH-Annotation wird der relative Pfad angegeben, an dem die Klasse zu erreichen ist. Diese Annotation ist wichtig, um die Ressource zu identifizieren und einen Zugriff auf die Klasse über die angegebene URL zu ermöglichen. Jede HTTP-Anfrage an diese URL wird an diese Klasse weitergeleitet.

Die @GET-Annotation bezieht sich auf die Methode *HTMLInputs*. Sie sorgt dafür, dass diese Methode bei einer GET-Anfrage ausgeführt wird. Es kann also eine HTTP-GET-Anfrage von dieser Ressourcen-Klasse beantwortet werden.

Eine GET-Anfrage enthält eine Angabe über den MIME-Typ (Multi-Purpose Internet Mail Extensions), also das Datenformat, das sie akzeptiert. Die @Produces-Annotation an derselben Methode gibt an, welcher MIME-Typ von der Methode produziert werden kann. Dies sorgt im hier gezeigten Fall dafür, dass die Ressourcen-Klasse eine HTTP-GET-Anfrage beantworten kann, welche den MIME-Typ application/xml, also eine XML-Datei akzeptiert.

Mittels der POJOs und der Annotationen ist es so möglich, für eine bestimmte HTTP-Anfrage an eine der URLs des Webservices eine bestimmte Aktion bzw. Methode auszuführen.

Um REST-Webservices zu nutzen, können beispielsweise der Web-Browser, Kommandozeilen-Clients oder in einer beliebigen Programmiersprache geschriebene Clients eingesetzt werden (Tilkov 2011). Um den hier vorgestellten Webservice zu nutzen, wurden HTTP-Anfragen aus Java heraus verwendet. Der für diesen Client verwendete Programmcode ist im Anhang (Kapitel D-12) enthalten.

3.3.3 Fazit zur Modellanbindung mit REST

Bei dem hier vorgestellten REST-Webservice handelt es sich um ein einfaches Beispiel, um die grundlegenden Prinzipien eines REST-Webservices zu verdeutlichen und die Eignung von REST für die Modellanbindung zu testen. Ungewohnt ist zunächst der Aufbau des Webservices aus Ressourcen, denn dieser Aufbau unterscheidet sich von den anderen hier betrachteten Ansätzen. Möchte man ein forstliches Modell mit einem REST-Webservice bereitstellen, muss es gelingen, die Modellfunktion mittels der HTTP-Methoden und der Ressourcen anzubieten. Außerdem ist es nötig, Repräsentationen für die Ressourcen bereitzustellen sowie Funktionen, um diese Ressourcen zu manipulieren. Bei der Erstellung von REST-Webservices müssen also mehrere Entscheidungen hinsichtlich des Aufbaus bzw. der Architektur des Webservices getroffen werden, bevor der Webservice erstellt werden kann. Daneben sorgt die Nutzung einer bestimmten Software, die für die Erstellung eines REST-Webservices gedacht ist, nicht automatisch dafür, einen Webservice zu erstellen, der auch wirklich REST-konform ist. Da kein Standard vorliegt, ist der Entwickler gefragt, die REST-Prinzipien anzuwenden und sie in dem zu erstellenden Webservice auch umzusetzen bzw. anzuwenden.

3.4 Modellanbindung mit WPS

Mit dem WPS wird ein Webservice implementiert, welcher eine Baumhöhenberechnung aus dem Baumalter durchführt. Der implementierte WPS liest die Koordinaten und das Alter einer beliebigen Anzahl Bäume ein und erstellt daraus eine GML-Datei mit den Koordinaten und den mit der Höhenfunktion aus dem Alter berechneten Höhen. GML ist eine Auszeichungssprache zum Austausch raumbezogener Daten auf der Basis von XML. Diese GML-Datei kann u.a. in einem Desktop-GIS, wie beispielsweise QGIS (Lang 2010), verwendet und grafisch dargestellt werden. Aus diesem Grund wird dieser WPS zusätzlich zu dem bereits gezeigten Oberhöhenmodell (enthalten im Anhang: Kapitel E) implementiert.

Die Baumhöhenberechnung für den WPS basiert auf der Trendfunktion nach Sloboda (1971), auf deren Grundlage Nothdurft (2007) ein hierarchisches Höhenwachstumsmodell erstellte. Die Wachstumsfunktion wurde von Schoneberg (2011) anhand von Daten aus ertragskundlichen Versuchsflächen der Nordwestdeutschen Forstlichen Versuchsanstalt für die Modellierung des Höhenwachstums von Fichten in Reinbeständen in Niedersachsen parametrisiert.

$$y(t) = \psi^{(1)} \left(\frac{\psi^{(4)}}{\psi^{(1)}} \right)^{exp\left[\frac{\psi^{(2)}}{\left(\psi^{(3)}-1\right)t^{\left(\psi^{(3)}-1\right)}} - \frac{\psi^{(2)}}{\left(\psi^{(3)}-1\right)t_0^{\left(\psi^{(3)}-1\right)}} \right]}$$

$\psi^{(1)} = 288.124077$ $\psi^{(3)} = 1.503162$ y = Baumhöhe
$\psi^{(2)} = 2.596616$ $\psi^{(4)} = 19.141894$ t = Baumalter
$t_0 = 50$

Gleichung 2: Modell zur Berechnung des Baum-Höhenwachstums von Fichten in Reinbeständen. Quelle: Schoneberg (2011)

3.4.1 Programmierung des WPS

Für die Programmierung des WPS wird Pywps in der Version 3.2.1 (Cepicky 2011) verwendet, ein Framework für die Implementierung von Web Processing Services des OGC in der Programmiersprache Python. Zur Nutzung von Pywps wird daneben Python 2.7 und der Apache HTTP Server Version 2.2 (Apache Software Foundation 2012b) eingesetzt.

Jeder WPS-Prozess, der mit PyWps erstellt wird, basiert auf einem eigenständigen Python-Script. Dieses Script enthält immer die Python-Klasse *Process*, welche eine abgeleitete Klasse ihrer Basisklasse *WPSProcess* darstellt, ihren Konstruktor *__init()__* und die Methode *execute*. Listing 13 zeigt diesen schematischen Aufbau eines PyWps-Prozesses:

```
from pywps.Process import WPSProcess

class Process(WPSProcess):

  def __init__(self):
    #init process
    WPSProcess.__init__(self,
        # Prozess's information like: identifier, title, status )

    #Inclusion of inputs and outputs to process class

  def execute(self):
    #Code
```

Listing 13: Gerüst eines PyWps-Prozesses Quelle: de Jesus et al. (2010)

Im Konstruktor der Klasse können bzw. müssen wichtige Prozessinformationen festgelegt werden, wie beispielsweise die Identifizierung, der Titel und die Version des WPS. Außerdem können hier Einstellungen vorgenommen werden, wie z.B., ob der Prozess asynchrone Aufrufe oder die Speicherung von Prozessergebnissen auf dem Server unterstützt. Auch die Ein- und Ausgaben des WPS mit ihren Bezeichnungen und den Datentypen müssen im Konstruktor festgelegt werden. Innerhalb der Methode *execute* steht schließlich der Python-Programmcode, welcher durchlaufen wird, wenn die Execute-Operation des WPS ausgeführt wird.

Bei dem in diesem Kapitel erstellten WPS besteht die Methode *execute* aus drei Abschnitten (vgl. Listing 14).

```
from readin import readin
from createXML import createXML
from calculateHeight import CalculateHeight

...

def execute(self):
  #1. Read input-data into arrays
  array_x, array_y, array_age= readin(self.dataIn.getValue())

  #2. Modell execution. Calculate height from age
  array_height= CalculateHeight(array_age)

  #3. Create GML-document
  self.dataOut.setValue(createXML(array_x, array_y, array_age,
      array_height, "GML-Document"))

  return
```

Listing 14: Programmcode der Methode „execute“ des implementierten WPS

Zuerst werden die Eingangsdaten verarbeitet, um sie in der Methode *execute* verwenden zu können. Diese Daten bestehen aus einer CSV-Datei, in der die Koordinaten und das Alter der Bäume angegeben sind, welche für die Weiterverarbeitung in Arrays zwischengespeichert werden. Danach werden die Baumhöhen aus den eingelesenen Werten für das Baumalter berechnet. Abschließend wird aus den Koordinaten der Bäume, dem Baumalter und den Baumhöhen eine GML-Datei erstellt. Diese Datei wird im gleichen Schritt als Ausgabedatei des WPS festgelegt und dann von PyWps in die Antwort auf die Execute-Operation aufgenommen. Die Programmcodes für die in diesen Schritten verwendeten Funktionen sowie der Programmcode für den erstellten WPS sind im Anhang (Kapitel F) enthalten.

Zusätzlich zu dem Programmcode des WPS wird noch eine Konfigurations-Datei benötigt, in der weitere Informationen für den WPS angegeben werden müssen. Diese Informationen erscheinen später in den Antworten auf die beiden WPS-Operationen GetCapabilities und DescribeProcess.

Für die Erstellung eines WPS mit PyWps sind außer dem WPS-Programmcode und der Konfigurationsdatei keine weiteren Programmierungen oder Einstellungen nötig. PyWps erstellt daraus alle Funktionalitäten und der WPS ist anschließend auf dem Server verfügbar.

3.4.2 WPS-Clients

Alle drei Operationen des WPS können ohne Einschränkungen über den Web-Browser erreicht werden. Dies ist möglich, weil PyWps alle drei Operationen des WPS über HTTP-GET implementiert. Für die Nutzung des WPS über den Web-Browser ist es lediglich notwendig,

die URL des WPS in die Adressleiste des Browsers einzugeben und alle für die jeweilige Operation nötigen Parameter an die URL anzuhängen, unter Verwendung von KVP.

Daneben ist es auch möglich, den WPS automatisch zu nutzen bzw. ein Programm für die Clienterstellung zu verwenden, weil durch die Operationen GetCapabilities und DescribeProcess eine Schnittstellenbeschreibung geliefert wird. Dazu steht beispielsweise in QGIS das Plugin WPS-Client von Düster (2012) zur Verfügung. Dieses Plugin ermöglicht es, einen WPS aus QGIS heraus zu benutzen und die Ergebnisse in QGIS anzuzeigen und weiterzuverwenden, wenn es sich um ein in QGIS darstellbares Format handelt, wie beispielsweise eine GML-Datei.

Nach der Installation und dem Aufruf des Plugins in QGIS kann über die URL eines WPS eine Verbindung zu dem WPS-Server aufgebaut werden. Anschließend steht eine Auswahl aller Prozesse zur Verfügung, die der WPS-Server anbietet. Diese Auswahl wird aus der GetCapabilities-Operation erzeugt, welche eine Übersicht über alle verfügbaren Prozesse bietet. Nach der Auswahl eines Prozesses öffnet sich ein Dialogfenster für den gewählten Prozess. Abbildung 11 zeigt dieses Dialogfenster für den in diesem Kapitel vorgestellten WPS.

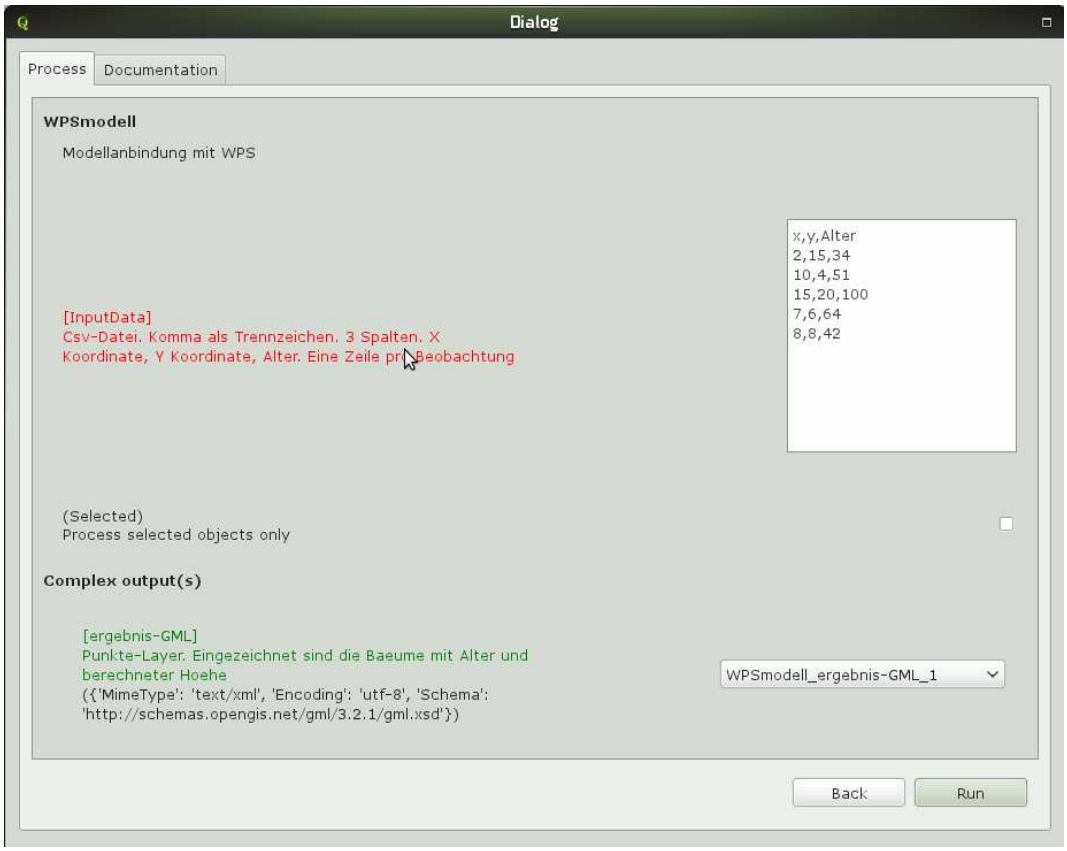

Abbildung 11: Nutzung des WPS aus QGIS mit dem WPS-Client Plugin

Im oberen Teil kann die CSV-Datei als Eingabedatensatz angegeben werden. Der untere Teil informiert über die zu erwartende Ausgabe, in diesem Fall ist dies eine GML-Datei. Durch

Betätigung des Run-Buttons wird die Execute-Operation ausgeführt und die GML-Datei wird anschließend in QGIS angezeigt.

3.4.3 Fazit zur Modellanbindung mit WPS

Die Erstellung des WPS ist durch den Einsatz des PyWps-Frameworks in den meisten Punkten ähnlich einfach und schnell zu realisieren wie die Implementierung des schon gezeigten SOAP-Webservices. Im Unterschied dazu müssen aber noch Angaben für die Schnittstellenbeschreibung in der Konfigurationsdatei des WPS gemacht werden, die später in den Antworten auf die beiden Operationen GetCapabilities und DescribeProcess erscheinen. Auch die Ein- und Ausgaben des WPS werden nicht automatisch festgelegt, sondern sie müssen vom Entwickler definiert werden. Dadurch wird zwar die Erstellung des WPS etwas aufwendiger, aber dies trägt auch zum besseren Verständnis des grundlegenden Konzeptes des WPS bei. Ein Vorteil des WPS ist, dass er vollständig über den Web-Browser genutzt werden kann. Dies erleichtert zum einen das Testen des Webservices, und verbessert zum anderen den Zugang zu dem Webservice durch andere Nutzer, weil nicht zwangsläufig ein Client für den WPS programmiert werden muss.
Ursprünglich ist der WPS für räumliche Prozesse gedacht, er kann aber auch in Bereichen ohne Bezug zu räumlichen Daten oder Operationen genutzt werden.

4. Bewertung der Webservice-Ansätze

In den beiden vorangegangenen Kapiteln, den Grundlagen und den Implementierungen, wird deutlich, dass teilweise große Unterschiede zwischen den verschiedenen Ansätzen existieren und sie jeweils bestimmte Vor- und Nachteile aufweisen. Um Vor- und Nachteile der vorgestellten Ansätze herausarbeiten und eine Empfehlung liefern zu können, sollen die Ansätze anhand von vier verschiedenen Kriterien untersucht und bewertet werden. Grundlage der Bewertung sollen neben den im Rahmen der Arbeit durchgeführten Implementierungen auch die Arbeiten anderer Autoren sein, die sich mit den verschiedenen Webservice-Ansätzen beschäftigt haben. Zunächst werden dafür in dem folgenden Kapitel die zugrundeliegenden Bewertungskriterien vorgestellt. Anschließend werden die einzelnen Webservices in Bezug auf diese Kriterien beurteilt. Die Beurteilung erfolgt auf folgender vierstufigen Skala:

- A: Webservice-Ansatz erfüllt das Bewertungskriterium vollständig
- B: Ansatz erfüllt das Kriterium mit einigen Einschränkungen
- C: Das Kriterium kann nur zum Teil erfüllt werden
- D: Webservice-Ansatz erfüllt das Kriterium nicht oder weist in Bezug auf das Kriterium gravierende Mängel auf

4.1 Die Bewertungskriterien

4.1.1 Standardisierung

Die Standardisierung ist für Webservices von entscheidender Bedeutung, weil durch die Bereitstellung und die Einhaltung von Standards Interoperabilität und Kompatibilität verbessert bzw. ermöglicht werden (Widjaja 2010). Erfüllt ein Webservice einen Standard, wird die Anbindung an einen Webservice stark vereinfacht, das Verbinden von verschiedenen Webservices wird erleichtert, und auch die Struktur und der Aufbau des Webservices kann besser nachvollzogen werden.

Bei der Überprüfung dieses Kriteriums stellt sich als erstes die Frage, ob ein Standard bzw. eine Spezifikation für den jeweiligen Webservice-Ansatz vorliegt. Daneben soll die Institution betrachtet werden, die den betreffenden Standard veröffentlicht hat. Es gilt zu überprüfen, ob es sich um ein anerkanntes Standardisierungsgremium handelt, welches auch zu Erweiterungen bzw. Verbesserungen des Standards beiträgt. Dies würde auf einen gut ausgearbeiteten Standard hindeuten, der von Experten auf dem Gebiet entwickelt worden ist und der eine gute Verbreitung und Akzeptanz besitzt.

4.1.2 Schnittstellenbeschreibung

Die Schnittstellenbeschreibung spielt eine wichtige Rolle bei der Erstellung von Clients für Webservices. Sie ermöglicht zum einen die automatische Client-Generierung und liefert zum anderen dem Nutzer nötige Informationen über den Webservice und dessen Nutzung. Außerdem hilft eine Schnittstellenbeschreibung dabei, Inkompatibilitäten auszugleichen, welche durch Änderung des Webservices in seinem Entwicklungsprozess zustande kommen können (Pautasso et al. 2008). Ohne eine standardisierte Schnittstellenbeschreibung müssen Clients für Webservices in der Regel von Hand erstellt werden. Dazu müssen u.a. die Methodennamen, die Eingabeparameter inklusive ihrer Datentypen und die Serveradresse bekannt sein. Wenn es für die Übermittlung dieser Informationen keinen standardisierten Weg gibt, kann es hier zu Abstimmungsschwierigkeiten kommen, und die Nutzung eines Webservices wird stark erschwert oder unmöglich.

In Kapitel 2 wurde deutlich, dass nicht jeder der Webservice-Ansätze eine Schnittstellenbeschreibung besitzt, mit deren Hilfe automatisch Clients für den Ansatz erstellt werden können. Zum anderen wird abgeprüft, ob die Schnittstellenbeschreibung auch dafür geeignet ist, dem menschlichen Nutzer die Nutzung des Webservices zu erleichtern, indem eine gute Beschreibung für den Webservice mit der Schnittstellenbeschreibung geliefert werden kann.

4.1.3 Umsetzbarkeit

Im Kapitel 3 (Implementierungen) wurde festgestellt, dass die jeweiligen Webservice-Ansätze einen unterschiedlich großen Aufwand erforderten, um mit ihnen Modelle anzubinden. Mit dem Kriterium der Umsetzbarkeit soll dieser Aufwand für die einzelnen Ansätze bewertet werden. Es soll auch bewertet werden, wie viele Programmierkenntnisse für die Umsetzung erforderlich sind. Dies ist zu großen Teilen von der eingesetzten Software abhängig, denn für die verschiedenen Ansätze gibt es unterschiedliche Bibliotheken, Entwicklungsumgebungen und Toolkits, die für die Programmierung eingesetzt werden können. Bei der Bewertung soll auch darauf geachtet werden, welche Software für den jeweiligen Webservice-Ansatz zur Verfügung steht und welcher Aufwand für die Erstellung der Webservices damit entsteht. Die Software unterliegt aber einem zeitlichen Wandel, weil sich die Verfügbarkeit von Softwareprodukten schnell ändern kann bzw. relativ schnell neue Software veröffentlicht werden kann. Aus diesem Grund ist auch die Bewertung der Umsetzbarkeit in gewissem Maße von der Zeit abhängig, in der sie durchgeführt wird.

4.1.4 Flexibilität

Mit diesem Kriterium soll die Fülle an Möglichkeiten abgeprüft werden, die der Webservice-Ansatz bietet. Dazu zählen beispielsweise, wie viele Datentypen angeboten werden, welche Transportprotokolle zur Nachrichtenübertragung eingesetzt werden können und welche Möglichkeiten zur Clienterstellung für den Webservice zur Verfügung stehen. Ein weiterer Aspekt ist, inwieweit die vorliegenden Modelle angepasst werden müssen, um sie als Webservices zur Verfügung stellen zu können. Je nachdem welcher Ansatz verwendet wird, ist es für den Programmierer nötig, den Programmcode des Modells mehr oder weniger stark zu modifizieren, damit er in einen Webservice integriert werden kann.

4.1.5 Einfachheit des Konzeptes

Mit diesem Bewertungskriterium soll abgeprüft werden, welche Vorkenntnisse für den jeweiligen Ansatz erforderlich sind bzw. wie komplex der Ansatz ist. Dieses Kriterium hängt damit auch von der Flexibilität des Ansatzes ab, denn bei einer geringen Flexibilität ist der Ansatz meist auch einfacher zu verstehen und es ist leichter, den gesamten Umfang des Ansatzes zu erfassen. Dennoch ist es wichtig, dieses Kriterium separat abzuprüfen. Durch einen leicht zu verstehenden Ansatz kann die Einarbeitungszeit in die jeweilige Technologie geringer sein und ein sichererer Umgang mit der jeweiligen Technologie kann besser gewährleistet werden. Außerdem kann ein zu komplexer Webservice dazu führen, dass er bei kleinen Änderungen

nicht mehr funktioniert und dann aufgrund der Komplexität schwierig zu testen (debuggen) ist.

Dieses Kriterium lässt sich von der Umsetzbarkeit abgrenzen, weil die Erstellung eines Webservices nicht unbedingt auch sein Verständnis voraussetzt, da dies u.U. weitgehend automatisch mit Hilfe von bestimmter Software geschehen kann. Auch wenn dieses Kriterium, stärker als die anderen, der Subjektivität unterliegt, soll es abgeprüft werden. Es geht hier aber nicht darum zu bewerten, wie gut der Ansatz dokumentiert ist, oder ob gute Quellen zum Verständnis vorliegen, sondern ob es sich um ein eher einfaches oder komplexes Konzept handelt.

4.2 Bewertung von XML-RPC

Standardisierung: Für XML-RPC liegt kein eigens definierter Standard vor, sondern eine Spezifikation von 1999, in welcher der Entwickler das XML-RPC Protokoll beschreibt, wie es in UserLand Frontier 5.1, einer Web-Entwicklungsumgebung, implementiert worden ist (Winer 1999). Somit existiert zwar eine Beschreibung von XML-RPC, an die sich bei der Implementierung eines XML-RPC gehalten werden kann, jedoch gibt es keinen allgemein anerkannten Standard, anhand dessen das Konzept von XML-RPC überarbeitet wird und mit dessen Hilfe Erweiterungen der Technologie eingearbeitet und zentral dokumentiert werden. XML-RPC ist also als endgültige Spezifikation anzusehen (Bayer 2002).

Die klaren Vorgaben in der zugrundeliegenden Spezifikation sind ein Vorteil hinsichtlich der Standardisierung bzw. der Spezifikation des XML-RPC. Beispielsweise sind die einzusetzenden Techniken, nämlich XML und HTTP, die verwendbaren Datentypen und der Aufbau der Nachrichten, die zwischen Client und Server ausgetauscht werden, genau vorgegeben. Damit liegt dem XML-RPC eine detaillierte und klare Spezifikation zugrunde, deren Nachteil es aber ist, dass sie nicht erweitert wird (Burghardt und Hagenhoff 2003) bzw. dass keine Verbesserungen des Ansatzes eingearbeitet werden. Dies liegt daran, dass die Spezifikation keinem Standardisierungsgremium unterliegt, wie beispielsweise dem W3C, das zur Aktualisierungen der Spezifikation beitragen könnte. Aus diesen Gründen kann XML-RPC hinsichtlich der Standardisierung als gut mit geringen Einschränkungen bewertet werden (Bewertung B).

Schnittstellenbeschreibung: Eine deutliche Schwäche von XML-RPC ist, dass keine Schnittstellenbeschreibung verfügbar ist (Bayer 2002). Zwar ist der XML-RPC durch die Spezifikation klar definiert und bietet dadurch nur relativ geringe Variationsmöglichkeiten bezüglich des Aufbaus eines Services, dennoch wäre es nötig, dass die Spezifikation eine Schnittstellenbeschreibung für den XML-RPC definiert. Zum einen, weil dem Anwender die genutzten Datentypen, die Namen der Operationen und die Adresse des Services bekannt gegeben werden

müssen. Zum anderen könnte der XML-RPC mit einer Schnittstellenbeschreibung automatisch nutzbar gemacht werden. Durch diesen Mangel kann es zu Informationslücken in Bezug auf die mit dem Webservice angebotenen Funktionen und zu Schwierigkeiten bei der Nutzung des XML-RPC kommen. Das Kriterium der Schnittstellenbeschreibung wird damit vom XML-RPC nicht erfüllt (Bewertung D).

Umsetzbarkeit: Die einfache Nutzung ist ein großer Vorteil von XML-RPC (Burbiel 2007). In dem Kapitel zur Implementierung eines Webservices mit XML-RPC konnte gezeigt werden, dass nur wenige Zeilen Programmcode nötig sind, um einen XML-RPC zu programmieren. Die dafür genutzten Bibliotheken übernahmen die XML-RPC-spezifischen Programmteile, wie das Erstellen der XML-Dokumente und das Versenden, wodurch kaum Programmierkenntnisse vorausgesetzt werden. Dies betrifft nicht nur die im Rahmen dieser Arbeit genutzten Programmiersprachen. Prinzipiell müssen bei der Nutzung einer verfügbaren XML-RPC Bibliothek für die Erstellung von Client und Server nicht alle Details des XML-RPCs verstanden werden (Laurent et al. 2001). In Bezug auf die Umsetzbarkeit zeigt XML-RPC also klare Vorteile gegenüber den anderen Webservice-Ansätzen und ist deswegen als sehr gut zu bewerten (Bewertung A).

Flexibilität: XML-RPC ist zwar einfach anzuwenden und schnell zu implementieren (Laurent et al. 2001), dies wird aber erst dadurch möglich, das XML-RPC nicht besonders flexibel eingesetzt werden kann. Diese geringe Flexibilität wird besonders daran deutlich, dass nur wenige Datentypen genutzt werden können und dass ein XML-RPC sich auf synchrone Aufrufe mit dem HTTP-Protokoll beschränkt. Durch diese geringe Flexibilität kann es nötig sein, dass ein Modell, welches angebunden werden soll, noch an den Webservice angepasst werden muss. Dies betrifft in erster Linie die Ein- oder Ausgaben des Modelles, da sie ggf. an das Typsystem des XML-RPC angepasst werden müssen, wenn die Datentypen des Modells nicht direkt mit den Datentypen des XML-RPC abbildbar sind. Beispielsweise können keine eigenen XML-Strukturen übertragen werden, was den XML-RPC sehr unflexibel macht (Burghardt und Hagenhoff 2003). Deswegen müssten Modelle, die XML-Dokumente als Eingangsdaten benötigen, vor der Nutzung mit XML-RPC so umgebaut werden, dass die in dem XML-Dokument enthaltenen Informationen auf die vom XML-RPC akzeptierten Datentypen abgebildet werden.

Es werden zwar die grundlegendsten Funktionalitäten vom XML-RPC angeboten, um einen Webservice zu erstellen, weil aber auf Flexibilität zugunsten der Einfachheit des Ansatzes verzichtet wird und weil die anderen Webservice-Ansätze deutlich mehr Möglichkeiten anbieten, muss die Flexibilität des XML-RPC als schlecht bewertet werden (Bewertung C).

Einfachheit des Konzeptes: Da dem Nutzer nur wenige Freiräume hinsichtlich der Gestaltung des Webservices gelassen werden und da der XML-RPC auf den weit verbreiteten und bekannten Standards HTTP und XML basiert, ist dieser Ansatz leicht zu verstehen und nachzuvollziehen. Auch die Spezifikation ist sehr übersichtlich und kurz gehalten, was zur Einfachheit des Konzeptes beiträgt. Die Verständlichkeit des Konzeptes des XML-RPC kann somit als sehr gut bewertet werden (Bewertung A).

Tabelle 1: Übersicht der Bewertung von XML-RPC

Kriterium	Bewertung
Standardisierung	B
Schnittstellenbeschreibung	D
Umsetzbarkeit	A
Flexibilität	C
Einfachheit des Konzeptes	A

4.3 Bewertung von SOAP-Webservices

Standardisierung: Die Bestandteile von SOAP-Webservices, d.h. SOAP und WSDL, sind vom W3C standardisiert. Das W3C ist ein Standardisierungsgremium, welches seit 1994 besteht und viele bekannte Standards definiert hat wie beispielsweise HTML oder XML (vgl. W3C 2004).

Der SOAP-Standard ist nicht nur für Webservices gedacht, sondern mit dem SOAP-Protokoll lassen sich alle Informationen verschicken, die sich als XML-Dokument darstellen lassen (Melzer 2008). Der Standard gibt lediglich verpflichtend vor, dass ein SOAP-Dokument ein Wurzelelement und ein Body-Element enthalten muss (vgl. Kapitel 2.3.1). Zusätzlich gibt es eine Fülle von optionalen Elementen (vgl. Gudgin et al. 2007), wodurch der Standard viele Möglichkeiten für Webservices bietet. Im Gegensatz zu der XML-RPC-Spezifikation macht der SOAP-Standard wesentlich weniger Vorgaben. Aus diesem Grund ist das WSDL-Dokument von entscheidender Bedeutung für den Webservice. Dadurch ist es möglich, die Flexibilität des Webservices, die durch den SOAP-Standard geboten wird, zu beschreiben und damit die Interoperabilität sicherzustellen. Wichtig für SOAP-Webservices ist also die Kombination der beiden Standards. Da es sich hier um detaillierte und etablierte Standards (Zeppenfeld und Finger 2009) eines großen und bekannten Gremiums handelt, kann das Kriterium der Standardisierung als sehr gut bewertet werden (Bewertung A).

Schnittstellenbeschreibung: Die Schnittstellenbeschreibung ist, wie bereits erwähnt, unabdingbar für die Nutzung von SOAP-Webservices. WSDL bietet den Vorteil der automatischen Client-Erstellung für den Webservice, wie bereits auch im Kapitel zur Implementation von SOAP-Webservices gezeigt werden konnte. Ein Nachteil von WSDL ist, dass es alleine nicht

ausreicht, um die Semantik einer Schnittstelle zu verstehen, weshalb die Schnittstellendokumentation in vielen Fällen noch durch Textdokumente begleitet wird (Tilkov 2011). Demnach kann WSDL in dem Fall, dass aufgrund der vielfältigen Möglichkeiten sehr komplexe Webservices aufgebaut werden, Schwächen dabei aufweisen, dem User die Schnittstelle bzw. den Webservice verständlich zu machen. Das Kriterium der Schnittstellenbeschreibung wird also von den SOAP-Webservices mit geringen Einschränkungen erfüllt (Bewertung B).

Umsetzbarkeit: Werden für die Erstellung von SOAP-Webservices Toolkits eingesetzt, können die Webservices mit wenig Aufwand erstellt werden und es reicht aus, die Grundzüge von SOAP-Webservices verstanden zu haben. Dies gilt besonders für die Programmiersprachen Java und C# (Richardson und Ruby 2007). Ohne derartige Toolkits ist die Erstellung von SOAP-Webservices schwierig, da nicht nur der Webservice mit seinen Funktionalitäten, sondern auch ein funktionierendes WSDL-Dokument erstellt werden muss, um den Webservice nutzbar zu machen. SOAP-Webservices ohne Toolkits zu erstellen ist somit aufwendiger als einen XML-RPC zu erstellen, gerade wenn die verwendete Programmiersprache keine Funktionalitäten für das Erzeugen des WSDL-Dokumentes bereitstellt. Daneben setzen SOAP-Webservices mehr Kenntnisse des Ansatzes voraus als ein XML-RPC, da das Konzept komplizierter zu verstehen ist (Burbiel 2007). Die Umsetzung wird als gut mit wenigen Einschränkungen bewertet, weil sie bei der Verfügbarkeit von Toolkits einfach zu realisieren ist, eine Umsetzung ohne solche Toolkits aber schwieriger wird und sie nur für die sehr weit verbreiteten Programmiersprachen verfügbar sind (Bewertung B).

Flexibilität: Die Flexibilität, welche durch den SOAP-Standard ermöglicht wird, ist der große Vorteil von SOAP-Webservices. Wie schon erwähnt, werden über 40 verschiedene Datentypen angeboten, und es können auch XML-Dokumente versendet werden. Daneben sind auch nicht nur Webservices, die auf dem Anfrage-/Antwort-Prinzip beruhen, möglich, sondern es können auch Zwischenstationen für die SOAP-Nachrichten angegeben werden (Melzer 2008). Damit wäre es auch möglich, durch einen einzelnen Aufruf mehrere Modelle gleichzeitig aufzurufen oder komplexere Modellverbindungen zu realisieren. Das ganze Maß an Flexibilität der SOAP-Webservices ist aber nicht unbedingt nötig für forstliche Modelle. SOAP-Webservices haben ihre Stärken eher in professionellen SOA-Szenarien (Service orientierte Architektur) (Pautasso et al. 2008). Für einfache Anwendungen kann die hohe Flexibilität von SOAP-Webservices auch hinderlich sein, da mit ihr eine große Komplexität einhergeht (Richardson und Ruby 2007). Hinsichtlich der Flexibilität werden die SOAP-Webservices insgesamt als sehr gut bewertet (Bewertung A).

Einfachheit des Konzeptes: Zwar bieten SOAP-Webservices eine hohe Flexibilität an, jedoch ist es auch möglich, auf diese Flexibilität zu verzichten und sich auf mehr oder weniger

grundlegende Funktionen zu beschränken. Damit wäre ein SOAP-Webservice nicht unbedingt sehr komplex oder schwierig zu verstehen. Betrachtet man jedoch das gesamte Konzept der SOAP-Webservices mit all seinen Möglichkeiten im Vergleich zu den anderen Webservice-Ansätzen, handelt es sich hier um einen eher schwierig zu verstehenden Ansatz. Daher wird die Verständlichkeit bzw. Einfachheit des Konzeptes als eher schlecht bewertet (Bewertung C).

Tabelle 2: Übersicht der Bewertung von SOAP-Webservices

Kriterium	Bewertung
Standardisierung	A
Schnittstellenbeschreibung	B
Umsetzbarkeit	B
Flexibilität	A
Einfachheit des Konzeptes	C

4.4 Bewertung von REST-konformen Webservices

Standardisierung: REST-konforme Webservices sind nicht standardisiert. Bei REST handelt es sich um generell gehaltene Richtlinien zur Gestaltung von Software-Architekturen (Richardson und Ruby 2007), die auch für Webservices angewendet werden können bzw. werden. Weil es keine Sammlung von festgelegten Richtlinien für REST-Architekturen gibt (Burbiel 2007), besteht eine gewisse Konfusion hinsichtlich der optimalen Vorgehensweise, REST-konforme Webservices zu erstellen (Pautasso et al. 2008). Aus diesen Gründen existieren viele Webservices, die als REST-konform deklariert werden, ohne dass sie bei strenger Betrachtung REST-konform sind (Tilkov 2011). Aufgrund des fehlenden Standards kann es also zu Inkompatibilitäten bzw. zu Problemen bei der Nutzung von REST-Webservices kommen. Daneben können der fehlende Standard und die nicht vorhandenen, allgemein akzeptierten Vorgaben zu Problemen bei der Erstellung und der Nutzung von REST-konformen Webservices führen, weil nicht explizit festgelegt ist, wie sie aufgebaut sein müssen. Daher ist das Kriterium Standardisierung von REST-Webservices als nicht erfüllt zu bewerten (Bewertung D).

Schnittstellenbeschreibung: Für REST-konforme Webservices gibt es die Schnittstellenbeschreibung WADL (Web Application Description Language), mit der, ähnlich wie mit WSDL für SOAP-Webservices, automatisch Clients für REST-konforme Webservices erzeugt werden können (Richardson und Ruby 2007). Im Unterschied zu SOAP-Webservices ist jedoch eine Schnittstellenbeschreibung bei REST-konformen Webservices nicht zwingend erforderlich, um den Webservice nutzen zu können. Dies liegt daran, dass eines der Grundprinzipien

von REST die uniforme Schnittstelle ist, und dass nicht für jeden Service eine eigene Schnittstelle mit ganz spezifischen Operationen und Datentypen entworfen wird (Tilkov 2011), wie dies bei den anderen, in der Arbeit betrachteten Ansätzen der Fall ist. Es werden lediglich die HTTP-Methoden angeboten, um mit dem Webservice zu interagieren, und diese werden bei jedem REST-konformen Webservice immer auf die gleiche Art eingesetzt. Ist dem Nutzer bekannt, wie REST-Webservices allgemein aufgebaut sind bzw. funktionieren, werden in der Regel kaum zusätzliche Informationen benötigt, um einen REST-konformen Webservice nutzen zu können. Dennoch kann die Schnittstellenbeschreibung WADL durchaus sinnvoll sein, weil damit nicht nur die Schnittstelle beschrieben wird, sondern der gesamte REST-Webservice besser und schneller verstanden werden kann. Die Bewertung für die Schnittstellenbeschreibung fällt somit sehr gut aus (Bewertung A).

Umsetzbarkeit: Eine Hürde bei der Umsetzung von REST-Webservices ist der fehlende Standard. Dadurch entstehen Schwierigkeiten für den Entwickler, denn es fehlen Vorgaben, welche Bedingungen bei dem Webservice eingehalten werden müssen bzw. wie der Webservice genau aufgebaut werden soll. Dadurch bleiben dem Entwickler zwar mehr Freiheiten, aber das Treffen von Entscheidungen hinsichtlich der Umsetzung eines Webservices ist ohne eine Orientierungshilfe, die ein Standard darstellen würde, schwieriger.

Für die programmiertechnische Umsetzung gibt es Toolkits, wie für SOAP-Webservices, welche die Entwicklung stark vereinfachen können und damit eine relativ einfache Umsetzung ermöglichen. Im Gegensatz zu SOAP-Webservices ist die Umsetzung aber wesentlich aufwendiger, weil für ein Modell nicht nur eine Schnittstelle erstellt werden muss, sondern für alle Ressourcen des Modells Funktionalitäten benötigt werden.

Die gute Umsetzbarkeit wird als nur teilweise erfüllt bewertet (Bewertung C), weil sie bei REST-Webservices aufwendiger ist als bei den anderen Ansätzen und weil der fehlende Standard die Erstellung erschwert.

Flexibilität: Für REST-Webservices gibt es keine Beschränkungen hinsichtlich der verwendbaren Datentypen und Dateiformate. Auch ist ein Vorteil, dass für REST-Webservices die Client-Erstellung sehr einfach ist, da die Interaktion über die HTTP-Methoden geschieht und diese für jeden REST-Webservice immer auf die gleiche Art eingesetzt werden. Die Restriktionen von REST betreffen eher den Aufbau des Webservices und den Umgang mit dem HTTP-Protokoll. Dies führt aber nicht zum Verlust von Funktionalität, REST-Webservices bieten prinzipiell alle nötigen Voraussetzungen, um forstliche Modelle verfügbar zu machen. Bei bestehenden Modellen ist aber meist ein großer Aufwand nötig, den Webservice und das Modell aufeinander abzustimmen. Es müssen beispielsweise sinnvolle Ressourcen identifiziert und dafür Repräsentationen bereitgestellt werden. Dies ist bei den anderen Webservice-

Ansätzen nicht nötig. Dort gilt es die Schnittstelle zu definieren. Diese Abstimmung von Modell und Webservice ist ein entscheidender Nachteil von REST in Bezug auf forstliche Modelle bzw. die Anbindung von bereits vorhandenen Anwendungen. Weil REST-Webservices durch den nicht vorhandenen Standard eine große Flexibilität ermöglichen, aber bestehende Modelle teilweise stark umgestaltet werden müssen, wird die Flexibilität als gut mit einigen Einschränkungen bewertet (Bewertung B).

Einfachheit des Konzeptes: Wie Tilkov (2009) bereits feststellte, ist der Entwurf der „Schnittstelle einer Anwendung zur Außenwelt nach REST-Prinzipien auf Basis von HTTP, URIs u.s.w. […] nicht einfach, auch wenn dies häufig behauptet wird. Die Herausforderung liegt vor allem im Umdenken – die Entwurfsansätze und -muster sind häufig neu und ungewohnt." Probleme bei dem Konzept sind also die Einarbeitung in die Thematik und das eher ungewohnte Vorgehen beim Konstruieren des Webservices. Ist diese Hürde der Einarbeitung aber überwunden, handelt es sich bei REST-konformen Webservices durchaus um ein verständliches Konzept, da zum einen die Nutzung eines REST-konformen Webservices einfach und zum anderen auch das grundlegende Konzept durchaus schlüssig und gut anwendbar ist. Hinsichtlich der Einfachheit des Konzeptes werden REST-konforme Webservices daher als gut mit geringen Einschränkungen bewertet (Bewertung B).

Tabelle 3: Übersicht der Bewertung von REST-konformen Webservices

Kriterium	Bewertung
Standardisierung	D
Schnittstellenbeschreibung	A
Umsetzbarkeit	C
Flexibilität	B
Einfachheit des Konzeptes	B

4.5 Bewertung von WPS

Standardisierung: Die Definition des Web Processing Service liegt in der Version 1.0.0 vor und ist vom OGC standardisiert. Das OGC setzt sich zusammen aus vielen Experten verschiedener Fachrichtungen, die OGC-Standards sind durchweg von hoher Qualität und verwendbar in einer Vielzahl von Anwendungsgebieten (Michaelis und Ames 2009).

Der WPS Standard gibt detailliert vor, mit welchen Technologien ein WPS implementiert werden muss, wie Ein- und Ausgaben gemacht werden müssen und wie die drei Operationen des WPS aufgebaut sein müssen. Daneben erhält der Standard auch Beispiele, um einen WPS auf Standardkonformität zu überprüfen.

Bei dem WPS-Standard handelt es sich somit um einen gut ausgearbeiteten Standard von einem anerkannten Gremium, der nicht nur zur Interoperabilität beiträgt, sondern auch dem Nutzer den Einstieg in WPS erleichtern kann und deswegen als sehr gut bewertet wird. (Bewertung A).

Schnittstellenbeschreibung: Die Schnittstellenbeschreibung eines WPS setzt sich zusammen aus der Antwort auf die GetCapabilities- und auf die DescribeProcess-Operation. Dabei gibt GetCapabilities alle verfügbaren Prozesse an und DescribeProcess enthält mehr oder weniger die gleichen Informationen wie ein WSDL-Dokument für einen speziellen Prozess (Stollberg und Zipf 2007). Durch die automatische Erzeugung der WPS-Prozessbeschreibungen kann ein Großteil manueller Arbeit eingespart und eine Anbindung von GIS-Funktionalitäten schnell ausgeführt werden (Brauner 2008). Wie erwähnt, ähnelt die Schnittstellenbeschreibung sehr stark dem WSDL-Dokument von SOAP-Webservices. Daher wird die Schnittstellenbeschreibung genauso bewertet wie bei den SOAP-Webservices (Bewertung B).

Umsetzbarkeit: Mit dem detaillierten und anerkannten Standard sowie WPS-Frameworks ist die Erstellung eines WPS gut zu realisieren und mit wenig Aufwand möglich. Frameworks sind für die gängigsten Programmiersprachen vorhanden und grundlegende Programmierkenntnisse sind ausreichend, um ein Modell mittels eines WPS verfügbar zu machen. Im Vergleich zu einem XML-RPC ist die Umsetzung etwas aufwendiger, weil die Schnittstellenbeschreibung für jeden WPS angepasst werden muss und weil der WPS eine größere Flexibilität bietet, wie z.B. eine weit größere Anzahl an Datentypen, welche ggf. für den WPS eingestellt werden müssen. Ein Vorteil hinsichtlich der Umsetzung ist, dass der Web-Browser für das Abrufen der erstellten Prozesse genutzt werden kann, wodurch eine Erstellung von Clients nicht unbedingt nötig wird und somit auch das Testen der Prozesse erleichtert werden kann.
Dadurch kann die Umsetzbarkeit insgesamt als gut mit wenigen Einschränkungen bewertet werden (Bewertung B), weil die Umsetzung im Vergleich zum XML-RPC etwas aufwendiger ist.

Flexibilität: Der WPS-Standard bietet trotz der vielen verpflichtenden Vorgaben eine gute Flexibilität. Es werden viele Datentypen unterstützt, besonders was räumliche Darstellungen angeht. Außerdem wird das Abspeichern von Ein- und Ausgaben mit der Angabe einer Referenz-URL ermöglicht, wodurch auch asynchrone Aufrufe möglich werden. Daneben unterstützt der WPS-Standard auch SOAP und es ist möglich, Web Processing Services zu erstellen, welche konform zu den REST-Prinzipien sind (vgl. Foerster et al. 2011). Diese Flexibilität bedingt dabei aber nicht unbedingt eine Zunahme der Komplexität des angebotenen Webservices. Das liegt daran, dass jeder WPS seine drei Operationen verpflichtend immer mit der gleichen Technik anbieten muss und lediglich zusätzlich dazu weitere Möglichkeiten geboten

werden. Auch die Client-Erstellung für den WPS trägt zur Flexibilität des Ansatzes bei, weil auch der Web-Browser als Client eingesetzt werden kann oder weil sie aus einem GIS heraus eingesetzt werden können, beispielsweise aus QGIS (vgl. Kapitel 3.4.2). Insgesamt soll die Flexibilität von WPS deswegen als sehr gut bewertet werden (Bewertung A).

Einfachheit des Konzeptes: Die Execute-Operation des WPS wird für die eigentliche Ausführung des WPS eingesetzt, die beiden anderen Operationen stellen die Schnittstellenbeschreibung dar. Die für die Execute-Operation eingesetzten Techniken entsprechen dem XML-RPC, nämlich HTTP-POST und XML. Damit ist die Ausführung des WPS vom Konzept her ähnlich einfach wie ein XML-RPC. Da aber der Standard des WPS wesentlich weiter geht als die XML-RPC-Spezifikation und auch mehr Möglichkeiten vom WPS angeboten werden, wird die Einfachheit des Konzeptes von WPS etwas schlechter bewertet als die des XML-RPC (Bewertung B).

Tabelle 4: Übersicht der Bewertung von WPS

Kriterium	Bewertung
Standardisierung	A
Schnittstellenbeschreibung	B
Umsetzbarkeit	B
Flexibilität	A
Einfachheit des Konzeptes	B

5. Diskussion

5.1 Vor- und Nachteile beim Einsatz von Webservices für forstliche Simulationsmodelle

Um forstliche Simulationsmodelle für den Nutzer verfügbar zu machen, gibt es unterschiedliche Wege. Sie können beispielsweise als ausführbare Programme zum Download angeboten werden oder der Quellcode der Modelle wird veröffentlicht. Dadurch erhält der Nutzer die Möglichkeit, die Modelle auf dem eigenen Rechner zu installieren und sie lokal auszuführen. Webanwendungen können ebenfalls für die Bereitstellung von Simulationsmodellen verwendet werden. In diesem Fall läuft der Modellcode dann entweder auf dem Rechner des Nutzers ab, wie z.B. bei einem Java-Applet, oder der Modellcode wird auf dem Server ausgeführt. Bei einer Webanwendung läuft die Kommunikation zwischen dem Nutzer und dem Modell über den Web-Browser ab. Über ihn werden die Eingangsdaten und Einstellungen für das Modell festgelegt und auch die Modellausführung gestartet.

Gegenüber diesen Arten der Modellbereitstellung bieten Webservices einige Vorteile, weshalb sie für den Einsatz für forstliche Simulationsmodelle interessant sind. Im Gegensatz zu

Modellen, die als Stand-Alone Programme verfügbar gemacht werden, bieten Webservices den Vorteil, dass sie, bei vorhandener Internetverbindung, von überall erreicht werden können und dass deren Nutzung nicht vom System abhängig ist, welches dem Nutzer zur Verfügung steht (Haubrock et al. 2009) bzw. auf dem das Modell installiert worden ist. Dies wird durch die offenen Standards (z.B. W3C und OGC) erreicht, die den Webservices zugrunde liegen und die deren Nutzung unabhängig von der zur Verfügung stehenden Hardware und Software möglich machen (Feng et al. 2011).

Bei Stand-Alone Programmen ist auf jedem System, auf dem das Simulationsmodell ausgeführt werden soll, eine Installation der Software erforderlich. Dieser Aufwand ist bei Simulationsmodellen, die über Webservices angeboten werden, nicht nötig, der Zugang zu den Simulationsmodellen ist über das Internet möglich, was die Verfügbarkeit von Webservices gegenüber Stand-Alone Programmen erleichtert. Um Webservices zu nutzen, muss jedoch ein Client programmiert oder zur Verfügung gestellt werden. Der Web-Browser kann nicht für alle Webservices als Client verwendet werden, wie dies z.B. bei dem im Kapitel 3 (Implementierungen) vorgestellten WPS möglich ist. Daher wird bei der Nutzung eines Webservices ggf. ein größeres Maß an Know-how auf Seiten des Anwenders vorausgesetzt als bei der Installation und der Anwendung eines Stand-Alone Programmes. Um dem Nutzer ohne das nötige Know-how dennoch den Zugang zu einem Webservice zu erleichtern, kann der Anbieter des Webservices die Nutzung beispielsweise über ein Webportal, über das auf die Webservices zugegriffen werden kann, anbieten. In einem solchen Fall müsste dann abgewogen werden, ob der Aufwand für die Bereitstellung eines Simulationsmodells mit einem Webservice sinnvoll ist.

Ein weiterer Vorteil von Webservices gegenüber Stand-Alone Programmen ist, dass mit Webservices auch Modelle und Algorithmen veröffentlicht werden können, die sich noch in der Entwicklungsphase befinden. Das ist deswegen praktikabel, weil im Fall, dass eine neue Version des Modells entwickelt wurde, nur ein Update auf dem Server nötig ist und damit jeder Nutzer automatisch Zugang zu der neusten Version erhält (Michaelis und Ames 2009). Webservices bieten also den Vorteil, dass Software-Updates an einer zentralen Stelle ausgeführt werden können und nicht einzeln von jedem Nutzer und für jedes System durchgeführt werden müssen.

Die eben genannten Vorteile von Webservices, die Unabhängigkeit vom System des Nutzers und die einfachen Aktualisierung des Simulationsmodells, können auch bei der Nutzung von Webanwendungen für Simulationsmodelle erfüllt werden. Webanwendungen unterscheiden sich insofern von Webservices, dass eine Webanwendung immer über den Webbrowser angeboten wird und die direkte Interaktion zwischen dem Benutzer und der Webanwendung bzw.

dem Simulationsmodell vorsieht. Die Modellanbindung über eine Webanwendung bindet den Nutzer an die durch den Modellierer entwickelte Schnittstelle, wohingegen ein Webservice ein wesentlich höheres Maß an Flexibilität bietet (Guha 2008). Ein Webservice muss nicht zwangsläufig aus einem Web-Browser heraus aufgerufen werden, sondern der Webservice kann auch von Computern bzw. anderer Software genutzt werden.

Dies bietet dem Nutzer mehr Möglichkeiten, das Simulationsmodell einzusetzen. Es kann leicht aus anderen Programmen heraus aufgerufen werden und dadurch einfach in andere, bestehende Prozesse eingebunden oder mit anderen Modellen verbunden werden. Dies ist bei der Bereitstellung des Simulationsmodells als Stand-Alone Programm oder mittels einer Webanwendung nicht ohne weiteres möglich, weil das Simulationsmodell in diesen Fällen nicht über eine standardisierte Schnittstelle verfügt und damit nicht unabhängig vom System und der Programmiersprache verwendet werden kann. Dieser Aspekt erleichtert auch die Wiederverwendung von Modellen, da sie aufgrund der standardisierten Schnittstelle relativ leicht in bestehende Anwendungen integriert werden können. Ein Vorteil von Webservices ist also ein höheres Maß an Flexibilität hinsichtlich der Nutzungsmöglichkeiten. Dies bringt aber auch, wie schon angesprochen, den Nachteil mit sich, dass bei Simulationsmodellen, die über Webservices angeboten werden, ein höheres Maß an Know-how beim Nutzer vorausgesetzt wird als bei Webanwendungen oder bei Stand-Alone Programmen.

Nach Theisselmann et al. (2009) bringt der Einsatz von Webservices für Simulationsmodelle bei den genannten Vorteilen folgende Nachteile mit sich: Bedarf nach Experten für den Entwicklungsprozess der Webservices, den Mangel an technischen Support speziell für Simulationsmodelle und die Notwendigkeit, bestehende Simulationsmodelle für den Einsatz mit Webservices umgestalten zu müssen.

Durch den Einsatz von Webservices steigt nicht nur das nötige Know-how auf Seiten des Nutzers, sondern gerade die Entwicklung von Webservices macht meist ein hohes Maß an Expertenwissen erforderlich. Dieser Nachteil wird durch den angesprochenen Mangel an Support speziell für Simulationsmodelle noch verstärkt. Im Kapitel 3 (Implementierungen) konnte gezeigt werden, dass Webservices für einfache Simulationsmodelle relativ leicht erstellt werden können, dennoch war auch hier ein gewisses Grundverständnis von Webservices nötig. Werden die anzubindenden Modelle komplexer und sollen zusätzlich noch weitere Möglichkeiten von Webservices genutzt werden, wie beispielsweise asynchrone Kommunikation oder komplexe Verbindungen von einzelnen Webservices, ist man schnell auf Experten aus dem Bereich Webservices angewiesen. Der Einsatz von Webservices bei komplexeren Aufgaben kann also leicht zu einem hohen Aufwand führen und ohne Expertenwissen nicht mehr zu bewältigen sein. Hier muss eine Abwägungsentscheidung getroffen werden, ob die

mit dem Webservice zu erreichenden Ziele bzw. die daraus entstehenden Vorteile den möglicherweise großen Aufwand rechtfertigen können oder ob eine andere, einfacher umzusetzende Form der Modell-Bereitstellung doch ausreichend ist.

Eine ähnliche Nutzen-Aufwand-Abwägung ist auch hinsichtlich des von Theisselmann et al. (2009) angesprochene Nachteils, dass bestehende bzw. ältere Simulationsmodelle umgestaltet werden müssen, nötig. Der Aufwand, ältere Simulationsmodelle als Webservices anzubieten, ist u.U. schwer abzuschätzen und kann eine grundlegende Änderung des Simulationsmodells nötig machen.

Neben dem eben behandelten Einsatz von Webservices für Simulationsmodelle, also dem Anbinden des gesamten Simulationsmodells mittels eines Webservices, stellen Chandrasekaran et al. (2002) noch zwei weitere Einsatzgebiete von Webservices für Simulationsmodelle heraus: Zum einen das Einbinden von bestimmte Modellkomponenten als Webservices. Dabei werden einzelne Bestandteile des Simulationsmodells über Webservices in das Modell integriert, wie z.B. eine Datenbank, bestimmte Kalkulationsfunktionen oder Werkzeuge zur Visualisierung der Modellergebnisse. Demnach können Webservices auch in Kombination mit Stand-Alone Programmen oder Webanwendungen verwendet werden, so dass die Funktionalität des Basisprogramms durch den Einsatz von Webservices erweitert wird. Eine Entscheidung für Webservices muss also nicht zwangsläufig bedeuten, dass ausschließlich auf Webservices gesetzt wird, sondern man kann sie auch zur Ergänzung bestehender Modelle einsetzen.

Zum anderen können Simulationsmodelle komplett aus unterschiedlichen Webservices zusammengestellt werden. Das Simulationsmodell baut sich in einem solchen Fall aus einzelnen Webservices auf, die miteinander vernetzt sind und jeweils bestimmte Aufgaben im Modellablauf übernehmen.

In beiden Fällen bieten Webservices für den Modellierer den Vorteil, auf bereits bestehende Funktionalitäten oder Submodelle zugreifen zu können und damit den Aufwand bei der Erstellung des eigenen Modells zu reduzieren. Neben geringeren Entwicklungskosten erlauben Webservices die einfachere und schnellere Kombination von heterogenen Anwendungen (Chang und Lee 2005). Werden, wie beim letztgenannten Einsatzgebiet von Webservices, Simulationsmodelle durch die Kombination von mit Webservices erstellten Submodellen gebildet, können einzelne Submodelle leicht ausgetauscht werden. Außerdem können die einzelnen Komponenten unabhängig voneinander entwickelt werden, wodurch Modelle einfacher aus Bestandteilen aufgebaut werden können, die aus unterschiedlichen Institutionen stammen (Theisselmann et al. 2009). Webanwendungen und Stand-Alone Programme bieten diese beiden Möglichkeiten nicht, da hier keine uniformen Schnittstellen für die einzelnen Submodelle

zur Verfügung stehen, und somit Erweiterungen und Veränderungen nicht unabhängig von der Programmiersprache und dem System implementiert werden können.

Der Einsatz von Webservices für Simulationsmodelle bietet also viele Vorteile. Besonders die Steigerung der Nutzungsmöglichkeiten und der Wiederverwendbarkeit der Simulationsmodelle sorgen dafür, dass die Simulationsmodelle durch Webservices einem größeren Nutzerkreis verfügbar gemacht werden können. Zu beachten ist jedoch, dass die Entwicklung von Webservices mit einem großen Aufwand verbunden sein kann und aus diesem Grund der Einsatz von Webservices für forstliche Simulationsmodelle nicht in jedem Fall sinnvoll ist. Weitere Gründe, die gegen den Einsatz von Webservices sprechen könnten, sind die Sicherheit und die Schnelligkeit von Webservices. Sie werden, separat zu den hier gennannten Vor- und Nachteilen, im nächsten Kapitel behandelt.

5.2 Sicherheit und Schnelligkeit von Webservices

„Neben dem Thema Sicherheit […] ist die Performance einer der großen Kritikpunkte im Zusammenhang mit Webservices“ (Melzer 2008). Snell et al. (2002) heben hinsichtlich der Sicherheit von Webservices besonders zwei Aspekte in den Vordergrund: Authentifizierung und Schutz der übermittelten Information. Bei einem sicheren Webservice vertraut der Sender einer Information darauf, dass der Empfänger der Information wirklich der ist, für den er sich ausgibt, und andersherum (Snell et al. 2002). Daneben muss gewährleistet werden, dass die Informationen nur für den jeweils adressierten Empfänger lesbar sind. Diese Aspekte können bei dem Einsatz von Webservices für forstliche Simulationsmodelle wichtig werden, wenn die Nutzung des Modells nicht frei zugänglich sein soll und/oder wenn mit dem Modell Daten prozessiert werden, die für Dritte nicht einsehbar sein dürfen.

Die Spezifikationen bzw. Standards (SOAP, WSDL, XML-RPC-Spezifikation, WPS-Standard) der in dieser Arbeit behandelten Webservice-Ansätze bieten selber keine Mechanismen für die Sicherheit an. Für SOAP-Webservices gibt es zusätzliche Standards, wie z.B. WS-Security und WS-Trust, die den SOAP-Standard erweitern und es ermöglichen, Sicherheitsaspekte zu berücksichtigen. Diese Möglichkeit steht jedoch nur speziell für SOAP-Webservices und nicht für die anderen in dieser Arbeit behandelten Webservice-Ansätze zur Verfügung und erfordert eine tiefere Einarbeitung. Aus diesen beiden Gründen soll an dieser Stelle nicht weiter darauf eingegangen werden.

Mehrere Autoren (Bielski et al. 2011, Haubrock et al. 2009, Melzer 2008, Tilkov 2011) verweisen in Bezug auf die Berücksichtigung von Sicherheitsaspekten bei Webservices auf den Einsatz von HTTPS. Bei HTTPS handelt es sich um eine Kombination von HTTP und SSL (Secure Sockets Layer) bzw. TLS (Transport Layer Security). Mit dieser Technik können Da-

ten verschlüsselt, signiert oder authentisiert ausgetauscht werden (Schwenk 2005). Detaillierte Beschreibungen zu HTTPS und SSL finden sich beispielsweise bei Schwenk (2005). Möglich ist die Absicherung der Webservices über HTTPS bei allen in dieser Arbeit behandelten Webservice-Ansätzen, weil alle HTTP als Transportprotokoll verwenden, bzw. verwenden können. Auch für SOAP-Webservices ist SSL bzw. HTTPS ein bewährter und effektiver Weg, wenn es zwischen den Kommunikationspartnern keine Zwischenstationen für die SOAP-Nachrichten gibt (Melzer 2008), d.h. wenn nur synchrone Kommunikation eingesetzt wird.

Für die Umsetzung auf Clientseite besteht die Möglichkeit, HTTPS-Anfragen über eine HTTP-Bibliothek abzusetzen, die den Zugriff über SSL sowie das Setzen von Benutzername und Passwort unterstützt, was praktisch mit jeder HTTP-Bibliothek möglich ist (Tilkov 2011). Auf der Serverseite kann das SSL-Handling und die Prüfung von Benutzername und Passwort i.d.R. vom eingesetzten Webserver übernommen werden, der entsprechend konfiguriert werden kann (Tilkov 2011). Die im Kapitel 3 (Implementierungen) eingesetzten Webserver Apache HTTP-Server und Apache Tomcat bieten beide entsprechende Funktionalitäten an (vgl. Chopra et al. 2011, Ristic 2005).

Zwar gibt es erprobte und effektive Möglichkeiten, um sichere Webservices zu erstellen, jedoch sollte immer beachtet werden, dass damit ein erheblicher Aufwand bezüglich der Entwicklung und der Administration einhergeht und daher immer eine kritische Betrachtung der tatsächlichen Sicherheitsrisiken erfolgen muss (Tilkov 2011). Aus diesen Gründen kann es u.U. auch sinnvoll sein, auf den Einsatz von Webservices bei forstlichen Simulationsmodellen zu verzichten und andere Methoden für die Bereitstellung zu nutzen, wenn die Sicherheit einen hohen Stellenwert hat.

Nachteilig bei der Erfüllung von Sicherheitsaspekten ist neben dem erhöhtem Aufwand auch die Performance, weil diese durch die Anwendung von SSL verschlechtert wird (vgl. Melzer 2008, Tilkov 2011). Die Performance bei dem Einsatz von Webservices für Simulationsmodelle muss aber auch aus anderen Gründen kritisch betrachtet werden. Sie steht stark im Zusammenhang mit dem XML-Format, welches für den Datenaustausch bei Webservices verwendet wird. XML besitzt neben den Vorteilen, dass es einfach geparst und verstanden werden kann (Feng et al. 2011) und damit die Interoperabilität von Webservices möglich macht, den Nachteil, dass es einen bedeutenden Overhead besitzt (Lawrence 2004). Durch diesen Overhead wird der Datenaustausch über das Internet erhöht, was die Performance des Webservices verringert, besonders wenn große Datenmengen ausgetauscht werden (Feng et al. 2011). Neben dem Verschicken der Daten müssen die Modelleingaben und -ausgaben von dem Webservice in XML-Dokumente kodiert bzw. aus den Dokumenten dekodiert werden. Dieser Aufwand steigt ebenfalls mit höherem Datenaufkommen.

Die Schnelligkeit von Webservices wird also maßgeblich davon beeinflusst, wie groß die Datenmengen sind, die zwischen dem Client und dem Webservice ausgetauscht werden. Deswegen muss bei der Entscheidung, ob Webservices für die Anbindung von forstlichen Simulationsmodellen eingesetzt werden sollen, das Datenaufkommen kritisch betrachtet werden. Ein ausreichender Vergleich zwischen allen vier behandelten Webservice-Ansätzen bezüglich der Schnelligkeit liegt zwar nicht vor, jedoch ist festzustellen, dass alle Ansätze vorzugsweise XML und HTTP für den Datenaustausch verwenden. Deswegen sollten die Ansätze tendenziell eine ähnliche Performance aufweisen. Um hier allerdings klarere Aussagen hinsichtlich der Schnelligkeitsunterschiede zwischen den Webservice-Ansätzen treffen zu können, müssten weiterführende Untersuchungen durchgeführt werden.

Hinsichtlich der Sicherheit und der Schnelligkeit bleibt festzuhalten, dass dies kritische Aspekte bezüglich Webservices sind. Ggf. benötigte Sicherheit kann durch den Einsatz von SSL hergestellt werden, dies ist jedoch mit einem Mehraufwand verbunden und geht zulasten der Performance von Webservices. Diese wird daneben maßgeblich durch den für das Simulationsmodell nötigen Datentransfer bestimmt, der aus diesem Grund mitentscheidend dafür ist, ob der Einsatz von Webservices für ein bestimmtes Simulationsmodell zweckmäßig ist. Bei der Performance von Webservices spielen daneben auch die Art der Internetverbindung sowie die Leistungsfähigkeit des Servers eine Rolle. Es sollte also auch beachtet werden, welche Bandbreite die Internetverbindung der Zielgruppe des Webservices besitzt und welche Menge an Anfragen an den Server von dieser Zielgruppe zu erwarten ist.

5.3 Arten von Modellen, die für den Einsatz von Webservices geeignet sind

Die Kapitel zu den Vor- und Nachteilen von Webservices und der Sicherheit und der Schnelligkeit geben erste Hinweise darauf, welche Arten von Simulationsmodellen für den Einsatz mit Webservices geeignet sind. Der erste Aspekt, der beachtet werden sollte, ist, dass die Modelle aus einer Programmiersprache heraus aufgerufen werden können, für die Software erhältlich ist, die den Entwickler bei der Erstellung von Webservices unterstützt. Dies ist auch abhängig vom gewählten Webservice-Ansatz, da hier Unterschiede in der Verfügbarkeit in unterschiedlichen Programmiersprachen vorliegen können.

Sinnvoll ist der Einsatz von Webservices besonders bei Modellen, die oft benötigt werden bzw. in vielen Bereichen eingesetzt werden und die auch mit anderen Programmen bzw. Modellen zusammen angewendet werden. In solchen Fällen kann sich der zusätzliche Aufwand, der sich durch die Erstellung der Webservices für das Modell ergibt, lohnen. Nach der Anbindung mit einem Webservice kann das Modell anschließend einfach wiederverwendet werden und lässt sich leicht in andere Programme integrieren. Dadurch steigen Nutzungsmöglichkei-

ten und Wiederverwendbarkeit und das Modell muss nicht für andere Projekte neu erstellt oder aufwendig eingebunden werden. Modelle mit einen sehr kleinen bzw. engen Anwendungsgebiet, die möglicherweise nur einem kleinen Nutzerkreis zur Verfügung gestellt werden sollen oder nur für sehr spezifische Aufgaben eingesetzt werden, sind für den Einsatz mit Webservices eher weniger geeignet. In solchen Fällen kann nur geringfügig von der Möglichkeit der einfachen Integration in andere Programme und der Wiederverwendbarkeit profitiert werden. Der erhöhte Aufwand würde sich möglicherweise nicht auszahlen.

Modelle, die sich aus mehreren Submodellen zusammensetzen, eignen sich ebenfalls gut für die Anbindung mit Webservices. Diese Submodelle könnten einzeln mit Webservices verfügbar gemacht und anschließend miteinander verbunden werden. Dadurch können die Submodelle unabhängig voneinander weiterentwickelt, modifiziert oder durch andere Submodelle ausgetauscht werden. Auch wird die Zusammenarbeit unterschiedlicher Institutionen oder Fachrichtungen an einem gemeinsamen Modell durch den Einsatz von Webservices erleichtert. Nach der Anbindung mit Webservices sind nur wenige Abstimmungen zwischen den Submodellen nötig und sie können auf unterschiedlichen Servern verteilt sein, da sie über das Internet aufgerufen werden können. Durch die standardisierte Schnittstelle ist es dann auch möglich, die verwendeten Submodelle in andere Anwendungen zu integrieren, ohne die Schnittstellen erneut anpassen zu müssen.

Problematisch sind Webservices, wie schon im Kapitel 5.2 herausgestellt, bei großen Datenmengen, die zwischen ihnen und dem Client ausgetauscht werden. Außerdem werden bei Webservices die Ausgangs- bzw. Eingangsdaten in XML-Dokumente kodiert bzw. dekodiert. Dieser Vorgang wird bei anderen Möglichkeiten, Simulationsmodelle bereitzustellen, wie z.B. Stand-Alone Programme oder Webanwendungen, nicht durchgeführt, wodurch Webservices langsamer sein können als andere Methoden, um Modelle bereitzustellen. Daraus ergibt sich, dass sich Simulationsmodelle, die große Mengen an Eingangsdaten benötigen und/oder große Mengen an Ausgangsdaten produzieren, weniger für den Einsatz mit Webservices eignen. Auch sind Simulationsmodelle, die sich aus Submodellen zusammensetzen, zwischen denen große Datenflüsse zustande kommen, für den Einsatz mit Webservices eher ungeeignet. Die Quantifizierung, ab wann die Datenmengen zu groß sind für den Einsatz von Webservices, ist dabei allerdings schwierig zu treffen. Sie hängt auch davon ab, welche Toleranz die Nutzer hinsichtlich der Bearbeitungszeit der Modelle haben.

5.4 Vergleich der vorgestellten Webservice-Ansätze für den Einsatz mit forstlichen Modellen

Alle vier Ansätze für Webservices eigenen sich prinzipiell für das Anbinden von forstlichen Simulationsmodellen. Der XML-RPC bietet den Vorteil, dass mit ihm relativ schnell und einfach Webservices für Simulationsmodelle erstellen werden können, weil ihm ein einfaches Konzept zugrunde liegt, das eine einfache Umsetzbarkeit ermöglicht. Problematisch bei dem XML-RPC ist aber die fehlende Schnittstellenbeschreibung, wodurch er nicht automatisch angebunden werden kann und keine Dokumentation für die Schnittstelle zur Verfügung stellt. Dies kann die Nutzung des XML-RPC stark erschweren, denn der Nutzer ist auf Informationen über die Schnittstelle angewiesen, um den Webservice aufrufen zu können. In diesem Punkt sind die anderen drei Webservice-Ansätze besser für den Einsatz mit forstlichen Simulationsmodellen geeignet, denn sie bieten alle eine Schnittstellenbeschreibung an, mit der auch automatisch Clients generiert werden können. Die anderen drei Webservice-Ansätze besitzen außerdem eine größere Flexibilität als der XML-RPC, besonders was die verfügbaren Datentypen betrifft. Dazu gehört beispielsweise auch die Möglichkeit, XML- und Textdokumente als Ein- und Ausgaben zu nutzen. Simulationsmodelle, die größere Datenmengen einlesen müssen oder mehrere Ausgabewerte mit unterschiedlichen Datentypen besitzen, können deswegen mit einem XML-RPC nicht, oder nur mit sehr hohem Aufwand angebunden werden. Der XML-RPC ist also, aufgrund der fehlenden Schnittstellenbeschreibung und der geringen Flexibilität, schlechter für Simulationsmodelle geeignet als die anderen drei Webservice-Ansätze.

REST-Webservices besitzen im Vergleich zu den anderen Webservice-Ansätzen, die in dieser Arbeit behandelt werden, den Nachteil, dass ihnen kein Standard bzw. keine Spezifikation zugrunde liegt. Weil also keine allgemein anerkannten Richtlinien für REST-konforme Webservices verfügbar sind, kann die Erstellung schwieriger sein. Eine Auswirkung des fehlenden Standards ist beispielsweise, dass viele Webservices als REST-konform deklariert werden, ohne dass sie bei genauerer Betrachtung REST-konform sind (Tilkov 2011). Dies kann zu Problemen oder Inkompatibilitäten bei der Nutzung von REST-Webservices bzw. als REST-konform deklarierten Webservices führen. Die anderen zwei Ansätze (SOAP-Webservices und WPS) sind hinsichtlich der Standardisierung also besser für den Einsatz von Simulationsmodellen geeignet. Dadurch sind sie zum einen besser zu nutzen, weil sie sich an einem Standard bzw. einer Spezifikation orientieren. Zum anderen wird die Erstellung in dem Punkt erleichtert, dass man sich an fest vorgegebenen Richtlinien orientieren kann. Die Erstellung bzw. Umsetzung wurde, neben der Standardisierung, bei REST-Webservices schlechter bewertet als bei den anderen Ansätzen. Bei REST-Webservices ist der Aufwand der Erstellung

eines Webservices für ein bestimmtes Modell größer, weil nicht nur eine Schnittstelle für das Modell konstruiert werden muss, wie dies bei den anderen drei Ansätzen der Fall ist. REST-Webservices werden genutzt, indem der Anwender mit den einzelnen Ressourcen des Webservices interagiert. Zu diesem Zweck müssen für alle diese Ressourcen entsprechende Funktionalitäten bzw. Schnittstellen erstellt werden, um die einzelnen Ressourcen zu manipulieren. Bei den anderen Ansätzen ist die Umsetzung somit weniger aufwendig, die Anbindung eines Modells kann bei ihnen deswegen meist schneller erfolgen. Aufgrund des fehlenden Standards und der aufwendigeren Umsetzung eignen sich REST-Webservices also weniger für die Anbindung von Modellen als SOAP-Webservices oder WPS.

Im Kapitel 4 (Bewertung der Webservice-Ansätze) wurden diese beiden Ansätze in vier der fünf Bewertungskriterien gleich bewertet. Lediglich die Einfachheit des grundlegenden Konzeptes wurde bei dem WPS besser bewertet als bei SOAP-Webservices. Diese beiden Ansätze eigenen sich am besten für die Bereitstellung von forstlichen Simulationsmodellen. Beiden liegt ein Standard von anerkannten Standardisierungsgremien zugrunde. Vor allem dadurch sind sie besser geeignet als REST-Webservices, da sich Inkompatibilitäten besser vermeiden lassen und bei beiden Ansätzen klare Richtlinien für die Erstellung von Webservices vorliegen. Daneben besitzen beide Ansätze eine Schnittstellenbeschreibung, wodurch die Nutzung der Webservices einfacher ist als bei einem XML-RPC. Außerdem heben sie sich von diesem durch die höhere Flexibilität ab. Der Aspekt, dass bei beiden Ansätzen Eingangs- und Ausgabedaten, die als XML-Dokumenten vorliegen, verarbeitet werden können, ist einer der Vorteile gegenüber dem XML-RPC, der aus der höheren Flexibilität resultiert.

Der WPS-Standard schränkt den Entwickler stärker ein als der SOAP-Standard, welcher weniger Vorgaben enthält und mehr Möglichkeiten bietet. Dadurch verschafft ein SOAP-Webservice dem Entwickler mehr Freiheiten, jedoch hat der stärker einschränkende Standard des WPS zur Folge, dass die Interoperabilität verbessert wird und dass der Webservice weniger komplex werden kann. Zu den zusätzlichen Möglichkeiten, die ein SOAP-Webservice bietet, gehören beispielsweise die Unabhängigkeit vom Transportprotokoll und die Möglichkeit, eine Route mit mehreren Zwischenstationen für eine SOAP-Nachricht festzulegen und damit komplexe Verbindungen zwischen einzelnen Webservices zu realisieren. Dabei ist dann aber auch mehr Know-how für die Erstellung eines SOAP-Webservices nötig als bei einem WPS. Ebenso erfordert die Clienterstellung für SOAP-Webservices mehr Know-how, denn beim WPS kann auch der Web-Browser als Client eingesetzt werden, wodurch auf eine Client-Programmierung verzichtet werden kann.

Insgesamt sind also SOAP-Webservice und WPS am besten für den Einsatz mit forstlichen Simulationsmodellen geeignet, wobei der WPS meist einfacher zu erstellen und zu nutzen ist

und ein SOAP-Webservice mehr Möglichkeiten anbietet als ein WPS, dann aber auch ein erhebliches Know-how voraussetzt. Somit bestimmen auch die bzw. das anzubindende Simulationsmodell darüber, welcher Webservice-Ansatz eingesetzt werden sollte. Für Modelle, bei denen sehr komplexe Verbindungen der Submodelle realisiert werden sollen und ggf. auch asynchrone Aufrufe nötig werden, also für Modelle mit einer komplexen serviceorientierten Architektur, bieten sich SOAP-Webservices eher an. Die Anbindung von Simulationsmodellen als Ganzes oder von Modellen, bei denen die Submodelle nacheinander ablaufen bzw. zwischen denen die Interaktionen weniger komplex sind, kann dagegen meist einfacher und mit weniger Vorwissen mit einem WPS realisiert werden.

6. Schlussfolgerungen

In dieser Arbeit wurden, ausgehend von einem allgemeinen Überblick über Webservices (Kapitel 2), die grundlegenden Funktionsweisen von vier Ansätzen für Webservices herausgearbeitet und vorgestellt. Bei diesen vier Ansätzen handelt es sich um XML-RPC, SOAP-Webservices, REST-Webservices und WPS. Mit diesen Webservice-Ansätzen wurde jeweils ein einfaches Simulationsmodell angebunden (Kapitel 3), womit die grundsätzliche Eignung dieser Webservice-Ansätze für den Einsatz für forstliche Simulationsmodelle festgestellt werden konnte. Im weiteren Verlauf der Arbeit wurden die grundlegenden Unterschiede zwischen diesen Ansätzen aufgezeigt, um die verschiedenen Ansätze zu bewerten (Kapitel 4) und anschließend eine Empfehlung abgeben zu können, welche der vier Ansätze sich am besten für den Einsatz mit forstlichen Simulationsmodellen eignen. Dabei zeigte sich, dass der WPS und Webservices, die auf SOAP und WSDL basieren, am besten für die Bereitstellung von forstlichen Simulationsmodellen eingesetzt werden können. Abhängig ist die Eignung eines Ansatzes dabei auch von dem anzubindenden Simulationsmodell und den Kenntnissen des Entwicklers über den genutzten Webservice-Ansatz. Außerdem konnten mehrere Vorteile von Webservices gegenüber anderen Methoden zur Bereitstellung von Simulationsmodellen, wie beispielsweise Stand-Alone Programme oder Webanwendung aufgezeigt werden. Dazu gehören vor allem die erhöhte Wiederverwendbarkeit von Modellen und die Integration von Modellen in andere Programme, unabhängig von der Programmiersprache und dem Betriebssystem. Außerdem zeigte sich, dass mit Webservices auch Sicherheitsaspekte bezüglich der Informationssicherheit und der Authentifizierung berücksichtigt werden können. Herausgestellt hat sich auch, dass bei Webservices vor allem die Performance problematisch sein kann, die bei dem Einsatz für komplexe Simulationsmodelle unbedingt berücksichtigt werden muss. Aufgrund der Performance von Webservices lassen sich auch nicht alle Arten von Simulationsmodellen

problemlos mit Webservices bereitstellen. Besonders Modelle, die mit großen Datenmengen operieren, sind eher schlecht für den Einsatz mit Webservices geeignet.

In dieser Arbeit konnte festgestellt werden, dass Webservices grundsätzlich zur Anbindung von forstlichen Simulationsmodellen eingesetzt werden können, die im Zuge des BEST-Projekts bereitgestellt werden. Als nächster Schritt gilt es, die hier herausgearbeiteten Erfahrungen in die Praxis zu übertragen und Simulationsmodelle aus dem BEST-Projekt mit Webservices für den Nutzer verfügbar zu machen. Aus dem Praxiseinsatz von Webservices können wichtige Erfahrungen gewonnen werden, beispielsweise wie verschiedene Nutzergruppen Webservices annehmen, welche Ansprüche diese an die Webservices stellen oder wie sich das Verhalten von Webservices bei erhöhten Nutzer-Anfragen verändert. Daneben sind Erfahrungen bezüglich der Performance von Webservices nötig, und die Verarbeitung größerer Datenmengen sollte in der Praxis getestet werden. Neben dem Anbinden von einzelnen Modellen als Ganzes ist auch die Erstellung von Modellen interessant, die sich aus mehreren Submodellen zusammensetzen, welche mit Webservices erstellt wurden.

Die Arbeit kann einen ersten Überblick über Webservices liefern, nun gilt es, Webservices in der Praxis anzuwenden, um weitere Möglichkeiten und Grenzen von Webservices für forstliche Simulationsmodelle aufzeigen zu können.

Literaturverzeichnis

Abts, D. 2010. Masterkurs Client/Server-Programmierung mit Java: Anwendungen entwickeln mit Standard-Technologien: JDBC, UDP, TCP, HTTP, XML-RPC, RMI, JMS und JAX-WS. Wiesbaden: Vieweg+Teubner Verlag.

Apache Software Foundation. 2010. About Apache XML-RPC. http://ws.apache.org/xmlrpc/ (Zugegriffen März 6, 2012).

Apache Software Foundation. 2011. Apache Axis2/Java. http://axis.apache.org/axis2/java/core/ (Zugegriffen März 6, 2012).

Apache Software Foundation. 2012a. Apache Tomcat. http://tomcat.apache.org/index.html (Zugegriffen März 6, 2012).

Apache Software Foundation. 2012b. Dokumentation zum Apache HTTP Server Version 2.2. http://httpd.apache.org/docs/2.2/de/ (Zugegriffen März 9, 2012).

Bayer, T. 2002. Einführung in Webservices mit XML-RPC. http://www.oio.de/public/xml/xml-rpc.htm (Zugegriffen März 12, 2012).

Bayer, T., und D. M. Sohn. 2007. REST Web Services. http://www.thomas-bayer.com/resources/rest/rest_webservices.pdf (Zugegriffen März 12, 2011).

Bielski, C., S. Gentilini, und M. Pappalardo. 2011. Post-Disaster Image Processing for Damage Analysis Using GENESI-DR, WPS and Grid Computing. Remote Sensing 3: 1234 - 1250.

Booth, D. et al. 2004. Web Service Architecture. http://www.w3.org/TR/ws-arch (Zugegriffen März 12, 2012).

Brauner, J. 2008. Web Processing Service Schnittstelle für GIS Funktionalitäten. http://tu-dresden.de/die_tu_dresden/fakultaeten/fakultaet_forst_geo_und_hydrowissenschaften/fachrichtung_geowissenschaften/gis/dateien/brauner_download/da_brauner.pdf (Zugegriffen März 12, 2012).

Burbiel, H. 2007. SOA & Webservices in der Praxis. Poing: Franzis.

Burghardt, M., und S. Hagenhoff. 2003. Web Services - Grundlagen und Kerntechnologien. http://webdoc.sub.gwdg.de/ebook/lm/arbeitsberichte/2003/22.pdf (Zugegriffen Februar 8, 2012).

Cepicky, J. 2011. Pywps Documentation. http://pywps.wald.intevation.org/ (Zugegriffen März 6, 2012).

Cerami, E. 2002. Web Services Essentials. Sebastopol, CA: O'Reilly Media.

Chang, H., und K. Lee. 2005. Applying Web Services and Design Patterns to Modeling and Simulating Real-World Systems. In: Artificial Intelligence and Simulation, vol. 3397, Hrsg. T. G. Kim, 351-359. Berlin, Heidelberg: Springer.

Chinnici, R., J.-J. Moreau, A. Ryman, und S. Weerawarana. 2007. Web Services Description Language (WSDL) Version 2.0. http://www.w3.org/TR/wsdl20/ (Zugegriffen März 12, 2012).

Chopra, V., S. Li, und J. Genender. 2011. Professional Apache Tomcat 6. Indianapolis, Indiana: John Wiley & Sons.

Christensen, E., F. Curbera, G. Meredith, und S. Weerawarana. 2001. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/2001/NOTE-wsdl-20010315 (Zugegriffen März 12, 2012).

Dedov, A. 2012. Libiqxmlrpc. http://libiqxmlrpc.wikidot.com/ (Zugegriffen März 12, 2012).

Django Software Foundation. 2012. Django. https://www.djangoproject.com/ (Zugegriffen März 12, 2012).

Düster, H. 2012. QGIS Web Processing Client. http://kappasys.org/cms/index.php?id=10 (Zugegriffen März 12, 2012).

Eclipse Foundation. 2010. Eclipse Indigo R Packages. http://www.eclipse.org/downloads/packages/release/indigo/r (Zugegriffen März 6, 2012).

Eclipse Foundation. 2011. Eclipse documentation. http://help.eclipse.org/indigo/index.jsp (Zugegriffen März 1, 2012).

FZW - Forschungszentrum Waldökosysteme der Georg-August-Universität Göttingen. 2012. Bioenergie-Regionen stärken (BEST) - Neue Systemlösungen im Spannungsfeld ökologischer, ökonomischer und sozialer Anforderungen. http://best-forschung.uni-goettingen.de/ (Zugegriffen März 12, 2012).

Feng, M., L. Shuguang, N. H. Euliss Jr., C. Young, und D. M. Mushet. 2011. Prototyping an online wetland ecosystem service model using open model sharing standards. Environmental Modelling & Software 26 (2011): 458 - 468.

Fielding, R. T. 2000. Architectural Styles and the Design of Network-based Software Architectures. http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (Zugegriffen März 12, 2012).

Foerster, T., A. Brühl, und B. Schäffer. 2011. RESTful Web Processing Service. http://ifgi.uni-muenster.de/~tfoer_01/articles/AGILE2011_Foerster_etal_RESTful_WPS.pdf (Zugegriffen März 12, 2012).

von Gadow, K. 2004. Wuchsmodelle für die Forsteinrichtung. In: Waldwachstumsmodelle für Prognosen in der Forsteinrichtung, Hrsg. M. Hanenwinkel und K. von Teuffel, Freiburg: Forstlische Versuchs- und Forschungsanstalt Baden-Württemberg: 1 - 14.

Gillett, C., C. Rutstein, G. Schreck, C. Buss, und H. Liddell. 2002. Organic IT. http://www.organic-computing.org/literature/Forrester_report.pdf (Zugegriffen März 12, 2012).

Goldhammer, J. 2010. Was sind Web Services? http://blog.fme.de/allgemein/2010-04/was-sind-web-services (Zugegriffen März 12, 2012).

Gudgin, M. et al. 2007. SOAP Version 1.2 Part 1. http://www.w3.org/TR/soap12-part1/ (Zugegriffen März 12, 2012).

Guha, R. 2008. Flexible Web Service Infrastructure for the Development and Deployment of Predictive Models. Journal of Chemical Information and Modeling 48: 456 - 464.

Haubrock, S., F. Theisselmann, H. Rotzoll, und D. Dransch. 2009. Web-based management of simulation models - concepts, technologies and the users needs. In: Proceeding of the 18th World IMACS Congress MODSIM09 International Congress on Modelling and Simulation, Hrsg. R. Anderssen, R. D. Braddock, und L. T. H. Newham, 880 - 886.

Heimann, D., J. Nieschulze, und B. König-Ries. 2010. A flexible statistics web processing service - Added value for information systems for experiment data. Journal of Integrative Bioinformatics 7: 140 - 155.

de Jesus, J., L. Casagrande, und J. Cepicky. 2010. PyWPS a tutorial for beginners and developers. http://www.slideshare.net/JorgeMendesdeJesus/pywps-a-tutorial-for-beginners-and-developers (Zugegriffen März 12, 2012).

Kaiser, P., und J. Ernesti. 2007. Python: Das umfassende Handbuch. Bonn: Galileo Computing.

Kurth, W., J. Saborowski, und J.C. Thiele. 2010. UP3: Räumliche Informationssysteme - Cluster UP: Umsetzung und Partizipation. http://best-forschung.uni-goettingen.de/wcm/wp-content/uploads/2010/12/UP_3_poster_Kurth_Thiele.pdf (Zugegriffen März 9, 2012).

Lang, J.-P. 2010. Quantum GIS Desktop. http://hub.qgis.org/projects/quantum-gis/wiki/DownloadDe (Zugegriffen März 9, 2012).

Laurent, S. St., E. Dumbill, und J. Johnston. 2001. Programming Web Services with XML-RPC. Sebastopol, CA: O'Reilly Media.

Lawrence, R. 2004. The space efficiency of XML. Information and Software Technology 46: 753 - 759.

Löwenstein, B., und O. Kraeft. 2011. Entwickeln von Webservices mit JAX-WS und JAX-RS. iX - Magazin für professionelle Infromationstechnik 8/2011: 38 - 44.

Melzer, I. 2008. Service-orientierte Architekturen mit Web Services: Konzepte - Standards - Praxis. Heidelberg: Spektrum Akademischer Verlag.

Michaelis, C. D., und D. P. Ames. 2009. Evaluation and Implementation of the OGC Web Processing Service for Use in Client-Side GIS. Geoinformatika 13: 109 - 120.

Microsoft. 2012. .NET Framework 4. http://msdn.microsoft.com/de-de/netframework/default.aspx (Zugegriffen März 12, 2012).

Nagel, J. 2003. TreeGrOSS eine Java basierte Softwarekomponente zur Waldwachstumsmodellierung für Forschung, Lehre und Praxis. In: Tagungsband der 15. Tagung der Sektion Forstliche Biometrie und Informatik des Deutschen Verbandes Forstlicher Forschungsanstalten, Freiburg 9.-10. Oktober 2003, Die grüne Reihe, Hrsg. U. Wunn und D. Quedenau: 33 - 37.

Nagel, J., und S. Sprauer. 2009. Langfristige Simulation der Zielstärkennutzung in Buchenbeständen. In: Deutscher Verband Forstlicher Forschungsanstalten. Sektion Ertragskunde. Jahrestagung 25.-27. Mai 2009: 159 - 164.

Nothdurft, A. 2007. Ein nichtlineares, hierarchisches und gemischtes Modell für das Baum-Höhenwachstum der Fichte (Picea abies (L.) Karst.) in Baden-Württemberg. Dissertation: Georg-August-Universität Göttingen

Novell. 2011. OpenSUSE Portal 11.4. http://de.opensuse.org/Portal:11.4 (Zugegriffen März 14, 2012).

Open Geospatial Consortium. 2007. OpenGIS Web Processing Service. http://www.opengeospatial.org/standards/wps (Zugegriffen März 12, 2012).

Oracle. 2011. Java SE 6 Documentation. http://docs.oracle.com/javase/6/docs/ (Zugegriffen März 7, 2012).

Oracle. 2012. Jersey. http://jersey.java.net/ (Zugegriffen März 9, 2012).

Ortel, J., J. Noehr, und N. Van Gheem. 2012. Suds. https://fedorahosted.org/suds/ (Zugegriffen März 10, 2012).

Pautasso, C., O. Zimmermann, und F. Leymann. 2008. RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision. http://www.jopera.org/files/www2008-restws-pautasso-zimmermann-leymann.pdf (Zugegriffen März 12, 2012).

Pretzsch, H., R. Grote, B. Reineking, Th. Rötzer, und St. Seifert. 2008. Models for Forest Ecosystem Management: A European Perspective. Annals of Botany 101: 1065 -1087.

Python Software Foundation. 2011. Python Programming Language - Official Website. http://www.python.org/ (Zugegriffen März 14, 2012).

Python Software Foundation. 2010. SimpleXMLRPCServer. http://www.python.org/doc//current/library/simplexmlrpcserver.html (Zugegriffen März 7, 2012).

Rails-Core-Team, D. 2012. Rails. http://rubyonrails.org/ (Zugegriffen März 12, 2012).

Richardson, L., und S. Ruby. 2007. Restful Web Services. Sebastopol, CA: O'Reilly Media.

Ristic, I. 2005. Apache security. Beijing [u.a.]: O'Reilly.

Ruby, S., G. Silver, J. A. Miller, J. Cardoso, und A. P. Sheth. 2002. Web Service Technologies and their Synergy with Simulation. In: Proceedings of the 2002 Winter Simulation Conference 606 - 615.

Schoneberg, S. 2011. „Ein raumbezogenes, klimasensitives, nichtlineares, hierarchisches gemischtes Modell für das Baum-Höhenwachstum der Fichte (Picea abies (L.) Karst.) in Reinbeständen im niedersächsischen Staatswald". Göttingen: Unveröffentlichte Masterarbeit, Georg-August-Universität Göttingen.

Schwenk, J. 2005. Sicherheit und Kryptographie im Internet: Von sicherer E-Mail bis zu IP-Verschlüsselung. Wiesbaden: Vieweg.

Sloboda, B. 1971. Zur Darstellung von Wachstumsprozessen mit Hilfe von Differentialgleichungen erster Ordnung. Mitteilungen der Baden-Württembergischen Forstlichen Versuchs- und Forschungsanstalt, 32. Freiburg.

Snell, J., D. Tidwell, und P. Kulchenko. 2002. Programming Web Services with SOAP. Sebastopol, CA: O'Reilly Media.

Stollberg, B., und A. Zipf. 2007. OGC web processing service interface for web service orchestration: aggregating geo-processing services in a bomb threat scenario. In: Proceedings of the 7th international conference on Web and wireless geographical information systems, W2GIS'07: 239 - 251.

Theisselmann, F., D. Dransch, und S. Haubrock. 2009. Service-oriented Architecture for Environmental Modelling - The Case of a Distributed Dike Breach Information System. In: Proceedings of the 18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation, Hrsg. R.S. Anderssen, R. D. Braddock, und L. T. H. Newham, 938 - 944.

Tilkov, S. 2009. Der bessere Web Service? Javamagazin 1.2009: 74 - 79.

Tilkov, S. 2011. REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien. Heidelberg: Dpunkt Verlag.

Tilkov, S., und P. Ghadir. 2006. REST: Die Architektur des Web. http://www.sigs.de/publications/os/2006/05/tilkov_ghadir_OS_05_06.pdf (Zugegriffen März 12, 2012).

Tiobe Software. 2012. TIOBE Programming Community Index for Februar 2012. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (Zugegriffen März 9, 2012).

Treegross Development. 2003. TreeGrOSS: Tree Growth Open Source Software. http://treegross.sourceforge.net/ (Zugegriffen März 12, 2012).

W3C. 2004. World Wide Web Consortium - Der technische Hintergrund. http://www.w3c.de/Flyer/OnePage%202004.pdf (Zugegriffen März 12, 2012).

Widjaja, T. 2010. Standardisierungsentscheidungen in mehrschichtigen Systemen: Untersuchung am Beispiel serviceorientierter Architekturen. Wiesbaden: Gabler Verlag.

Winer, D. 1999. XML-RPC Specification. http://xmlrpc.scripting.com/spec.html (Zugegriffen März 2, 2012).

ZOO Project. 2012. Zoo - Open WPS Platform. http://www.zoo-project.org/ (Zugegriffen März 9, 2012).

Zeppenfeld, K., und P. Finger. 2009. SOA und WebServices. Berlin, Heidelberg: Springer.

Anhang

Inhaltsverzeichnis Anhang

A Quellcode des SOAP-Webservices für das Modell Oberhöhe

A-1 SOAP-Server für Modell Oberhöhe

Aus dieser Klasse kann analog zum Treegross-Webservice mit Apache Axis 2 (vgl. Kapitel 3.2.1) der Server des Webservices erzeugt werden.

```
package ohws;

public class OberhoeheWS {
    public double getOberhoehe(int A, double SI) {
                double alpha = -0.0006;
                double delta = 1.507;
                double oberhoehe = SI * Math.pow(((1 - Math.exp(alpha * SI
                            *(double)A)) / (1 - Math.exp(alpha * 100.0 *
                            (double)A))), delta);
            return oberhoehe;
          }
}
```

A-2 SOAP-Client für Modell Oberhöhe

Diese Klasse stellt den SOAP-Client dar, mit der die Methode *getOberhoehe* des SOAP-Servers aufgerufen werden kann. Der Client und seine Erstellung entsprechen dem in Kapitel 3.2.2 gezeigten Client für das Modell Treegross.

```
package ohws;

import ohws.OberhoeheWSStub;
import ohws.OberhoeheWSStub.GetOberhoehe;
import ohws.OberhoeheWSStub.GetOberhoeheResponse;
import java.rmi.RemoteException;

public class TestClient {

    public static void main(String[] args) throws RemoteException{
        //New Stub:
        OberhoeheWSStub stub = new OberhoeheWSStub();
        GetOberhoehe run = new GetOberhoehe();
        //set parameters:
        run.setA(50);
        run.setSI(39.5);

        //run Web Service and get response:
        GetOberhoeheResponse res = stub.getOberhoehe(run);

        System.out.println(res.get_return());
    }
}
```

B Quellcode für das Modell Treegross

Mit der Klasse *TG_scheduler* wird das Wachstum eines gleichaltrigen Reinbestandes simuliert (vgl. Kapitel 3.2). Sie enthält die Methoden *runTG*, mit der die Simulation durchgeführt werden kann, und die Methode *calcu*, welche von der Methode *runTG* zur eigentlichen Berechnung des Wachstums genutzt wird.

```
package treews;

import treegross.base.*;
import treegross.treatment.*;
import java.net.*;


public class TG_scheduler {
            URL urlcodebase = null;

    public double [] runTG(int inage, int inspec, double inhg,
           double indg, double ing, int inperiod){

        Stand st = null;

        //create file:
        String localPath="";
        java.io.File fi = new java.io.File("");
        try{ localPath= fi.getCanonicalPath();
           } catch (Exception e){};

        try {

            //set Stand id:
            String idAlt="000";
            String id = "mein Wald";

            if (id.compareTo(idAlt)!= 0 ){
                if (idAlt.compareTo("000")!= 0) calcu(st, inperiod);
                idAlt=id;
                //create new Stand and SpeciesDefMap
                SpeciesDefMap SDM = new SpeciesDefMap();
                st = new Stand();

                //read from FileXMLSettings
                try {
                       SDM.readFromPath(st.FileXMLSettings);
                } catch (Exception ex) {
                   ex.printStackTrace();
                }

                //set Stand variables:
                st.setSDM(SDM);
                st.programDir=localPath;
                st.ntrees = 0;
                st.year = 2007;
                st.size = 0.25;
                st.standname=id;
                st.ingrowthActive=false;
                st.randomGrowthEffects=false;

                //set species:
                int art = inspec;

                //generate new forest:
                if (art > 100){ //art==0 free space
                    int alter = inage;
                    double hg = inhg;
                    double dg = indg;
                    double g = ing;
                    //generate Trees:
                    GenDistribution gdb = new GenDistribution();
                    if (hg > 0.0 && dg > 0.0 && g > 0.0){
                        gdb.weibull(st,art,alter,dg,hg,dg*1.3,g*st.
```

```
                            size);
                    SIofDistrib siod = new SIofDistrib();
                    siod.si(st,art,alter,dg,hg);

                    FunctionInterpreter fin = new
                          FunctionInterpreter();
                    for (int j=0;j<st.ntrees;j++){
                        if (st.tr[j].h==0.0) st.tr[j].h=fin.
                                getValueForTree(st.tr[j],st.tr[j].
                                sp.spDef.uniformHeightCurveXML);
                    }
                    //generate missing data of trees
                    st.missingData();

                  //generate tree coordinates:
                  GenerateXY gxy=new GenerateXY();
                  gxy.zufall(st);
                  st.sortbyd();
                }
            }
      }

   } catch (Exception e) {e.printStackTrace();}
   //set the site index for all species
   st.setSiteIndex(0, 60.0);

   //grow and treat Stand:
   if (st.ntrees>0) {
       calcu(st, inperiod);
   }

   //prepare Array for return:
   double[] returnarray = new double[4];
   returnarray[0] = st.sp[0].h100age; //age
   returnarray[1] = st.hg;//hg
   returnarray[2] = st.dg;//dg
   returnarray[3] = st.bha;//g

   return returnarray;
}


static void calcu(Stand standhere, int period){
    standhere.descspecies();

    try {
        //notice volume of species at beginning:
        for (int i=0; i<standhere.nspecies;i++){
            standhere.sp[i].totalPrice=standhere.sp[i].vol;
        }

        //settings for treatment and simulation:
        standhere.trule.thinningIntensity=1.0;
        for (int j=0; j< standhere.nspecies; j++){
             TreatmentElements2 te2 = new TreatmentElements2();
             standhere.sp[j].trule.numberCropTreesWanted=(int)
                     Math.round(te2.numberOfCropTrees(standhere.
                     sp[j],standhere.sp[j].trule.targetDiameter,
             standhere.sp[j].trule.targetCrownPercent));
             standhere.sp[j].trule.targetCrownPercent=standhere.
                         sp[j].percCSA;
             standhere.sp[j].trule.thinningIntensity=standhere.
                     trule.thinningIntensity;
        }
```

```
            //Set single tree selection:
            standhere.trule.typeOfThinning=0;
            standhere.trule.thinArea=false;
            standhere.trule.selectCropTrees=true;
            standhere.trule.reselectCropTrees=true;
            standhere.trule.selectCropTreesOfAllSpecies=false;
            standhere.trule.releaseCropTrees=true;
            standhere.trule.cutCompetingCropTrees=false;
            standhere.trule.releaseCropTreesSpeciesDependent =true;
            standhere.trule.minThinningVolume=0;
            standhere.trule.maxThinningVolume=60;
            standhere.trule.thinAreaSpeciesDependent =true;
            standhere.trule.thinningIntensityArea=0.0;
            standhere.trule.minHarvestVolume=0.0;
            standhere.trule.maxHarvestVolume=120.0;
            standhere.trule.typeOfHarvest=0;
            standhere.trule.harvestLayerFromBelow =false;
            standhere.trule.maxHarvestingPeriode=6;
            standhere.trule.lastTreatment=0;

            Treatment2 treatment2 = new Treatment2();
            treatment2.executeManager2(standhere);

            //growth and treatment:
            for (int x=1; x < period; x++){
                standhere.grow(5,false);
                standhere.descspecies();
                   treatment2.executeManager2(standhere);
                   //printing:
            }

            //growth:
            standhere.grow(5,false);
            standhere.descspecies();

            //Calculate Results:
            for (int i=0; i<standhere.nspecies;i++){
                double vorn=0.0;
                double zielst=0.0;
                double tote =0.0;
                for (int j=0; j<standhere.ntrees;j++){
                    if (standhere.tr[j].code==standhere.sp[i].code
                            && standhere.tr[j].out > 0 ){
                        if (standhere.tr[j].outtype==2) vorn = vorn
                                +standhere.tr[j].v*standhere.tr[j].
                                fac/standhere.size;
                        if (standhere.tr[j].outtype==3) zielst =
                                zielst +standhere.tr[j].v*standhere.
                                tr[j].fac/standhere.size;
                        if (standhere.tr[j].outtype==1) tote = tote +
                                standhere.tr[j].v*standhere.tr[j].
                                fac/standhere.size;
                    }
                }
            }
        }
    catch (Exception e) {
        System.out.println(e);
    }
   }
}
```

C Quellcode des REST-Webservices für das Modell Oberhöhe

Der Quellcode für das Modell Oberhöhe setzt sich aus drei Klassen zusammen. Die Klasse *Output* beschreibt die mit dem Webservice angebotene Ressource. Die Klasse *OuputResource* stellt den eigentlichen Webservice dar. In ihr wird festgelegt, an welcher URL der Webservice zu erreichen ist und welche HTTP-Anfragen mit welcher Methode beantwortet werden. Mit der dritten Klasse *Client* wird eine beispielhafte Nutzung des Webservices demonstriert.

C-1 Klasse für die Output-Ressource

Diese Java-Klasse beschreibt die Output-Ressource des Webservices mit ihren Attributen und Konstruktoren. Innerhalb des als Zweites gezeigten Konstruktors wird die eigentliche Modellberechnung durchgeführt.

```
package restoberhoehe.objekte;

import javax.xml.bind.annotation.XmlRootElement;

//@XmlRootElement to map Java class to XML or JSON representation
@XmlRootElement
public class Output {
    public double oberhoehe;
    public int alter;
    public double SI;
    //constructors:
    public Output(){}

    //constructor, which calculates „oberhoehe“ from age and SI:
    public Output(int A, double SI){
        this.alter = A;
        this.SI = SI;

        double alpha = -0.0006;
        double delta = 1.507;
        this.oberhoehe = SI * Math.pow(((1 - Math.exp(alpha * SI
                                *(double)A)) / (1 - Math.exp(alpha*100.0*
                                (double)A))), delta);
}
    public Output(int A, double SI, double oberhoehe) {
        this.alter = A;
        this.SI = SI;
        this.oberhoehe = oberhoehe;
    }
}
```

C-2 Klasse für Anfragen an die Output-Ressource

Diese Java-Klasse stellt den eigentlichen Webservice dar. Sie legt fest, an welcher URL auf die Ouput-Ressource zugegriffen werden kann und welche Methoden bei einer bestimmten HTTP-Anfrage ausgeführt werden. Der Webservice kann HTTP-GET-Anfragen bearbeiten, um eine Repräsentation der Ressource auszuliefern. Dafür stehen die Methoden *getresultJSON* und *getresultxml* zur Verfügung, welche beide zu diesem Zweck mit einer @GET-Annotation versehen sind. Erstgenannte wird ausgeführt, wenn an die URL eine HTTP-GET-Anfrage gesendet wird, welche den Datentyp JSON anfordert. Dies wird durch

die @Produces-Annotation erreicht. Die zweite Methode beantwortet beispielsweise HTTP-GET-Anfragen, welche über die Adresszeile des Web-Browsers abgesendet werden können. Mit diesen beiden Methoden ist es also möglich, verschiedene Repräsentationen der Ressource auszuliefern. Außerdem bearbeitet diese Klasse HTTP-PUT-Anfragen mit der Methode *postOberhoehe*, um eine neue Berechnung des Modells für bestimmte Eingangsdaten, die mit der HTTP-PUT-Anfrage mitgeliefert werden müssen, durchzuführen.

```
package restoberhoehe.resources;

import javax.ws.rs.Consumes;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import restoberhoehe.objekte.*;
//import java.io.File;
//import java.io.FileWriter;
//import java.io.IOException;
import java.io.*;

//@Path: Sets the path to base URL + /Inputs
@Path("/Output")
public class OutputResource {

    // This method is called if HTML is requested (e.g. Browser):
    @GET
    @Produces({MediaType.TEXT_XML})
    public Output getresultxml() throws IOException{

            FileReader file = new FileReader("datenspeicher.txt");
            BufferedReader bufRead = new BufferedReader(file);

            double oberhoehe = Double.parseDouble(bufRead.readLine());
            int A = Integer.parseInt(bufRead.readLine());
            double SI = Double.parseDouble(bufRead.readLine());
            Output output = new Output(A, SI, oberhoehe);

        return output;
    }
    // This method is called if JSON-Format is requested:
    @GET
    @Produces({MediaType.APPLICATION_JSON})
    public Output getresultjson() throws IOException{
        //save data for the resource to a file:
        FileReader file = new FileReader("datenspeicher.txt");
        BufferedReader bufRead = new BufferedReader(file);

        double oberhoehe = Double.parseDouble(bufRead.readLine());
        int A = Integer.parseInt(bufRead.readLine());
        double SI = Double.parseDouble(bufRead.readLine());
        Output output = new Output(A, SI, oberhoehe);

        return output;
    }
    //@PUT: method will perform if HTTP-PUT is requested
    //@Consumes: specifies the MIMEtype, the resource can handle
    @PUT
    @Consumes("application/x-www-form-urlencoded")
```

```
    @Produces(MediaType.TEXT_HTML)
    public void postOberhoehe(@FormParam("alter") int alter,
            @FormParam("SI") double SI){
        //create new observation or update old one:
        Output output = new Output(alter, SI);

        // save results to a file:
        try{
            File file = new File("datenspeicher.txt");
            FileWriter fw = new FileWriter(file);
            fw.write(((Double)output.oberhoehe).toString() +"\n"
                    + output.alter +"\n"
                    + ((Double)output.SI).toString());
            fw.flush();
            fw.close();
        }
        catch( IOException e ){
            e.printStackTrace();
        }
    }
}
```

C-3 REST-Client für das Modell Oberhöhe

Der Client für den REST-Webservice führt folgende Anfragen durch: 1. Die Modell-Berechnung wird mittels einer HTTP-PUT-Anfrage an die URL der Output-Ressource des Webservices ausgeführt. 2. Das Ergebnis des Durchlaufs wird mit HTTP-GET vom Webservice abgeholt und auf der Konsole ausgegeben.

```
package restoberhoehe.client;

import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.Reader;
import java.io.StringWriter;
import java.io.Writer;
import java.net.HttpURLConnection;
import java.net.URL;


public class Client {

    public static void main(String[] args) throws Exception{
        Client client = new Client();
        //execute Model:
        client.putOutput(50, 39.5);
        //get the results:
        client.get("http://localhost:8080/restoberhoehe/rest/"
                + "Output");
    }
    // Method to trigger the Web Service to calculate Model:
    public void putOutput(int alterIn, double SI)
            throws Exception {

        URL url = new URL("http://localhost:8080/restoberhoehe/rest/"+
                "Output");
        //connect to REST-Server:
        HttpURLConnection httpCon = (HttpURLConnection) url.
                openConnection();
```

```
        httpCon.setDoInput(true);
        httpCon.setDoOutput(true);
        httpCon.setRequestMethod("PUT");// choose HTTP-method
        httpCon.connect();
        //set the input-data:
        OutputStream outStream = httpCon.getOutputStream();
        OutputStreamWriter writer = new OutputStreamWriter(outStream);
        writer.write("&alter=" + alterIn + "&SI=" + SI);
        writer.flush();
        writer.close();
        outStream.close();
        //getInputStream to actually send data:
        httpCon.getInputStream();
        httpCon.disconnect();
    }

    //method to get a JSON-representation of the result:
    public void get(String path) throws Exception {

        URL url = new URL(path);
        HttpURLConnection httpCon = (HttpURLConnection)
                url.openConnection(); //open connection to server
        httpCon.setDoInput(true); // set "DoInput" to read from url
        //RequestProperty to get data as JSON-Format:
        httpCon.setRequestProperty("Content-Type","application/json");
        httpCon.setRequestProperty("Accept", "application/json");
        httpCon.setRequestMethod("GET");// choose HTTP-method

        //read the response and cause the request to actually be send:
        InputStream inStream = httpCon.getInputStream();

        // convert InputStream for printing:
        Client client = new Client();
        System.out.println(client.
                InputStreamToString(inStream));
        httpCon.disconnect(); //close connection to server
}
    //Method to convert received data to String
    public String InputStreamToString(InputStream inStream)
            throws Exception {
        Writer writer = new StringWriter();
        char[] buffer = new char[1024];

        Reader reader = new BufferedReader(new InputStreamReader(
                inStream, "UTF-8"));
        int n;
        while ((n = reader.read(buffer)) != -1) {
            writer.write(buffer, 0, n);
}
        inStream.close();
        return writer.toString();
    }
}
```

D Quellcode des REST-Webservices für das lineare Regressionsmodell

Der Quellcode für den in Kapitel 3.3 gezeigten REST-Webservice setzt sich zusammen aus elf verschiedenen Klassen und einer Klasse für den REST-Client. Auf die Aufgabe jeder dieser Klassen wird in den jeweiligen Unterkapiteln näher eingegangen.

D-1 Datenspeicherung Inputliste

Diese Enumeration wird für die Speicherung aller Beobachtungen bzw. der Eingangsdaten der linearen Regression verwendet.

```
package restmodell.daten;

import java.util.TreeMap;
import java.util.Map;
import restmodell.objekte.*;


//Enum adapted from http://www.vogella.de/articles/REST/article.html
public enum Inputliste {
    instance;

        //Java Hash Map to save data:
        public Map<String, Beobachtung> contentProvider = new
                TreeMap<String, Beobachtung>();

        //constructor
        private Inputliste() {
            this.insert("1", 46, 25.5);
            this.insert("3", 67, 27.0);
            this.insert("4", 31, 20.0);
            this.insert("6", 57, 25.0);
            this.insert("12", 101, 40.5);
            this.insert("18", 82, 38.2);
        }

        //method to create observations:
        public void insert(String id, int alter, double bhd){
            Beobachtung beobachtung = new Beobachtung(id, alter, bhd);
            contentProvider.put(id, beobachtung);
        }

        //getter
        public Map<String, Beobachtung> getModel(){
            return contentProvider;
        }
}
```

D-2 Datenspeicherung Outputliste

Diese Enumeration wird zur Speicherung der Modellgleichung bzw. der Modellparameter verwendet.

```
package restmodell.daten;

import java.util.Map;
import java.util.TreeMap;
import restmodell.objekte.*;

//Enum adapted from http://www.vogella.de/articles/REST/article.html
public enum Outputliste {
    instance;

    //Java Hash Map to save data:
    public Map<String, Modell> contentProvider = new
            TreeMap<String, Modell>();

    private Outputliste() {
        this.insert("1", "linear Modell",
```

```
                "y = a * x + b");
    }

    //method to insert data:
    public void insert(String id, String art, String gleichung){
        Modell modell = new Modell(id, art, gleichung);
        contentProvider.put(id, modell);
    }

    //getter:
    public Map<String, Modell> getModel(){
        return contentProvider;
    }
}
```

D-3 Klasse für eine Beobachtung

Diese Klasse beschreibt bzw. steht für eine Beobachtungs-Ressource, mit ihren Variablen und Konstruktoren.

```
package restmodell.objekte;

import javax.xml.bind.annotation.XmlRootElement;

//@XmlRootElement to map Java class to XML representation
@XmlRootElement
public class Beobachtung {

        public String id;
        public int alter;
        public double bhd;

        //constructors:
        public Beobachtung(){

        }
        public Beobachtung (String id, int alter, double bhd){
            this.id = id;
            this.alter = alter;
            this.bhd = bhd;
        }
}
```

D-4 Klasse für Modell-Ressource

Diese Klasse beschreibt die Ressource für die Modellausgabe.

```
package restmodell.objekte;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Modell {
    public String id;
    public String art;
    public String gleichung;

    //constructors:
    public Modell(){

    }
    public Modell(String id, String art, String gleichung){
        this.id=id;
        this.art=art;
```

```
        this.gleichung=gleichung;
    }
}
```

D-5 Methoden für die XML-Erstellung

Diese Klasse (*DeliverXML*) enthält Methoden, um XML-Dokumente für die Antwort auf HTTP-GET-Anfragen des REST-Webservice-Nutzers zu erzeugen. Die Methoden dieser Klasse werden in den Klassen verwendet, die zur Beantwortung der HTTP-Anfragen (Anhang Kapitel D-7, D-9 und D-11) dienen. Die Klasse *DeliverXML* enthält drei Methoden, welche zur Erstellung der zwei XML-Dokumente aus Abbildung 6 (*XMLcreateServiceAccess*) und 7 (*XMLcreateB*) aus Kapitel 3.3.1 dienen, sowie für die Erstellung des XML-Dokuments, welches eine Übersicht zu den verfügbaren Modellen liefert (*XMLcreateM*).

```
package restmodell.objekte;

import java.io.File;
import java.io.FileWriter;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;
import java.util.List;


public class DeliverXML {

    public File XMLcreateB(List<Beobachtung> liste) throws Exception{
        //create XML-document:
        Document doc = new Document();
        Element root = new Element("Inputs");//create root-element
        doc.setRootElement(root);

        //add elements with links and content:
        for(int i=0; i<liste.size(); i++){
            Element element = new Element(liste.get(i).getClass().
                    toString().substring(25));
            element.setAttribute("link", "http://localhost:8080/" +
                    "restmodell/rest/Inputs/" + liste.get(i).id);
            element.addContent(liste.get(i).id);
            root.addContent(element);
        }

        //Save document to file:
        String dateiname = "Liste der Beobachtungen.xml";
        File datei = new File(dateiname);
        if (datei.exists()){
            datei.delete();//Delete content of file
            datei.renameTo(new File(dateiname));
        }
        FileWriter writer = new FileWriter(datei, true);
        XMLOutputter outp = new XMLOutputter();
        outp.setFormat(Format.getPrettyFormat());
        outp.output(doc, writer);//give doc to the writer

        return datei;
        }

    public File XMLcreateM(List<Modell> liste) throws Exception{
```

```
        //create xml-document:
        Document doc = new Document();
        Element root = new Element("Outputs");//create root-element
        doc.setRootElement(root);

        //create elements:
        for(int i=0; i<liste.size(); i++){
            Element element = new Element(liste.get(i).
                    getClass().toString().substring(25));
            element.setAttribute("link", "http://localhost:8080/" +
                    "restmodell/rest/Outputs/" + liste.get(i).id);
            element.addContent(liste.get(i).id);
            root.addContent(element);
        }
        //save doc to file:
        String dateiname = "Liste der Modelle.xml";
        File datei = new File(dateiname);
        if (datei.exists()){
            datei.delete();//delete content of file
            datei.renameTo(new File(dateiname));
        }
        FileWriter writer = new FileWriter(datei, true);
        XMLOutputter outp = new XMLOutputter();
        outp.setFormat(Format.getPrettyFormat());
        outp.output(doc, writer);

        return datei;
    }

    public File XMLcreateServiceAccess ()throws Exception{
        //create xml-document:
        Document doc = new Document();
        Element root = new Element("RESTmodell");//create root element
        doc.setRootElement(root);

        //create elements with links and content:
        Element element1 = new Element("Inputs");
        element1.setAttribute("link", "http://localhost:8080/" +
                "restmodell/rest/Inputs/");
        element1.addContent("Inputs");
        root.addContent(element1);
        Element element2 = new Element("Outputs");
        element2.setAttribute("link", "http://localhost:8080/" +
                "restmodell/rest/Outputs/");
        element2.addContent("Outputs");
        root.addContent(element2);

        //save doc to a file:
        String dateiname = "ServiceAccess.xml";
        File datei = new File(dateiname);
        if (datei.exists()){
            datei.delete();//delete content if file allready exists
            datei.renameTo(new File(dateiname));
        }
        FileWriter writer = new FileWriter(datei, true);
        XMLOutputter outp = new XMLOutputter();
        outp.setFormat(Format.getPrettyFormat());
        outp.output(doc, writer);

        return datei;
    }
}
```

D-6 Klasse für die Durchführung der linearen Regression

In dieser Klasse *LinearRegression* ist die Methode *compute* enthalten, welche für die Anpassung des linearen Modells an die Beobachtungen verwendet wird. Diese Methode wird in der Klasse *ModellResource* aus Kapitel D-10 verwendet.

```
package restmodell.objekte;

/*Class LinearRegression adapted from:
 * http://www.koders.com/java/fid9CE2040E2918F56FAAE00E5C58C34
 * 479304E24FC.aspx?s=linear+regression
 * */

public class LinearRegression {

    public static void main(String[] args) {
        //An example:
        double[] x = {20, 16, 15, 16, 13, 10};
        double[] y = {0, 3, 7, 4, 6, 10};
        LinearRegression instanz = new LinearRegression();
        System.out.println(instanz.compute(x, y,"x","y"));
    }
    //constructor:
    public LinearRegression() {

    }
    //Perform linear regression:
    public String compute(double[] x, double[] y,
            String x_name, String y_name){

        double meanX, meanY, slope, intercept;
        double n = x.length;
        double sumy = 0.0;
        double sumx = 0.0;
        double sumx2 = 0.0;
        double sumxy = 0.0;

        for (int i = 0; i < n; i++) {
            sumx += x[i];
            sumx2 += x[i] * x[i];
            sumy += y[i];
            sumxy += x[i] * y[i];
        }
        meanX = sumx / n;
        meanY = sumy / n;
        slope = (sumxy - sumx * meanY) / (sumx2 - sumx * meanX);
        intercept = meanY - slope * meanX;

        java.text.DecimalFormat Num = new java.text.
                DecimalFormat("#.000");
        return y_name +" = " + Num.format(slope)+" * " + x_name +
                " + " +Num.format(intercept);
    }
}
```

D-7 Klasse zur Bearbeitung von Anfragen an die Übersicht aller Beobachtungs-Ressourcen

Diese Klasse wird an der URL http://localhost:8080/restmodell/rest/Inputs erreicht. Sie enthält verschiedene Methoden, um mit den Eingangsdaten des Modells über HTTP-Anfragen

zu interagieren. Beispielsweise wird bei einer HTTP-POST-Anfrage an die oben genannte URL die Methode *newBeob* aufgerufen, zur Erzeugung einer neuen Beobachtung, die dann in die Liste über alle Beobachtungen eingefügt wird. Bei einer HTTP-GET-Anfrage wird eine Übersicht aller Beobachtungen ausgeliefert. Daneben leitet sie HTTP-Anfragen an die Klasse *BeobachtungResource* weiter, wenn die genannte URL mit der ID einer der Beobachtungen erweitert wurde. Dadurch kann eine einzelne Ressource manipuliert werden. Die dafür zuständigen Methoden sind in der Klasse *BeobachtungRessource* (Anhang: Kapitel D-8) enthalten.

```
package restmodell.resources;

import java.util.ArrayList;
import java.util.List;
import javax.ws.rs.Consumes;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.UriInfo;
import restmodell.daten.*;
import restmodell.objekte.*;
import java.io.File;
import java.io.IOException;

//Class adapted from http://www.vogella.de/articles/REST/article.html
//@Path: Sets the path to base URL + /Inputs
@Path("/Inputs")
public class Inputs {

    // @Context: Allows to insert contextual objects into the class,
    // e.g. ServletContext, Request, Response, UriInfo
    @Context
    UriInfo uriInfo;
    Request request;

    //@Produces: This method is called if XML is requested
    //@GET: Method is called if HTTP-method is HTTP-GET
    @GET
    @Produces(MediaType.TEXT_XML)
    public File XMLBeobliste() throws Exception{

        List<Beobachtung> daten = new ArrayList<Beobachtung>();
        daten.addAll( Inputliste.instance.getModel().values() );
        DeliverXML deliver = new DeliverXML();

        return deliver.XMLcreateB(daten);
    }

    @GET
    // This method is called if HTML is requested (e.g. Browser):
    @Produces(MediaType.APPLICATION_XML)
    public File HtmlBeobliste() throws Exception{
```

```
        List<Beobachtung> daten = new ArrayList<Beobachtung>();
        daten.addAll( Inputliste.instance.getModel().values() );
        DeliverXML deliver = new DeliverXML();

        return deliver.XMLcreateB(daten);
    }

    //@Path("{beob}"): method requested at: base URL + /Inputs/ + id
    @Path("{beob}")
    //@PathParam: take value form URI
    public BeobachtungResource getBeob(@PathParam("beob") String id) {
        //redirect to class "BeobachtungResource"
        return new BeobachtungResource(uriInfo, request, id);
    }

    //@POST: method is called, if HTTP-method is HTTP-POST
    //@Consumes: MIMEtype, which is required for this mehtod
    @POST
    @Produces(MediaType.TEXT_HTML)
    @Consumes("application/x-www-form-urlencoded")
    public void newBeob(
            //@FormParam: receive parameters from HTTP-POST
            @FormParam("alter") int alter,
            @FormParam("bhd") double bhd
    ) throws IOException {

        //create new Observation at first id, which is not occupied
        String newId="100";
        boolean found=false;
        Integer i = 1;

        //search for first id, which is not occupied
        do{
            if(Inputliste.instance.getModel().
                    get(i.toString())==null){
                newId=i.toString();
                found = true;
            }
            if(i==100) found = true;
            i++;
        } while(!found);

        //create new Observation
        Beobachtung beobachtung = new Beobachtung(newId, alter, bhd);
        Inputliste.instance.getModel().put(newId, beobachtung);
    }
}
```

D-8 Klasse zur Bearbeitung von Anfragen an eine einzelne Beobachtungs-Ressource

Diese Klasse bzw. ihre Methoden werden aufgerufen, wenn an die URL http://localhost:8080/restmodell/rest/Inputs/ eine ID angehängt wird. Sie enthält beispielsweise eine Methode zum Löschen einer Beobachtung (*deletBeob*), welche aufgerufen wird, wenn eine HTTP-DELETE-Anfrage gesendet worden ist, oder eine Methode für das Ausgeben einer Beobachtung bei einer HTTP-GET-Anfrage (*getBeob*).

```
package restmodell.resources;

import javax.ws.rs.Consumes;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
```

```
import javax.ws.rs.DELETE;
import javax.ws.rs.PUT;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.UriInfo;
import restmodell.objekte.*;
import restmodell.daten.*;


public class BeobachtungResource {
    //@Context to inject information from request
    @Context
    UriInfo uriInfo;
    Request request;
    String id;

    //constructor:
    public BeobachtungResource(UriInfo uriInfo, Request request,
                String id) {
        this.uriInfo = uriInfo;
        this.request = request;
        this.id = id;
    }
    //@GET: state, that method will process HTTP-GET-requests
    //@Produces: specify MIMEtype(s) which will be send back to client
    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Beobachtung getBeob() {
        Beobachtung beobachtung = Inputliste.instance.getModel().
                get(id);
        if(beobachtung==null)
            throw new RuntimeException("Get: Beob with " + id +
                    " not found");
        return beobachtung;
        }
    //@DELETE: this method will be performed, if HTTP-DELETE
    //is requested
    @DELETE
    public void deleteBeob(){
        //delte observation:
        Inputliste.instance.getModel().remove(id);
    }

    //@PUT: method will perform if HTTP-PUT is requested
    //@Consumes: specifies the MIMEtype, the resource can handle
    @PUT
    @Consumes("application/x-www-form-urlencoded")
    @Produces(MediaType.TEXT_HTML)
    public void putBeobachtung(@FormParam("alter") int alter,
            @FormParam("bhd") double bhd){
        //create new observation or update old one:
        Beobachtung beobachtung = new Beobachtung(id, alter, bhd);
        Inputliste.instance.getModel().put(id, beobachtung);
    }
}
```

D-9 Klasse zur Bearbeitung von Anfragen an die Übersicht der verfügbaren Modelle

Diese Klasse wird über die URL http://localhost:8080/restmodell/rest/Outputs erreicht. Es sind zwei Methoden enthalten, die beide bei HTTP-GET-Anfragen an die URL ausgeführt

werden, wie z.B. die Methode *XMLModelliste*, welche eine XML-Repräsentation dieser Ressource zurückgibt. Die Antwort enthält dann eine Übersicht über alle Modelle, die angepasst werden können, welche für diesen REST-Webservice aber nur aus einem Modell besteht. Daneben leitet diese Klasse Anfragen an die Klasse *ModellResource* weiter, wenn die URL dieser Klasse um die ID für das Modell erweitert wird.

```
package restmodell.resources;

import java.util.ArrayList;
import java.util.List;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.UriInfo;
import restmodell.daten.*;
import restmodell.objekte.*;
import java.io.File;

//Class adapted from http://www.vogella.de/articles/REST/article.html
@Path("/Outputs")
public class Outputs {

        @Context
        UriInfo uriInfo;
        Request request;

        //method is performed if XML is requested:
        @GET
        @Produces(MediaType.TEXT_XML)
        public File XMLModelliste() throws Exception{

            List<Modell> daten = new ArrayList<Modell>();
            daten.addAll( Outputliste.instance.getModel().values() );
            DeliverXML deliver = new DeliverXML();

            return deliver.XMLcreateM(daten);
        }

        //method is performed if HTML is requested (e.g. Browser)
        @GET
        @Produces(MediaType.APPLICATION_XML)
        public File HtmlModellliste() throws Exception{

            List<Modell> daten = new ArrayList<Modell>();
            daten.addAll( Outputliste.instance.getModel().values() );
            DeliverXML deliver = new DeliverXML();

            return deliver.XMLcreateM(daten);

        }

        @Path("{modell}")
        public ModellResource getBeob(@PathParam("modell")String id) {
            //redirect to class "ModellResource"
            return new ModellResource(uriInfo, request, id);
        }
}
```

D-10 Klasse zur Bearbeitung von Anfragen an die Modell-Ressource

Diese Klasse bzw. ihre Methoden werden aufgerufen, wenn an die URL http://localhost:8080/restmodell/rest/Outputs/ eine ID angehängt wird. Sie dient dazu, das lineare Regressionsmodell zu berechnen und das Ergebnis dieser Berechnung zu speichern. Die Berechnung der linearen Regression wird durch die Methode *putModell* aufgerufen, wenn eine HTTP-PUT-Anfrage an die URL gesendet worden ist. Diese Methode sorgt dann dafür, dass das Modell für alle gegebenen Eingangsdaten neu berechnet wird. Ausgeliefert wird das Ergebnis mit der Methode *getModell*, welche bei einer HTTP-GET-Anfrage an die URL der Klasse ausgeführt wird.

```
package restmodell.resources;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.UriInfo;
import restmodell.objekte.*;
import restmodell.daten.*;

//Class adapted from http://www.vogella.de/articles/REST/article.html
public class ModellResource {

    @Context
    UriInfo uriInfo;
    Request request;
    String id;

    //constructor:
    public ModellResource(UriInfo uriInfo, Request request,
            String id) {
        this.uriInfo = uriInfo;
        this.request = request;
        this.id = id;
    }

    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Modell getModell() {
        Modell modell = Outputliste.instance.getModel().get(id);
        if(modell==null)
            throw new RuntimeException("Get: Modell with " + id +
                    " not found");
        return modell;
    }

    @PUT
    @Consumes("application/x-www-form-urlencoded")
    @Produces(MediaType.TEXT_HTML)
    public void putModell(){

        //put only for existing model applicable:
            if(Outputliste.instance.getModel().get(id) == null)
            throw new RuntimeException("Modell with id= " + id +
```

```
                    " not found. " + "Put only possible for" +
                            " existing model(s)");
        //calculate new Parameters for model:

        //create arrays for calculation:
        double [] x = new double[Inputliste.instance
                                 .getModel().size()];
        int i = 0;
        for (Beobachtung b: Inputliste.instance.getModel().values()) {
            x[i] = (double)b.alter;
            i++;
        }
        double [] y = new double[Inputliste.instance.
                                 getModel().size()];
        i = 0;
        for (Beobachtung b: Inputliste.instance.getModel().values()) {
            y[i] = b.bhd;
            i++;
        }

        //perform Linear regression, based on created arrays:
        LinearRegression regression = new LinearRegression();
        regression.compute(x, y, "alter", "bhd");

        Modell modell= new Modell(id, Outputliste.instance.getModel().
                get(id).art,regression.compute(x, y, "alter", "bhd"));
        Outputliste.instance.getModel().put(id, modell);

    }
}
```

D-11 Klasse zur Bearbeitung von Anfragen an den REST-Webservice-Eintrittspunkt

Mit dieser Klasse werden HTTP-GET-Anfragen an den Eintrittspunkt des Webservices (URL: http://localhost:8080/restmodell/rest) beantwortet. Dafür stehen die Methoden *HtmlServiceAccess* und *XMLServiceAccess* zur Verfügung, die je nach angefordertem Datentyp eine Repräsentation der Ressource zurückgeben.

```
package restmodell.resources;

import java.io.File;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import restmodell.objekte.DeliverXML;


@Path("/")
public class ServiceAccess {

    //method is performed if HTML is requested
    @GET
    @Produces(MediaType.APPLICATION_XML)
    public File HtmlServiceAccess() throws Exception{
        DeliverXML deliver = new DeliverXML();
        return deliver.XMLcreateServiceAccess();

    }
    //method is performed if XML is requested
    @GET
    @Produces(MediaType.TEXT_XML)
```

```
    public File XMLServiceAccess() throws Exception{
        DeliverXML deliver = new DeliverXML();
        return deliver.XMLcreateServiceAccess();
    }
}
```

D-12 Client für den REST-Webservice

In der Main-Methode dieser Klasse werden verschiedene HTTP-Anfragen an den REST-Webservice abgesendet, die eine beispielhafte Nutzung des Webservices darstellen. Für die dafür verwendeten HTTP-Anfragen wurden jeweils eigene Methoden erstellt.

```
package restmodell_client.execute;

import java.net.*;
import java.io.*;
import java.io.InputStream;
import java.io.Writer;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class HttpRequests {
    public static void main(String[] args) throws Exception {
        HttpRequests httprequests = new HttpRequests();
        // 1. get-request for entry-point:
        httprequests.get("http://localhost:8080/restmodell/rest/");
        // 2. get-request for input-list:
        httprequests.get("http://localhost:8080/restmodell/" +
                "rest/Inputs");
        // 3. post-request to create new input-resource:
        httprequests.post(55, 21.5);
        // 4. put-request to update an input-resource:
        httprequests.putObservation("1", 85, 40.0);
        // 5. delete Resource:
        httprequests.delete("18");
        // 6. update model:
        httprequests.putModel();
        // 7. retrieve model-equation:
        httprequests.get("http://localhost:8080/restmodell/rest/" +
                "Outputs/1");
    }

    public void get(String path) throws Exception {
        URL url = new URL(path);
        HttpURLConnection httpCon = (HttpURLConnection)
                url.openConnection(); //open connection to server
        httpCon.setDoInput(true); // set "DoInput" to read from url
        httpCon.setRequestMethod("GET");// choose http-method
        //read the response and cause the request to actually be send:
        InputStream inStream = httpCon.getInputStream();
        // convert InputStream for printing:
        HttpRequests httpRequests = new HttpRequests();
        System.out.println(httpRequests.
                InputStreamToString(inStream));
        httpCon.disconnect(); //close connection to server
    }
    // Method to create new Input-Resource, Web Service determines
    //  new URI and ID:
    public void post(Integer ageIn, Double DBHIn) throws Exception {
        URL url = new URL("http://localhost:8080/restmodell/" +
                "rest/Inputs");
        HttpURLConnection httpCon = (HttpURLConnection) url.
                openConnection();
```

```
        httpCon.setDoInput(true);// read from URL
        httpCon.setDoOutput(true);// write to URL
        //modify general request properties:
        httpCon.setRequestProperty("Content-Type",
                "application/x-www-form-urlencoded");
        httpCon.setRequestMethod("POST");
        //retrieve the output to send data to the server:
        OutputStream outStream = httpCon.getOutputStream();
        // send data with http-POST:
        OutputStreamWriter writer = new OutputStreamWriter(outStream);
        writer.write("&alter=" + ageIn + "&bhd=" + DBHIn);
        writer.flush();
        writer.close();
        outStream.close();
        httpCon.getInputStream();
        httpCon.disconnect();
    }
    // Delete a resource via its id:
    public void delete(String idIn) throws Exception {
        URL url = new URL("http://localhost:8080/restmodell/" +
                "rest/Inputs/" + idIn);
        HttpURLConnection httpCon = (HttpURLConnection) url.
                openConnection();
        httpCon.setRequestMethod("DELETE");
        httpCon.connect();
        httpCon.getInputStream();
        httpCon.disconnect();
    }

    // Method to update the model:
    public void putModel() throws Exception {
        URL url = new URL("http://localhost:8080/restmodell/rest" +
                "/Outputs/1");
        HttpURLConnection httpCon = (HttpURLConnection) url.
                openConnection();
        httpCon.setDoInput(true);
        httpCon.setDoOutput(true);
        httpCon.setRequestMethod("PUT");
        httpCon.connect();
        httpCon.getInputStream();
        httpCon.disconnect();
    }

    // Method to update a specific resource:
    public void putObservation(String idIn, int alterIn, double DBHIn)
            throws Exception {
        URL url = new URL("http://localhost:8080/restmodell/rest/" +
                "Inputs/" + idIn);
        HttpURLConnection httpCon = (HttpURLConnection) url.
                openConnection();
        httpCon.setDoInput(true);
        httpCon.setDoOutput(true);
        httpCon.setRequestMethod("PUT");
        httpCon.connect();
        OutputStream outStream = httpCon.getOutputStream();
        OutputStreamWriter writer = new OutputStreamWriter(outStream);
        writer.write("&alter=" + alterIn + "&bhd=" + DBHIn);
        writer.flush();
        writer.close();
        outStream.close();
        httpCon.getInputStream();
        httpCon.disconnect();
    }
```

```
    // Method to convert InputStream to string:
    public String InputStreamToString(InputStream inStream)
            throws Exception {
        Writer writer = new StringWriter();
        char[] buffer = new char[1024];

        Reader reader = new BufferedReader(new InputStreamReader(
                inStream, "UTF-8"));
        int n;
        while ((n = reader.read(buffer)) != -1) {
            writer.write(buffer, 0, n);
        }
        inStream.close();
        return writer.toString();
    }
}
```

E Quellcode des WPS-Prozesses für das Modell Oberhöhe

Dieser WPS Prozess für das Oberhöhen-Modell entspricht dem Prozessaufbau, wie er in Kapitel 3.4.1 beschrieben ist. In der *execute*-Metohde wird das Oberhöhen-Modell ausgeführt und am Ende der Methode wird das Ergebnis als Prozess-Output gesetzt und damit in die Antwort auf die Execute-Operation einbezogen.

```
from pywps.Process.Process import WPSProcess
import math

class Process(WPSProcess):
     def __init__(self):
         # init process
         WPSProcess.__init__(self,
                             identifier="oberhoeheWPS",
                             title="Berechnung der Oberhohe",
                             version = "2.0",
                             storeSupported = "true",
                             statusSupported = "true",
                             abstract="Berechnung der oberhoehe"
                                    " aus Oberhoehenbonitaet und Alter",
                             grassLocation =False)
         self.alter=self.addLiteralInput(identifier = "alter",
                title="Das Alter", type = int)
         self.si=self.addLiteralInput(identifier= "si",
                title="Die Oberhoehenbonitaet", type = float)

         self.oberhoehe=self.addLiteralOutput(identifier = "oberhoehe",
                title = "Die Oberhoehe", type = float)

     def execute(self):
        alpha = -0.0006
        delta = 1.507
        A = float(self.alter.getValue())
        SI = self.alter.getValue()
        oberh = SI * math.pow(((1 - math.exp(alpha * SI * A)) /
        (1 - math.exp(alpha * 100 * A))), delta)

        self.oberhoehe.setValue(oberh)

        return
```

F Quellcode des WPS für das Baum-Höhenwachstums-Modell

Der WPS für das Baum-Höhenwachstums-Modell setzt sich zusammen aus dem Quellcode für den eigentlichen WPS-Prozess und den drei Modulen für das Einlesen der Daten, die Berechnung der Höhe und das Erstellen des GML-Dokumentes.

F-1 Der WPS-Prozess für das Baum-Höhenwachstums-Modell

Hier wird der komplette Quellcode des WPS-Prozesses für das Modell aus Kapitel 3.4.1 gezeigt, für das dort lediglich ein Ausschnitt dieses Programmcodes gezeigt wurde.

```
from pywps.Process import WPSProcess
from readin import readin
from createXML import createXML
from calculateHeight import CalculateHeight, modelfunk

class Process(WPSProcess):
    def __init__(self):
        # init process
        WPSProcess.__init__(self,
                identifier="WPSmodell", #the same as the file name
                title="Modellanbindung mit WPS",
                version = "2.0",
                storeSupported = "true",
                statusSupported = "true",
                abstract="Eingangsdaten sind x- und y-Koordinate und "
                    "das Baumalter. Daraus erstellt der WPS ein"
                    "GML-File, mit den Baum-Positionen und der"
                    "Baumhoehe, welche aus dem Alter berechnet"
                    "wird.",
                grassLocation =False)

        #input Data
        self.dataIn = self.addComplexInput(identifier="InputData",
                title="Csv-Datei. Komma als Trennzeichen. 3 Spalten. X"
                        "Koordinate, Y Koordinate, Alter. Eine Zeile pro"
                        " Beobachtung",
                formats = [{'mimeType':'text/plain'}])

        #output Data
        self.ausgabe = self.addComplexOutput(identifier="ergebnis-GML",
                title="Punkte-Layer. Eingezeichnet sind die Baeume mit"
                        "Alter und berechneter Hoehe",
                formats = [{'mimeType': 'text/xml', 'encoding':'utf-8',
                        'schema':'http://schemas.opengis.net/gml/3.2.1/gm'
                        'l.xsd'}])

    def execute(self):
        #1. read input-data into array
        array_x, array_y, array_age= readin(self.dataIn.getValue())

        #2. modell-application. Calculate height from age
        array_height= CalculateHeight(array_age)

        #3. create GML-File from input-data
        self.ausgabe.setValue(createXML(array_x, array_y, array_age,
            array_height, "GML-Document"))

        return
```

F-2 Funktion zur Verarbeitung der Eingangsdatei

Mit dieser Funktion wird die Eingangsdatei, die im CSV-Format vorliegt, eingelesen und die darin enthaltenen Daten werden zur Weiterverarbeitung in Arrays gespeichert.

```
def readin(dataname):
    readin = open (dataname, "r")
    lines = readin.readlines()
    readin.close()
    az = len(lines)

    #arrays for inputs
    array_x = []
    array_y = []
    array_age = []

    #readout lines and save to arrays
    for i in range (0, az, 1):
        lines[i] = (lines[i][:-1])
        split = lines[i].split(",")

        try:
            array_x.append(float(split[0]))
            array_y.append(float(split[1]))
            array_age.append(float(split[2]))
        except ValueError:
            pass

    #return of inputs in arrays
    return array_x, array_y, array_age
```

F-3 Funktion zur Anwendung der Modellgleichung

Innerhalb dieser Funktion werden die in den Arrays gespeicherten Eingangsdaten für das Baumalter für die Modellierung der Baumhöhen verwendet.

```
from math import exp

#function to create height-array from age-array
def CalculateHeight(array_age):

    length = len(array_age)
    array_height = []

    #for-loop to fill height-array
    for i in range (0, length, 1):
        array_height.append(modelfunk(array_age[i]))

    return array_height

#function to calculate height from age
def modelfunk(age):

    #set parameters
    v1 = 1.3567
    v2 = 2.596616
    v3 = 1.503162
    v4 = 19.141894

    #modell-function, devided into parts
    faktor1 = 65.0**v1
    faktor2 = v4/(65.0**v1)
    minuend = v2/((v3-1) * age**(v3-1))
```

```
    subtrahend = v2/((v3 - 1.0) * 50**(v3-1))
    height = faktor1 * faktor2**exp(minuend - subtrahend)

    return height
```

F-4 Funktion zur Erstellung des GML-Dokumentes

Mit dieser Funktion wird aus den Koordinaten der eingelesenen Beobachtungen und den modelierten Höhen eine GML-Datei erstellt.

```
import xml.dom.minidom

#function to create GML-File from Arrays
def createXML(array_x, array_y, array_age, array_height, name):

    length = len(array_x)

    doc = xml.dom.minidom.Document()

    #Create root-element
    root = doc.createElementNS("http://example.net/ns",
            "ogr:FeatureCollection")
    root.setAttributeNS("http://", "xmlns:gml",
           "http://www.opengis.net/gml")
    root.setAttributeNS("http://", "xmlns:ogr",
            "http://ogr.maptools.org/")
    doc.appendChild(root)

    #for-loop to create tags, namespaces and content for GML-File
    for i in range (0, length, 1):
        featureMember = doc.createElementNS('http://example.net/ns',
              'gml:featureMember')
        root.appendChild(featureMember)
        trees = doc.createElementNS('http://example.net/ns', 'trees')
        featureMember.appendChild(trees)

        age = doc.createElement("Age")
        age_content = doc.createTextNode(str(array_age[i]))
        age.appendChild(age_content)
        trees.appendChild(age)
        height = doc.createElement("Height")
        height_content = doc.createTextNode(str(array_height[i]))
        height.appendChild(height_content)
        trees.appendChild(height)
        point = doc.createElementNS('http://example.net/ns', 'gml:Point')
        trees.appendChild(point)
        coord = doc.createElement("coord")
        point.appendChild(coord)
        coord_x = doc.createElement("X")
        x_content=doc.createTextNode(str(array_x[i]))
        coord_x.appendChild(x_content)
        coord_y = doc.createElement("Y")
        y_content=doc.createTextNode(str(array_y[i]))
        coord_y.appendChild(y_content)
        coord.appendChild(coord_x)
        coord.appendChild(coord_y)

    #write GML-structure to a file
    data = open(name + ".gml", "w")
    doc.writexml(data, "", "\t", "\n")
    data.close()

    #return GML-file
    return data
```

Erklärung

Hiermit versichere ich gemäß § 9 Abs. 5 der Master-Prüfungsordnung vom 22.07.2005, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____

16.3.2012, Jonas Müller