Utilizing Open Travel Alliance-based ontology of golf in an agent-based travel support system

Agnieszka Cieślik[1], Maria Ganzha[2], and Marcin Paprzycki[2]

[1] Department of Mathematics and Information Technology,
Warsaw University of Technology, Warsaw, Poland
agnieszka.cieslik@gmail.com
[2] Systems Research Institute, Polish Academy of Science, Warsaw, Poland
{maria.ganzha,marcin.paprzycki}@ibspan.waw.pl

Abstract. Currently, we are developing an agent-based travel support system, in which ontologically demarcated data is used to facilitate personalized information provisioning. Recently we have shown how Open Travel Alliance golf-related messages can be reverse-engineered to create an *OTA ontology of golf*. The aim of this paper is to illustrate how these ontologies are going to be used in the system. In addition to the general scenario, details concerning implementation of needed translators will be discussed.

1 Introduction

Our current work is devoted to developing a comprehensive agent-based traveler support system, and involves a number of sub-projects. First we have been developing a model agent-based e-commerce system (see [2, 1], and references to our earlier work contained there). There we study utilization of agent-facilitated autonomous price negotiations in a general e-commerce scenario. This work was extended to facilitate possibility of airline ticket auctioning [25, 23, 24, 20, 26]. Second, we work on creation of an agent based *Travel Support System* (*TSS*) [18, 7, 10]. In the *TSS*, travelers are to obtain personalized information related to their travel needs (e.g. favorite hotels, restaurants, etc.). The main idea of the *TSS* is to utilize a central repository of semantically demarcated travel data, and operate on it to deliver personalized information [7, 8]. While these two projects (the airline ticket auctioning system and the Travel Support System) are being developed separately, in [25] we have discussed issues involved in their possible merger. Since the ontologically demarcated data is the central component of the *TSS*, the two projects were conceptually merged through development of a common travel ontology. Within the *TSS*, we have initially developed an ontology of hotels and restaurants [6, 9]. In the airline ticket auctioning system we have utilized the fact that the Open Travel Alliance (OTA) [15] has proposed a set of *messages* designed to facilitate meaningful communication about travel-related activities such as flights or golf course reservations. Here, it has to be stressed that, while on the way to become industry standard, these messages *do not* explicitly define an ontology. Therefore, in [26] we have proposed how OTA

air-travel-related messages can be used as a basis for development of an ontology of air-travel. We have proceeded with development of such ontology and later merged it with the existing ontology of restaurant and hotel (results—a complete ontology of restaurants, hotels and air travel—can be found within [22]). Most recently, following the example set forward in [26] we have shown how the *OTA golf messaging* can be turned into an ontology of golf [3]. The aim of this paper is to describe how systems utilizing the *OTA ontology of golf* can collaborate with entities that utilize *OTA golf messaging* (and, for instance, for one reason or another, do not work with ontologically demarcated data).

To this effect we proceed as follows. In the next section we briefly describe OTA golf messages as well as the proposed *OTA ontology of golf*. Next we provide context in which *OTA golf messages* and the *OTA ontology of golf* can interact. Finally, we discuss *how* these interactions can be implemented using currently existing technologies.

2 OTA golf messages and ontology

As all OTA messages concerning various "areas of travel," OTA golf-related messages come in pairs [17]. There is a request (RQ) message (a query) and, corresponding to it, a response (RS) message. As what concerns this paper, the OTA standard identifies three pairs of golf-related messages (detailed description can be found in [17] and [3], Table 1):

- *OTA_GolfCourseSearchRQ*—request for course information; used to find golf courses that satisfy a given set of criteria,
- *OTA_GolfCourseSearchRS*—list of courses that meet the requested criteria,
- *OTA_GolfCourseAvailRQ*—requests information about course availability,
- *OTA_GolfCourseAvailRS*—provides information about course availability,
- *OTA_GolfCourseResRQ*—requests a reservation of a given golf course,
- *OTA_GolfCourseResRS*—confirms (or denies) reservation of a given course.

These messages allow interested party: (1) to find a golf course with specific characteristics (claimed to include all features that any golfer could think off), (2) to check if a course of interest is available at a specific time and under a specified set of conditions (e.g. start time, or price), and (3) to make an actual reservation.

To illustrate the form of OTA messages, in Figure 1 we present an example on an *OTA_GolfCourseAvailRQ* message (see, [17]). In this message four friends specify that they would like to play golf on June 22nd, and the requested tee-off time is to be between 14:00 and 15:30. They are interested in playing at a specific golf course with the identifier *PL4321* (it is assumed that through an earlier query-message they have established that *PL4321* is the course that they are interested in). The maximum price that they are willing to pay for 18 holes is $75.00 per person. The aim of this message is to find if the *PL4321* course is available at a given time and if the price condition is satisfied.

```
<?xml version="1.0" encoding="UTF-8"?>
<OTA\_GolfCourseAvailRQ xmlns=
        "http://www.opentravel.org/OTA/2003/05"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
''http://www.opentravel.org/OTA/2003/05
                              OTA_GolfCourseAvailRQ.xsd''
EchoToken="12345"
                    TimeStamp="1003-05-31T13:20:00-05:00"
                    Target="Production" Version="1.001"
                    SequenceNmbr="123456">
<GolfCourseTeeTimes CourseID="PL4321">
<GolfCourseTeeTime Start="2003-10-31T14:00:00"
                    End="2003-10-31T15:30:00"
                    NumberOfGolfers="4"
                    NumberOfHoles="18"
                    NumberOfTimes="1"
                    MaxPrice="75.00"
                    CurrencyCode="USD">
</GolfCourseTeeTime>
</GolfCourseTeeTimes>
</OTA\_GolfCourseAvailRQ>
```

Fig. 1. Example of an OTA golf course availability query message

Analysis of OTA golf-related messages (see, [3] for a complete description) revealed that two core concepts have to be defined: (a) *Golf Course*—specifying static features of a golf course (e.g. Course ID, Address, Architect, etc.) and (b) *Golf Course Tee Time*—defining (dynamic) information necessary for a reservation of a golf course (e.g. Start date and time, Price, Number of golfers, etc.). After identifying these two core concepts, taking into account the fact that we are developing and extending an existing ontology of travel (the *TSS ontology*), we have analyzed which of its parts should be re-used. As a result, in [3] we have introduced the resulting *OTA golf ontology* and indicated how it is going to be integrated with the *TSS ontology*.

3 Utilizing OTA golf messages and OTA golf ontology

Let us now consider how the *OTA golf messages* and the *OTA golf ontology* can be utilized in the general context of "Internet-travel" and in the context of our *Travel Support System*. Here, we have to bring forward a slightly bigger picture of the "world of travel" that can provide a canvas for what we are doing. First, let us assume that the OTA messaging becomes an industry-wide standard for travel-related communication. The second assumption is that the idea of the Semantic Web takes off as expected by its proponents, and utilization of ontologically demarcated data starts to become a standard ([4]). Finally, let us also assume

that software agents start to play an important role in the computational fabric (see for instance arguments put forward in [13]). Note that we do not claim that these assumptions are uncontroversial; rather, we simply accept them as the departure point and focus on developing a system that is going to work if they are to be true. However, our work is also geared toward assessing feasibility of these assumptions by attempting at implementing a system that utilizes them as its foundation. From the three assumptions follows naturally a vision of a situation in which at least the following stakeholders participate in travel-related activities:

- *Users* represented by their *Personal Agents* (*PA*); here the notion of the *Personal Agent* follows the concept of "agent as a personal assistant" put forward in [14]; to support travel needs of *Users*, their *PA*s may contact either *Travel Service Providers* to obtain a specific information / reservation, or *Travel Agencies* which can provide, for instance, complete travel packages (e.g. airline ticket + car rental + hotel + golf). Obviously, in the first case content personalization will be facilitated by the *PA* alone. In the second case, it is possible that initial content personalization will take place within the *Travel Agency* (note that the *TSS*, can be viewed as such agency), which will utilize past interactions with a representative of a given *User* and, for instance, data clustering-based stereotypes, to select and rank offers out of existing possibilities.
- *Travel Service Providers* that provide information about, and facilitate reservations of, specific travel entities (e.g. hotel chains, individual hotels, restaurants, golf course operators), as well as global reservation systems (e.g. Amadeus). Their role is most likely going to be limited to content delivery. The only exception may be related to existing loyalty programs, which will allow some of such entities (e.g. Mariott Hotels) to acquire, store and utilize customer data.
- *Travel agencies*, which may play the same role as *Travel Service Providers*; here we assume that it is unlikely that "anyone" will have access to global reservation systems (e.g. for security reasons). They also provide integrated services (e.g. a vacation package to Milan, consisting of: airline reservation + hotel + opera tickets). Their profit may come, for instance, from selling extra services on the basis of knowledge of habits of their users (similarly to Amazon.com suggesting additional items based on similarities of behavior of their customers).

Let us now discuss how these stakeholders may want to store the necessary data and in this context consider the question: will there be a single ontology of travel. While the ideal situation is promoted by project CYC [16], where a single ontology of "everything" is to be developed, this vision is unlikely to materialize for a variety of reasons (e.g. multilinguality of the world, pragmatic needs of individual players etc.). Instead, we can expect that (a) some entities will move toward ontologies very slowly, e.g. old players such as global reservation systems, (b) some entities will utilize domain and business specific ontologies, e.g. hotel chains may use a combination of a "hotel as a tourist entity" ontology and "hotel

as a business entity" ontology, while have no use and knowledge of ontologies of other travel entities, (c) *Personal Agents* may use simplified ontologies, that are large enough to support their users, e.g. in such ontology concepts related to "hotel as a place for a conference" (including capacity of meeting rooms) may be omitted. Therefore, we can expect that different stakeholders of the "world of travel" will utilize different data representation (ontologically demarcated, or not). Furthermore, even if data will be stored in an ontologically demarcated fashion, different players will use different ontologies.

These considerations point back to one of main reasons of creation of the OTA messaging system. While each travel entity may use different data storage, they all should be able to communicate utilizing OTA messaging. Obviously this means that each time messages are to be exchanged, a number of translations needs to take place:

- In the case of *Travel Service Providers*, incoming OTA requests have to be translated into queries matching their internal data representation. Resulting responses have to be translated "back" into the OTA response messages and send to requesters.
- We should assume that communication between the *User* and its *Personal Agent* does not involve OTA messages. Rather, the *User* fills-in a form (e.g. an HTML template) and the resulting querystring is send to the *PA* (see, [5] for more details about non-agent entities communicating with software agents). The *Personal Agent* takes the *User*-query and translates it into an OTA request message, which can be send either to *Travel Service Providers*, or to *Travel Agencies*. Obtained OTA response has to be translated into the local ontology, as this is the data representation that is used by the *PA* to filter and order responses (later translated into user readable form and displayed on the user device; for more details see, [5]).
- The *Travel Agency* (e.g. the *TSS*) receives OTA requests from the *User*. Some of them can be answered directly by the *TSS*. For instance, since in the *TSS* we gather data, and keep it fresh by systematic updates, static elements such characteristics of the golf course (represented by the *Golf Course* concept) can be found by querying the internal database of teh *TSS*. Specifically, in the current design of the *TSS*, ontologically demarcated travel data is kept in the Jena repository [12]. Therefore, the OTA request message is translated into the SPARQL query [19] and executed. The result may then either be translated into an OTA response message and send to the *PA*, or further processed (e.g. to propose other travel related items that a given *User* may be interested in and in this way to maximize its profit [10]). The second possibility is that the original request requires access to *Travel Service Providers* (e.g. a request to check availability of a given golf course). Such message can be forwarded to an appropriate *Travel Service Provider* to obtain the necessary data (see above). The response is then treated as if it was obtained from the local database.

The scenario involving *Travel Service Providers* is uninteresting, as we cannot speculate what is their internal data representation. Furthermore, currently

the *Personal Agent* is an internal part of the *TSS* (see [10]). Therefore, to illustrate how to implement necessary translations we will focus on golf messages and the needs of the *Travel Support System*. However, at this stage we have not implemented golf-related functionalities directly within the *TSS*. Instead, for testing purposes, we have implemented it as a separate sub-system and introduced a number of auxiliary agents, out of which the most important one is the *Translation Agent* (*TA*). Actions undertaken by the *TA* depend on received messages and have been summarized in Table 3 (it should be obvious that the *TA*, or its functions could be used directly by (or within as a sub-agent of) the *Personal Agent* to fulfill its role in *User* support):

Table 1. TA actions depending on received messages

Message	**TA Actions**
message *TA_translate_from_OTAGolfCourseSearchRQ*	*TA* translates the *OTAGolfCourseSearchRQ* XML message to the structure *Conditions*
message *TA_translate_from_OTAGolfCourseSearchRS*	*TA* translates the *OTA_GolfCourseSeachRS* XML message to the list of instances of the *GolfCourse* ontology.
message *TA_translate_from_OTAGolfCourseAvailRS*	*TA* translates the *OTAGolfCourseAvailRS* XML message to the list of instances of the *GolfCourseTeeTime* ontology
message *TA_translate_to_OTAGolfCourseSearchRS*	*TA* translates the instances of the *GolfCourse* ontology to the *OTAGolfCourseSearchRS* XML message.
message *TA_translate_to_OTAGolfCourseAvailRQ*	*TA* translates the structure *Map* to the *OTAGolfCourseAvailRQ* XML message
message *Close_system_action*	*TA* finishes its activity

Here, the *Conditions* structure contains list of objects of the class *Condition* and has the form:

```
class Condition implements jade.content.Concept
{
        String name_;    /*name of the feature (e.g. ''Architect'')*/
        boolean required_;       /*is given criterion is required?*/
        String valueString;      /* value (e.g. ''Jan Kowalski'')*/
        String operation_;       /*operation*/
}
```

Class *Condition* is used to specify criteria of a requested golf course (criteria based on the *OTA_GolfCourseSearchRQ* message). This structure is used to generate the SPARQL query to be executed on the Jena repository.

The *Map* is a structure from the *TSS*. In the Golf sub-system it is used to specify details of the question regarding golf course availability. *Map* contains the list of objects of the class *MapEntry* and has the form:

```
class MapEntry implements jade.content.Concept
{
   private String key;    /*name of parameter (e.g. "golfCourseId")*/
   private String value; /*value of parameter(e.g. "AW313")*/
```

}

Classes *Conditions*, *Condition*, *Map* and *MapEntry* extend class *jade.content.Concept* and are part of the *GolfCourseOntology*.

3.1 Implementing message translations

Agent *TA*, during the above summarized translations of messages utilizes classes generated by the Castor and the Jastor software [21, 11]. Castor is an Open Source data binding framework for Java. Castor's Source Code Generator creates a set of Java classes which represents an object model for an *XMLSchema*. The input file for the source code generator is an *XSD* file. We used Castor to generate classes for all six OTA messages (*OTA_GolfCourseSearchRQ*, *OTA_GolfCourseSearchRS*, *OTA_GolfCourseAvailRQ*, *OTA_GolfCourseAvailRS*, *OTA_GolfCourseResRQ*, *OTA_GolfCourseResRS*). Castor generates classes, not only for messages but also for all types of their attributes. For instance let us consider a snippet of the the XMLSchema file for the *OTA_GolfCourseSearchRQ* message:

```
 elementFormDefault="qualified" version="1.005" id="OTA2006A">
 <xs:include schemaLocation="OTA_GolfCourseTypes.xsd"/>
 <!--xs:include schemaLocation="OTA_GolfCourseSearchRQTypes.xsd"/-->
 <!--xs:include schemaLocation="OTA_GolfCommonTypes.xsd"/>
 <xs:include schemaLocation="OTA_CommonTypes.xsd"/>
 <xs:include schemaLocation="OTA_AirCommonTypes.xsd"/>
 <xs:include schemaLocation="OTA_SimpleTypes.xsd"/-->

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.opentravel.org/OTA/2003/05"
  targetNamespace="http://www.opentravel.org/OTA/2003/05"
 <...appropriate headers come here...>
  <xs:annotation>
     <xs:documentation xml:lang="en"> </xs:documentation>
  </xs:annotation>
  <xs:element name="OTA_GolfCourseSearchRQ">
    <xs:annotation>
      <xs:documentation xml:lang="en"> </xs:documentation>
    </xs:annotation>
  <xs:complexType>
    <xs:sequence>
        <xs:element name="Criteria">
           <xs:complexType>
                <xs:sequence
                  <xs:element name="Criterion" maxOccurs="99">
                        <xs:complexType>
                        <xs:attributeGroup ref="CriteriaGroup"/>
                        </xs:complexType>
                  </xs:element>
                </xs:sequence>
        <...>
</xs:schema>
```

We can see that within the *OTA_GolfCourseSearch* message there is a list of *Criterion*, which is an attribute that has reference to the *CriteriaGroup*. Now, part of the *XMLSchema* file for the *CriteriaGroup* has the form:

```
        <xs:annotation>
                <xs:documentation xml:lang="en">
                </xs:documentation>
```

```
        </xs:annotation>
  <xs:attribute name="Name" type="StringLength1to32" use="required">
        <xs:annotation>
                <xs:documentation xml:lang="en">
                 </xs:documentation>
        </xs:annotation>

<xs:attributeGroup name="CriteriaGroup">
<...appropriate headers come here...>
        </xs:attribute>
  <xs:attribute name="Value" type="StringLength1to16" use="required">
          <xs:annotation>
           <xs:documentation xml:lang="en"></xs:documentation>
          </xs:annotation>
        </xs:attribute>
  <xs:attribute name="Required" type="xs:boolean" use="required">
                <xs:annotation>
                  <xs:documentation xml:lang="en"></xs:documentation>
                </xs:annotation>
        </xs:attribute>
<xs:attribute name="Operation" type="StringLength1to16" use="optional">
                <xs:annotation>
                  <xs:documentation xml:lang="en"></xs:documentation>
                </xs:annotation>
        </xs:attribute>
        </xs:attributeGroup>
```

Criterion has attributes: *Name* (type *String*; attribute *required*), *Value* (type *String*; attribute *required*), *Required* (type *Boolean*; attribute *required*), *Operation* (type *String*; attribute *optional*). Castor generates a class for the *Criterion* with methods *get* and *set*. Let us present a fragment of such resulting class:

```
public class Criterion implements java.io.Serializable {
  /** A code representing the criterion on which to filter */
        private java.lang.String _name;
  /** The value of the criterion */
        private java.lang.String _value;
  /** A flag establishing if this criterion
                must be met (value \textit{Yes}) */
        private boolean _required;
  /* keeps track of state for field: _required  */
        private boolean _has_required;
  /*Other operations to be used as the filter (e.g. GT, LT, etc.). */
        private java.lang.String _operation;
   //- Constructors -/
        public Criterion() {
                super();
        } //-- golfCourse.translations.castor.Criterion()
   //- Methods -/
  /*@return the value of field 'name'.  */
        public java.lang.String getName()
                {
                        return this._name;
                } //-- java.lang.String getName()
  /*@return the value of field 'operation'. */
        public java.lang.String getOperation()
                {
                        return this._operation;
                } //-- java.lang.String getOperation()
  /* @return the value of field 'required'.    */
        public boolean getRequired()
                {
                        return this._required;
                } //-- boolean getRequired()
  /*@return the value of field 'value'.    */
        public java.lang.String getValue()
                {
```

```
                return this._value;
            } //-- java.lang.String getValue()
  /*Method hasRequired */
        public boolean hasRequired()
            {
                return this._has_required;
            } //-- boolean hasRequired()
<...continued...>
```

In class generated for OTA_GolfCourseSearchRQ are methods to get and set list of Criteria:

```
public class OTA_GolfCourseSearchRQ implements java.io.Serializable {
...
   /** Field _criteria      */
    private golfCourse.translations.castor.Criteria _criteria;

...
   /** Returns the value of field 'criteria'.
     * @return the value of field 'criteria'.      */
    public golfCourse.translations.castor.Criteria getCriteria()
    {
        return this._criteria;
    } //-- golfCourse.translations.castor.Criteria getCriteria()
...
   /** Sets the value of field 'criteria'.
     * @param criteria the value of field 'criteria'.*/
    public void setCriteria(
                golfCourse.translations.castor.Criteria criteria)
    {
        this._criteria = criteria;
    } //-- void setCriteria(golfCourse.translations.castor.Criteria)
```

All requested classes generated by Castor have method *marshal* and static method *unmarshal*. These methods are used to convert Java classes to XML and to transform that XML back into the Java code. Method *marshal* converts an instance of a class to XML. Note using the method *marshal* we can transform only instances of a class, not the class itself. We instantiate (or obtain from a factory or from another instance-producing mechanism) that class to give it a specific form. Then, we populate fields of that instance with the actual data. Obviously that instance is unique; it bears the same structure as any other instances of the same class, but the data is separate. For instance, when we want to create the XML file from the *OTA_GolfCourseSearchRQ* message, we have two classes: *TA_GolfCourseSearchRQ* and *Criterion*. We must create instances of these classes and insert data into them. Here we will present only an example of utilization of the *marshall* method.

```
//create instance of OTA_GolfCourseSearchRQ class
OTA_GolfCourseSearchRQ ota = new OTA_GolfCourseSearchRQ();
// set data to this instance
...
// create instance of Criteria
Criteria criteria = new Criteria();
//put data from list of structure Condtion to Criteria
for(Iterator iter = conditions.getAllConditions(); iter.hasNext();)
{
        Condition condition = (Condition)iter.next();
//create instance of class Criterion
        Criterion criterion = new Criterion();
        criterion.setName(condition.getName_());
        criterion.setOperation(condition.getOperation_());
        criterion.setRequired(condition.getRequired_());
```

```
        criterion.setValue(condition.getValueString());
        criteria.addCriterion(criterion);
}
//put instance of Criteria to instance of class OTA_GolfCourseSearchRQ;
ota.setCriteria(criteria);
}
```

After that, we can convert these instances to XML:

```
/*put values to OTA (object of class OTA_GolfCourseSearchRQ)*/
...
Writer writer = new StringWriter();
try {   /* convert object to stream (XML text)*/
        ota.marshal(writer);
    }
 catch(MarshalException e) {...}
catch(ValidationException e) {...}
```

And we get XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<OTA_GolfCourseSearchRQ xmlns="http://www.opentravel.org/OTA/2003/05"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
   "http://www.opentravel.org/OTA/2003/05_OTA_GolfCourseSearchRQ.xsd"
        EchoToken="54321"
        TimeStamp="2003-11-12T10:30:00"
        Target="Production" Version="1.001"
        SequenceNmbr="2432"
        PrimaryLangID="en" ID="FL4902"
        DetailResponse="true">
<Criteria>
<Criterion Name="Architect" Value=''Robert Jones'' Required="false"/>
<Criterion Name="Slope" Value="110"
           Required="true" Operation="LessThan"/>
</Criteria>
</OTA_GolfCourseSearchRQ>
```

The second generator Jastor is used for work with ontologies. It generates classes for them (like Castor for the XMLSchema). Next, we can use Jastor to convert instances of these classes to instances of ontologies and transform back instances of ontologies to objects of these classes. Jastor generates Java interfaces, implementations, factories and listeners for ontologies. For instance, for the ontology *GolfCourse*, Jastor has generated four files:

- interface *GolfCourse* extends *com.ibm.adtech.jastor.Thing*
- interface *GolfCourseListener* extends *com.ibm.adtech.jastor.ThingListener*
- class *GolfCourseImpl* extends *com.ibm.adtech.jastor.ThingImpl*
- class *GolfCourseFactory* extends *com.ibm.adtech.jastor.ThingFactory*

We used Jastor to generate classes for all ontologies needed in the system: *GolfCourse*, *GolfCourseTeeTime*, *Contacts*, *Description*, *Price*, *Fee*, *AddressRecord*, and *OutdoorLocation*. For instance ontology *GolfCourseTeeTime* has parameters: *golfCourseId* (*String*), *amount* (*float*), *currencyCode* (*String*), *startDate* (*String*), *endDate* (*String*), *maxPrice* (*float*), *numberOfHoles* (*integer*), *numberOfTimes* (*integer*), *list of fees* (*Fee*). For this ontology Jastor generates the interface *GolfCourseTeeTime* with methods *get/set* for properties and class *GolfCourseTeeTimeImpl* that implements this interface. Let us see a snippet of this interface for the *golfCourseId*

```
public interface GolfCourseTeeTime extends com.ibm.adtech.jastor.Thing {
        ...
        /** Gets the 'golfCourseID' property value
         * @return              {@link java.lang.String}
         * @see                 #golfCourseIDProperty     */
        public java.lang.String getGolfCourseID()
                throws com.ibm.adtech.jastor.JastorException;

        /**Sets the 'golfCourseID' property value
         * @param               {@link java.lang.String}
         * @see                 #golfCourseIDProperty */
        public void setGolfCourseID(java.lang.String golfCourseID)
                throws com.ibm.adtech.jastor.JastorException;
<... continued for remaining parameters...>
```

Interfaces generated by Jastor for the ontology extend the interface *com.ibm.adtech.jastor.Thing*. Classes generated by Jastor extend the class *com.ibm.adtech.jastor.ThingImpl* that implements the interface *com.ibm.adtech.jastor.Thing*.

Work with Jastor is very similar to work with Castor. First Jastor generate classes for the ontologies (like Castor for XMLSchema) and then we work only with instances of these classes. We can convert instance of a class generated by Jastor to instance of an ontology (like instances of a class generated by Castor to XML). We can also transform back instances of an ontology to instances of a class generated by Jastor (like converting XML to instances of a class generated by Castor). During translation Agent *TA* uses classes generated by Castor and Jastor. So the *TA* has only to take values from the object of one class and put it to the object of another class.

4 Concluding remarks

In this paper we have discussed how OTA messages can be used to connect systems that utilize various forms of internal data representation (ontological or not). We have identified three groups of main stakeholders of the "world of travel," i.e. *Users*, *Service providers* and *Intermediaries* (e.g. *Travel Agencies*, or our own *Travel Support System*. Next, we have discussed scenarios that lead to communication between these three groups of players and specified what kind of translations between OTA messages and the *OTA ontology of golf* have to take place within our *TSS*. Finally we have discussed and illustrated on examples how Castor and Jastor software can be used to implement necessary translations. Our current work is devoted to merging the *OTA ontology of golf* and the translation mechanisms with the existing *TSS* and its *ontology of travel*.

References

1. C. Bădică, A. Bădită, M. Ganzha, and M. Paprzycki. *E-Service Intelligence—Methodologies, Technologies and Applications*, chapter Developing a Model Agent-based E-commerce System, pages 555–578. Springer, Berlin, 2007.
2. C. Bădică, M. Ganzha, and M. Paprzycki. *Journal of Universal Computer Science*, volume 13, chapter Implementing Rule-Based Automated Price Negotiation in an Agent System, pages 244–266. Springer, Berlin, 2007.

3. A. Cieślik, M. Ganzha, and M. Paprzycki. Developing open travel alliance-based ontology of golf. In *Proceedings of the 2008 WEBIST conference*.
4. D. Fensel. *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
5. M. Gawinecki, M. Gordon, P. Kaczmarek, and M. Paprzycki. The problem of agent-client communication on the internet. *Scalable Computing: Practice and Experience*, 6(1):111–123, 2005.
6. M. Gawinecki, M. Gordon, N. T. Nguyen, M. Paprzycki, and M. Szymczak. Rdf demarcated resources in an agent based travel support system. In G. M. et. al., editor, *Informatics and Effectiveness of Systems*, pages 303–310, Katowice, 2005. PTI Press.
7. M. Gawinecki, M. Gordon, N. T. Nguyen, M. Paprzycki, and Z. Vetulani. chapter Ontologically Demarcated Resources in an Agent Based Travel Support System, pages 219–240. Advanced Knowledge International, Adelaide, Australia, 2005.
8. M. Gawinecki, M. Kruszyk, and M. Paprzycki. Ontology-based stereotyping in a travel support system. In *Proceedings of the XXI Fall Meeting of Polish Information Processing Society*, pages 73–85. PTI Press, 2005.
9. M. Gordon, A. Kowalski, M. Paprzycki, T. Pełech, M. Szymczak, and T. Wasowicz. *Internet 2005*, chapter Ontologies in a Travel Support System, pages 285–300. Technical University of Wroclaw Press, 2005.
10. M. Gordon and M. Paprzycki. Designing agent based travel support system. In *ISPDC'2005:Proceedings of the ISPDC 2005 Conference*, pages 207–214, Los Alamitos, CA, 2005. IEEE Computer Society Press.
11. `http://jastor.sourceforge.net/`.
12. `http://jena.sourceforge.net/`.
13. N. R. Jennings. An agent-based approach for building complex software systems. *Commun. ACM*, 44(4):35–41, 2001.
14. P. Maes. Agents that reduce work and information overload. *Commun. ACM*, 37(7):30–40, 1994.
15. `http://www.opentravel.org`.
16. `http://www.cyc.com/`.
17. `OTA_MessageUserGuide2006V1.0`, 2006.
18. A. F. Salam and J. Stevens, editors. chapter Utilizing Semantic Web and Software Agents in a Travel Support System, pages 325–359. Idea Publishing Group, Hershey, USA.
19. `http://www.w3.org/TR/rdf-sparql-query/`.
20. M. Szymczak, M. Gawinecki, M. Vukmirovic, and M. Paprzycki. *Ontological Reusability in State-of-the-art Semantic Languages*, pages 129–142. Knowledge Management Systems. PTI Press.
21. `http://www.castor.org/`.
22. `http://www.e-travel.sourceforge`.
23. M. Vukmirovic, M. Ganzha, and M. Paprzycki. *Developing a Model Agent-based Airline Ticket Auctioning System*, pages 297–306. Springer, Berlin, 2006.
24. M. Vukmirovic, M. Paprzycki, and M. Szymczak. Designing ontology for the open travel alliance airline messaging specification. In M. B. et. al., editor, *Proceedings of the 2006 Information Society Multiconference*, 2006.
25. M. Vukmirovic, M. Szymczak, M. Ganzha, and M. Paprzycki. Utilizing ontologies in an agent-based airline ticket auctioning system. In V. L. et. al., editor, *Proceedings of the 28th ITI Conference*, pages 385–390, Piscatawy, NJ, 2006. IEEE.
26. M. Vukmirovic, M. Szymczak, M. Gawinecki, M. Ganzha, and M. Paprzycki. Designing new ways for selling airline tickets. *Informatica*, 31(3):93–104, 2007.