Developing and Utilizing Ontology of Golf Based on the Open Travel Alliance Golf Messages

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

[1] Department of Mathematics and Information Technology, Warsaw University of Technology, Warsaw, Poland
[2] Systems Research Institute Polish Academy of Sciences, Warsaw, Poland,
Maria.Ganzha@ibspan.waw.pl

Abstract. While the vision of the Semantic Web is an extremely appealing one, its success depends not only on development of new ontologies that represent various aspects of the world. Pragmatic view suggests that such activities have to go hand-in-hand with facilitating support for existing domain-specific real-world standards. One of such interesting standards that is systematically gaining popularity in the travel industry, is the *Open Travel Alliance* (*OTA*) messaging. Specifically, the *OTA* standardizes the way that businesses in the travel industry can communicate with each other. The aim of this chapter is to outline our efforts leading toward (re)engineering an ontology of golf (understood as a leisurely activity) anchored in the OTA golf messaging specification. Furthermore we discuss how such ontology could be used in a *Travel Support System*. Here, in addition to general scenarios, details concerning implementation of needed translators are presented.

Key words: ontology of golf, Open Travel Alliance, Travel Support System, ontological engineering, travel ontology utilization

1 Introduction

Let us start with a simple story. In late February of 2011, professor Hoffman from Stuttgart, Germany has to go to a conference in Florida. Professor Hoffman loves to play golf and would like to combine the conference with a few rounds of golf. Due to the rapid development of agent technology, prof. Hoffman on his laptop has the newest release of MPMG07 Personal Agent—his personal assistant. Furthermore, for some time already, he is using SRIPAS Travel Support System (*TSS*) for all his travel needs. To be able to arrange his trip, he informs his *Personal Agent* (*PA*) about dates of his trip and asks it to arrange all necessary details. The *PA* knows travel preferences of professor Hoffman and uses them to formulate a request to the *TSS*. The *TSS* communicates with various *Travel Service Providers* (e.g. airlines, hotels, golf courses) to make travel arrangements. Furthermore, since prof. Hoffman is one of its regular clients it can utilize this knowledge to (a) filter and sort potential offers, and (b) to make suggestions that go beyond the core of the query (e.g. to suggest a specific restaurant in Orlando that prof. Hoffman is likely to be interested in). In the latter

the *TSS* utilizes also knowledge mined from behaviors of its other clients. The proposal is returned to the *PA* of prof. Hoffman, that uses its own knowledge about his preferences to further filter and sort responses (it is the *PA* that may know that, due to the recently increased cholesterol level, prof. Hoffman stopped eating US Diner style food that he was fond off). Some of the decisions may be completed by the *PA*, some may involve prof. Hoffman himself. Some offers can be accepted immediately, while some may require further communication with the *TSS*. All communications between the *PA* and the *TSS* are utilized by the *TSS* to modify its profile of prof. Hoffman to serve him better in the future. As a result of these activities the itinerary is completed and prof. Hoffman can go to Orlando, present his work at the conference and play golf. During and after his trip he may communicate with his *PA* to provide explicit feedback about travel arrangements. This information is used by the *PA* to update prof. Hoffman's profile.

This futuristic story contains a number of issues that we try to address in our ongoing project, in which we are developing an agent-based Travel Support System. Its key features have been summarized in [23, 13, 16, 14] and in references to our earlier work collected there. In this chapter we will focus only on selected new developments, related to communication between various entities involved in making travel arrangements, as well as utilization of ontologies. Work presented here extends and complements material presented in [7, 8].

1.1 World of travel

The story of prof. Hoffman allows us to identify three key groups of key stakeholders of the world of travel.

- *Users*, which may be helped by and represented to the outside world by their *Personal Agents* (for more information about "agents as a personal assistants," see [20]). To check availability of a Marriott Hotel in St Louis, MO, the *PA* may interact with the *Travel Service Provider*, e.g. represented by a Marriott WWW site, or a Marriott Reservation Agent. Obviously, the *PA* may also contact a *Travel Support System* (which is a generalization of the notion of a *Travel Agency* and an example of a infomediary [17]). The *TSS* can provide, for instance, a complete travel package (e.g. airline ticket + car rental + hotel + golf). Obviously, in the first case content personalization will be facilitated solely by the *PA*. Here, we treat various loyalty programs as a part of itinerary preparation, and not as a form of content personalization. In the second case, as suggested above, it is likely that the initial content personalization will take place within the *TSS* (fulfilling its role of the *infomediary*) and the fine tuning (final filtering) will be done by the *PA*.
  Obviously, *Users* may also arrange their travel directly, by communicating via some form of web-interface with entities like the *Travel Support System*, or with *Travel Service Providers* to obtain specific information / reservation. For simplicity of description, from here on we focus our attention only on *Users* represented by their *PA*s.

- Two types of *Travel Service Providers* (*TSP*). The first groups represents, provides information about, and facilitates reservations of specific travel entities (e.g. hotel chains, individual hotels, restaurants, golf course operators). Second, *global reservation systems* (e.g. SABRE) that act as reservation aggregators. While these two groups differ to a certain extent (e.g. compare reservation system of a non-chain hotel San Max in Catania, Italy with the Amadeus global reservation system), still we can treat them all as end-point providers of reservations. Note also that while global reservation systems are somewhat similar to *Travel Support Systems*, we see the role of *Travel Service Providers* to be limited to activities involved in information provisioning and reservation processing. Finally, existence of loyalty programs, which allows some *TSP*s (e.g. Hilton Hotels) to acquire, store and utilize customer data, matters only as far as internal data processing is concerned (see below).
- *Travel Support Systems*, which in part play the same role as *Travel Service Providers*. Here, we assume that it is unlikely that "anyone" will be able to access global reservation systems directly (e.g. for security reasons). Therefore, *TSS*s will constitute an authorized entry point. They will also provide integrated, and to some extent personalized services (e.g. a vacation package to Milan, consisting of: airline reservation + hotel + opera tickets); see above. As in the case of typical infomediaries, they will not only respond to direct requests of customers, but attempt at selling extra services, selected on the basis of knowledge of habits of all of their customers (e.g. similarly to Amazon.com suggesting additional items based on similarities between customers).

Note that while it is obvious that the role of a *Travel Support System* is changing because of the Internet, still *Users* need to be able to have access to *TSP*s (e.g. to make reservations) while intermediaries (e.g. the *TSS*s) will attempt at earning money by providing value-added services. Therefore, regardless of specific form resulting form the evolution of all of the above identified stakeholders, the general picture presented here should remain valid for some time to come.

1.2 Travel-related data storage and processing

Let us now discuss how data is likely to be stored and represented by the three groups of stakeholders. While there exist arguments to the contrary, we accept the assumption that utilization of ontologically demarcated data is going to be a crucial part of future development of the Internet, and more generally, computational fabric of the world. Therefore, it is easy to see that the ideal situation would be realization of the vision put forward by the CYC project [21]. Here, a single ontology of "everything" is to be developed and accepted worldwide. If this vision would materialize, all problems related to data interoperability would be gradually solved (by all entities accepting such global ontology). Unfortunately, even casual observation of the way that the Semantic Web is developing suggests that this vision is unlikely to materialize for a very long time, if ever (e.g. due to multilinguality of the world, pragmatic/political needs of individual players,

etc.). Instead, we can expect that (a) some entities will move toward ontologies very slowly e.g. large existing players (such as global reservation systems), as well as very small players (such as individual non-chain hotels), (b) some players will utilize domain and business specific ontologies, e.g. hotel chains may use a combination of a "hotel as a tourist entity" ontology and a "hotel as a business entity" ontology, while have no direct use of other travel-related ontologies (e.g. they may want to be able to make a car reservation for their guests, but they will not store or process car rental related data and thus will not need ontology of car rental), (c) *Personal Agents* that reside on computers of their *Users* may utilize simplified ontologies, e.g. ontology of a hotel without concepts related to "hotel as a place for a conference" (including capacity of and equipment available in meeting rooms). Summarizing, we can expect that different entities within 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 are likely to use different ontologies. These ontologies may, but not have to be subsets of a larger, all agreed, comprehensive ontology of travel-related entities.

In our earlier work, we assumed that the *TSS* is going to store information in semantically demarcated form and use it to facilitate personalized information provisioning (see, [23, 13, 16, 14] and references collected there). Moving in this direction we have developed, and later merged, ontologies of a hotel and a restaurant. These two ontologies were created on the basis on the *concept of a hotel* as represented in travel-related WWW sites and *concept of a restaurant* as proposed in the ChefMoz project, respectively [12, 15]. Separately we have proceeded to develop a comprehensive ontology of air travel (see, [26, 25] for details), which was also merged with ontologies of hotel and restaurant. This ontology is available at [6].

Finally, it should be mentioned, that to represent user profiles and in this way facilitate personalized content delivery, we have adapted, to be used with ontologies, an overlay-based approach proposed originally in [19, 10]. In our work, each user is to have his/her preferences represented in a profile incorporated into the ontology of travel (see, [23, 13, 14, 12] for more details).

1.3 Communication in the world of travel

Thus far, we have argued that the world of travel has, and is likely to have for some time, at least three main groups of stakeholders. Furthermore, we have shown different players within the world of travel are likely to utilize different internal data representations. Thus, one has to ask a question: how will it be possible for them to communicate. One of the more promising answers has been proposed by the Open Travel Alliance (OTA) [2]. OTA was created in 2001 with the aim of developing a standard for communication between various entities represented the world of travel. They have designed message sets defining communication about practically all travel-related activities [4]. Interestingly, as time passes the OTA messaging standards is gaining popularity. For instance,

according to the OTA WWW site, its messaging has been adopted, among others, by American and Continental Airlines, Hilton and Marriott Hotels (for a complete list, see [3]). Specific OTA messages concern particular aspects of a travel-related activities and are defined as pairs: a request (RQ) message, and a response (RS) message. Depending on the field of interest, number of such message pairs varies, for instance, from three for a golf course related "conversation," to ten for the air travel.

Let us now assume that OTA messaging becomes a worldwide travel industry standard, which seems to be the case. Then the problem of communication between travel entities becomes solved. It should be clear, that while each of them may use different data representation, storage, and processing, they all will be able to communicate utilizing OTA messages. Obviously, this means that each time messages are to be exchanged, a number of translations needs to take place.

- Within *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 OTA responses and send to requesters.
- For the time being, we assume that travel-related communication between *Users* and their *Personal Agents* does not involve OTA messages. Rather, *Users* fill-in a form (e.g. an HTML template) and the resulting querystring is send to the *PA* (see, [11] for a discussion of how non-agent entities can communicate with software agents). Obviously, we can hope that one day we will be able to use natural language to communicate with the *PA*, but we omit this issue from considerations. The *Personal Agent* takes the *User*-query (expressed in any form) and translates it into an OTA request message, which can be send either to *Travel Service Providers*, and/or to *Travel Support Systems*. Received OTA responses have to be translated into instances of local ontology, as this is the data representation used by the *PA* to process information (e.g. to rank obtained proposals). Obviously, filtered and ordered responses have to be translated into user readable form and communicated to the user (for more details in the case of displaying information on the user device see, [11]).
- The *Travel Support System* receives OTA requests from *Personal Agents* representing *Users*. Some of them can be answered directly by the *TSS*. For instance, since the *TSS* gathers data, and keeps it fresh by systematic updates [16], static elements such as unchangeable characteristics of the golf course can be found by querying the local database of the *TSS*. Specifically, in the current design of the *TSS*, ontologically demarcated travel data is kept in the Jena repository [1]. Therefore, the OTA request message has to be translated into the SPARQL query [5] and executed. The result may then either be translated into an OTA response message and send to the *PA* "as is," 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 the profit of the *TSS* [16]). 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 *TSP* to obtain the necessary data (see above). The response is then treated as if it was obtained from the local database.

2 OTA golf messages

Let us now focus our attention on a specific case of travel-related communication–interactions concerning golf treated as a leisurely activity. Here, the OTA standard identifies three pairs of messages; summarized in Table 2 (see [22] for a complete description). These messages provide the following functionalities: (1) finding a golf course with specific characteristics, (2) checking if a course of interest (e.g. found utilizing the previous message) is available at a specific time and under a specific set of conditions (e.g. maximum price), and (3) making an actual reservation of a selected course.

To illustrate the specific form that OTA messages take, in Figure 1 we present an example on an *OTA_GolfCourseSearchRQ* message (based on [22]). In this message a person who is considered physically challenged under the ADA rules, and requires *Wheelchair Accessibility* (criterion specified as *true*). This person is seeking a course to be played alone (criterion *Singles* specified as *true*) and that has Robert Jones as its *Architect* (criterion is not required—specified as *false*).

```
<?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="Singles␣Confirmed" Value="Yes"
                                    Required="true"/>
        <Criterion Name="ADA␣Challenged" Value="Wheelchair"
                                    Required="true"/>
</Criteria>
</OTA_GolfCourseSearchRQ>
```

Fig. 1. Example of OTA golf course search request message

In response to the *OTA_GolfCourseSearchRQ* message depicted in Figure 1, the *OTA_GolfCourseSearchRS* message presented in Figure 2 could have been received. This message specifies that two golf courses satisfy the selected criteria. These courses have ID's *FL1234* and *FL4321*. Both of them satisfy the required criteria (*Wheelchair Accessibility* and *Singles Confirmed*, while only the first one has been designed by Robert Jones. However, since the *Architect* criteria was

Table 1. Summary of OTA golf messages

Message type	**List of fields**
OTA_ GolfCourseSearchRQ—message used to find golf courses that satisfy a given set of criteria; if attribute is specified as *Required* (set to *Yes*) then only courses that meet that criteria will be returned; if *Required* attribute is set to *No*, a course that does not meet that criteria may also be included in the list	Architect, ADAChallenged, Slope, Metal Spikes, Caddies available, Yardage, Personal Carts Permitted, Grass Type, Singles Confirmed
OTA_ GolfCourseSearchRS—response lists courses that meet the selected criteria	Golf Course ID, Golf Course address, Contact information—telephone number, List of requested criteria
OTA_ GolfCourseAvailRQ—requests information about availability of a specific golf course, satisfying a set of imposed conditions	Golf Course ID, Tee Time—start and end date, Number of golfers, Number of holes, Maximum price for one person
OTA_ GolfCourseAvailRS—response provides detailed information about availability	Golf Course ID, Tee Time, Number of golfers, Number of holes, Maximum price for one person, List of fees. Fee has name, information about amount, currency and taxes
OTA_ GolfCourseResRQ—message requests reservation of a given golf course	Information about person who makes reservation (first and last name, address, date of birth, telephone number), Mean of payment, Date of game, Number of golfers, Number of carts, List of fees
OTA_ GolfCourseResRS—confirms (or denies) reservation of a given golf course	Reservation ID, Information about person who makes reservation (first and last name, address, date of birth, telephone number), Mean of payment (credit cart information), Date of game, Number of golfers, Number of carts, List of fees, Information concerning cancellation penalties and date and time by which a cancellation must be made

not required, also the course designed by Jack Nicklaus can be correctly included in the response.

Assuming that one of these courses has been selected, it is likely that one would like to check its availability at a specific date and time, as well as satisfaction of various additional conditions (e.g. maximum price). This is achieved through the *GolfCourseAvailRQ* and *GolfCourseAvailRS* pair of messages. Finally, if the course is available and conditions are satisfied, a *GolfCourseResRQ*

```
<?xml version="1.0" encoding="UTF-8"?>
<OTA_GolfCourseSearchRS 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_GolfCourseSearchRS.xsd''
EchoToken="54321" TimeStamp="2003-11-12T10:30:15"
Target="Production" Version="1.002"
SequenceNmbr="2433" PrimaryLangID="en">
<Success/>
<GolfCourses>
  <GolfCourse ID="FL1234" Name=``Sea Grass Golf Resort''>
  <Address>
    <CityName>Jupiter</CityName>
    <PostalCode>21921</PostalCode>
    <County>Palm Beach</County>
    <StateProv StateCode="FL"/>
    <CountryName Code="US"/>
  </Address>
  <Phone AreaCityCode="444" PhoneNumber="423-8954"/>
  <Traits>
    <Trait Name="Architect" Value=``Robert Jones''/>
    <Trait Name=``Singles Confirmed'' Value="Yes"/>
    <Trait Name=``ADA Challenged'' Value="Wheelchair"/>
    <Trait Name="Slope" Value="110"/>
    <Trait Name="Metal_Spikes" Value="No"/>
    <Trait Name=``Caddies Available'' Value="No"/>
    <Trait Name="Yardage" Value="6345"/>
    <Trait Name=``Personal Carts Permitted'' Value="No"/>
    <Trait Name="Fivesome" Value="No"/>
    <Trait Name=``Grass Type'' Value="Bermuda"/>
  </Traits>
</GolfCourse>
<GolfCourse ID="FL4321" Name=``Beach Side Golf Resort''>
  <Address>
    <CityName>Palm Beach Gardens</CityName>
    <PostalCode>21932</PostalCode>
    <County>Palm Beach</County>
    <StateProv StateCode="FL"/>
    <CountryName Code="US"/>
  </Address>
  <Phone AreaCityCode="444" PhoneNumber="423-2876"/>
  <Traits>
    <Trait Name="Architect" Value=``Jack Nicklaus''/>
    <Trait Name=``Singles Confirmed'' Value="Yes"/>
    <Trait Name=``ADA Challenged'' Value="Wheelchair"/>
    <Trait Name="Slope" Value="112"/>
    <Trait Name=``Metal Spikes'' Value="Yes"/>
    <Trait Name=``Caddies Available'' Value="Yes"/>
    <Trait Name="Yardage" Value="7102"/>
    <Trait Name="Fivesome" Value="Yes"/>
    <Trait Name=``Grass Type'' Value="Rye"/>
  </Traits>
</GolfCourse>
</GolfCourses>
</OTA_GolfCourseSearchRS>
```

Fig. 2. Example of OTA golf course search response message

message could be send, requesting a reservation at a specific time. This message would then be followed by a *GolfCourseResRS* message that would confirm the reservation.

3 Designing the ontology—preliminary considerations

Now, we can discuss how *OTA golf messages* can be used as a basis for the development of an *OTA golf course ontology* (to be used, among others, within our *TSS*). Analysis of *OTA golf messages* indicated that two core concepts should be defined. The *Golf Course* concept identifies a golf course and specifies its features. This concept is based directly on the content of the first pair of OTA messages, where golf courses with specific features are sought. It defines an object (golf course) and its *static* features and is represented in Table 3.

Table 2. Golf Course concept and its features

Class GolfCourse	
Course ID	ID originates from the OTA_GolfCourseSearchRS message; can be used for getting information about golf course availability and for making reservations
Address	Address of golf course
Contact	Contact information (e.g. telephone number)
Features	List of golf course features

The second concept, named *Golf Course Tee Time*, defines information necessary for completing reservation of a golf course. Thus, the *Golf Course Tee Time* concept defines *dynamic* characteristics of a static object specified by the *Golf Course* concept. In Table 3 we list features that constitute the necessary information to define the *Golf Course Tee Time* concept. Since the "names of features" listed in the table are self-explanatory, we do not define them further.

Table 3. Golf Course Tee Time concept and its features

Class GolfCourseTeeTime
Course ID
Start date and time
End date and time
Price
Max price for one person
Number of holes
Number of golfers
Number of games
List if fees

After identifying two concepts that constitute the core of the *OTA golf ontology*, we have to address the following question: how does this ontology relate to the *TSS ontology*. In other words, we have to establish which already existing / defined concepts can be re-used in the new ontology.

3.1 Common concepts with the *TSS ontology*

Ontology re-use is one of important concepts in ontological engineering [9]. Therefore, we have compared the *OTA golf ontology* and the *TSS ontology* and analyzed which concepts can, and should, be re-used. Note that, as seen below, similarity of concepts can sometime be misleading as in actuality their represent different notions. Separately, one should keep in mind that the *OTA golf ontology*, should be made integrable with the *TSS ontology*. To help achieving this goal both ontologies should share as many concepts as possible. Thus, upon analysis of the *TSS* ontology we have identified the following existing concepts that could be re-used.

Outdoor Location—geographical location is associated with most objects populating the *TSS* ontology (i.e. restaurant, hotel, airport). Obviously, this concept is also associated with the golf course. The *OutdoorLocation* class from the *TSS ontology* describes geographical location through a set of geographical properties, such as: street address, country, city/town, region, zip code, reference points or location description (see the *TSS ontology* available at [6] for a complete listing). In the *TSS ontology*, the *Hotel*, the *Restaurant* and the *Airport* classes are sub-classes of the *OutdoorLocation* class. Therefore, the *GolfCourse* class proposed here should also become a subclass of the same *OutdoorLocation* class. This is a natural decision as the *Golf Course* should be an object of the same "nature" as the other objects mentioned here.

Discounts—is the concept that, in general, specifies:

- code of the particular discount,
- amount of reduction of the base-price,
- and contains a short description of the discount policy.

However, when dealing with air travel support we have realized that IATA defined special air travel discount codes [26, 25]. Therefore, the question has arisen: how to integrate these with hotel and restaurant discount codes (including both OTA-specific and general discounts—these omitted in the OTA specification). For the purpose of integration of ontologies, "domain-specific" discounts codes were distinguished and defined as subclasses of the general *DiscountTypes* class. Therefore, in the *TSS ontology* there exist three classes defining possible discounts:

- *OTADiscountTypes*—discount types originating from the OTA specification
- *IATADiscountTypes*—discount types originating from the IATA specification
- *DiscountTypes*—general class; all discount types

Obviously, classes *OTADiscountTypes* and *IATADiscountTypes* are subclasses of the *DiscountTypes* class. Note that since the proposed ontology of golf is based

on OTA messages, discount concepts used in the *OTA golf ontology* belong to the *OTADiscountTypes* class.

The remaining common parts between the *TSS ontology* and the *OTA golf ontology* are:

- *MeanOfPayment*—concept defining possible mean of payment (e.g cash, credit card, check, etc.),
- *AdressRecord*—class that in the *TSS ontology* describes the address,
- *Currency*—concept that defines what is the currency that the fees are in,
- *FareTax*—concept containing information about taxes,
- *Contacts*—class specifying possible ways of contacting an entity (e.g. the telephone number).

4 The *OTA golf ontology*

Based on the above considerations we can now present definitions of the two basic classes of the proposed *OTA golf ontology*. Its remaining features have been described in detail in [7]. First, in Figures 3 we present the class *OutdoorLocation* that the *Golf Course* concept is a subclass of. Next, in Figure 4, we present the RDF representation of the *Golf Course* class.

As discussed above, the *GolfCourse* class is a subclass of the *OutdoorLocation* class and utilizes the *Contacts* concept (from the *TSS ontology*). In its definition we use strings for: *id*, *courseName*, *architect*; and an integer for the *slope*.

The second concept that belongs to the core of the *OTA golf course ontology* is the *Golf Course Tee Time*. It is presented in Figure 5 (in the RDF notation) and in Figure 6 in the graphical representation. Finally, in Figure 7 we present the RDF description of the *Price* concept.

Observe that while the *GolfCourseTeeTime* class is relatively simple itself (it consists of strings for: *startDate*, *endDate* and *golfCourseID*; float for *maxPrice*; and an integer for *numberOfTimes*), it utilizes also a fairly extensive concept of a *Fee*. This points out to the fact that in addition to the two basic concepts (classes *GolfCourse* and *GolfCourseTeeTime*) we had to define the following additional concepts / classes:

- *Price*—concept of price (includes: amount, taxes, currency, etc.)
- *Fee*—concept of fee (e.g. green fee, cart fee)
- *Description*—contains all additional descriptions that are needed for the traveler to be able to effectively utilize the information provided by the system

Note that the concept of the *Price* is similar to that used in the *TSS ontology*, however in the case of a golf course it is much less complicated than in the case of air travel. Therefore we have decided, for the time being, to leave this concept golf-specific and return to this issue when the *OTA golf ontology* is going to be integrated with the *TSS ontology*.

```
base:OutdoorLocation a rdfs:Class;
        rdfs:subClassOf geo:SpatialThing;
        rdfs:comment ''Outdoor location.
                        Geographical and urban references.''.
base:address a rdf:Property;
        rdfs:comment ''Address details.'';
        rdfs:domain base:OutdoorLocation;
        rdfs:range adrec:AddressRecord.
base:attractionCategory a rdf:Property;
        rdfs:comment ''Nearby attractions.'';
        rdfs:domain base:OutdoorLocation;
        rdfs:range base:AttractionCategoryCode.
base:indexPoint a rdf:Property;
        rdfs:comment ''Reference map point.'';
        rdfs:domain base:OutdoorLocation;
        rdfs:range base:IndexPointCode.
base:indexPointDist a rdf:Property;
        rdfs:comment ''Distance from the reference map point.'';
        rdfs:domain base:OutdoorLocation;
        rdfs:range base:IndexPointCode.
base:locationCategory a rdf:Property;
        rdfs:comment ''Location category.'';
        rdfs:domain base:OutdoorLocation;
        rdfs:range base:LocationCategoryCode.
base:neighbourhood a rdf:Property;
        rdfs:label ''Neighbourhood'';
        rdfs:comment ''The neighborhood of the Outdoor location.'';
        rdfs:range xsd:string;
        rdfs:domain base:OutdoorLocation.
base:crossStreet a rdf:Property;
        rdfs:label ''Cross street'';
        rdfs:comment ''The nearest street that crosses the street that
                                        the travel object is on.'';
        rdfs:range xsd:string;
        rdfs:domain base:OutdoorLocation.
base:AttractionCategoryCode a rdfs:Class;
        rdfs:comment ''Possible categories of places which might be
                        of interest for visitors/guests and can be
                        found in the neighborhood.''.
base:IndexPointCode a rdfs:Class;
        rdfs:comment ''Possible reference map points.''.
base:LocationCategoryCode a rdfs:Class;
        rdfs:comment ''Possible location categories.''.
```

Fig. 3. OutdoorLocation concept; RDF representation

5 Utilizing OTA golf messages and OTA golf ontology

In section 1.3 we have summarized translations that need to take place when OTA messages are used to communicate between entities utilizing various forms of internal representation of travel data. In the remaining parts of this chapter we will concentrate our attention on translations involving *Travel Support System* that utilizes the above defined *OTA golf ontology*.

To facilitate the necessary translations, we have designed a *Translation Agent* (*TA*). Its actions are summarized in Table 4 (it should be obvious that the *TA*, or its functions could also be used directly by—or within; as a sub-agent of—the *Personal Agent* to fulfill its role in *User* support):

As it can be see in Table 4, in its work the *TA* utilizes two auxiliary structures—the *Conditions* and the *Map*:

```
base:GolfCourse   a rdfs:Class;
                rdfs:subClassOf   loc:OutdoorLocation;
rdfs:comment ''Used for city and geographical location description''.
base:id   a rdf:Property;
                rdfs:domain base:GolfCourse;
                rdfs:range xsd:string.
base:name  a rdf:Property;
                rdfs:domain base:GolfCourse;
                rdfs:range xsd:string.
base:contactInfo a rdf:Property;
                rdfs:comment ''Contact information.'';
                rdfs:domain base:GolfCourse;
                rdfs:range phc:Contacts.
base:architect a  rdf:Property;
rdfs:comment ''Golf course desiner'';
                rdfs:domain base:GolfCourse;
                rdfs:range xsd:string.
base:slope a    rdf:Property;
                rdfs:domain base:GolfCourse;
                rdfs:range xsd:integer.
base:availCaddy a rdf:Property;
                rdfs:domain base:GolfCourse;
                rdfs:range xsd:boolean.
base:permCart a rdf:Property;
rdfs:comment ''Information if personal carts are permitted''
                rdfs:domain base:GolfCourse;
                rdfs:range xsd:boolean.
base:yardage a  rdf:Property;
                rdfs:domain base:GolfCourse;
                rdfs:range xsd:float.
base:singlesConfirmed a rdf:Property;
                rdfs:domain base:GolfCourse;
                rdfs:range xsd:boolean.
base:metalSpikes a rdf:Property;
                rdfs:domain base:GolfCourse;
                rdfs:range xsd:boolean.
base:grass a rdf:Property;
                rdfs:domain base:GolfCourse;
                rdfs:range xsd:string;
```

Fig. 4. *Golf Course* concept; proposed *GolfCourse* class

- 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
{
```

```
base:GolfCourseTeeTime a rdfs:Class;

base:golfCourseID a rdf:Property;
        rdfs:domain base:GolfCourseTeeTime;
        rdfs:range xsd:string.
base:amount a rdf:Property;
        rdfs:domain base:GolfCourseTeeTime;
        rdfs:range xsd:float.
base:currencyCode a rdf:Property;
        rdfs:domain base:GolfCourseTeeTime;
        rdfs:range xsd:string.
base:startDate a rdf:Property;
        rdfs:comment ''Information about date and time in
                      format yyyy:MM:dd'T'HH:mm:ss'';
        rdfs:domain base:GolfCourseTeeTime;
        rdfs:range xsd:string.
base:endDate a rdf:Property;
        rdfs:comment ''Information about date and time in
                      format yyyy:MM:dd'T'HH:mm:ss'';
        rdfs:domain base:GolfCourseTeeTime;
        rdfs:range xsd:string.
base:maxPrice a rdf:Property;
        rdfs:domain base:GolfCourseTeeTime;
        rdfs:range xsd:float.
base:numberOfHoles a rdf:Property;
        rdfs:domain base:GolfCourseTeeTime;
        rdfs:range xsd:integer.
base:numberOfTimes a rdf:Property;
        rdfs:domain base:GolfCourseTeeTime;
        rdfs:range xsd:integer.
base:fee a rdf:Property;
        rdfs:domain base:GolfCourseTeeTime;
        rdfs:range fee:Fee.
```

Fig. 5. *Golf Course Tee Time* concept; proposed *GolfCourseTeeTime* class

```
private String key;    /*name of parameter (e.g. "golfCourseId")*/
private String value; /*value of parameter(e.g. "AW313")*/
```

Table 4. TA actions depending on received messages

Message	**TA Actions**
message *TA_translate_from_OTA-GolfCourseSearchRQ*	*TA* translates the *OTAGolfCourseSearchRQ* XML message to the structure *Conditions*
message *TA_translate_from_OTA-GolfCourseSearchRS*	*TA* translates the *OTA_GolfCourseSeachRS* XML message to the list of instances of the *GolfCourse* ontology.
message *TA_translate_from_OTA-GolfCourseAvailRS*	*TA* translates the *OTAGolfCourseAvailRS* XML message to the list of instances of the *GolfCourse-TeeTime* ontology
message *TA_translate_to_OTAGolf-CourseSearchRS*	*TA* translates the instances of the *GolfCourse* ontology to the *OTAGolfCourseSearchRS* XML message.
message *TA_translate_to_OTAGolf-CourseAvailRQ*	*TA* translates the structure *Map* to the *OTAGolf-CourseAvailRQ* XML message
message *Close_system_action*	*TA* finishes its activity

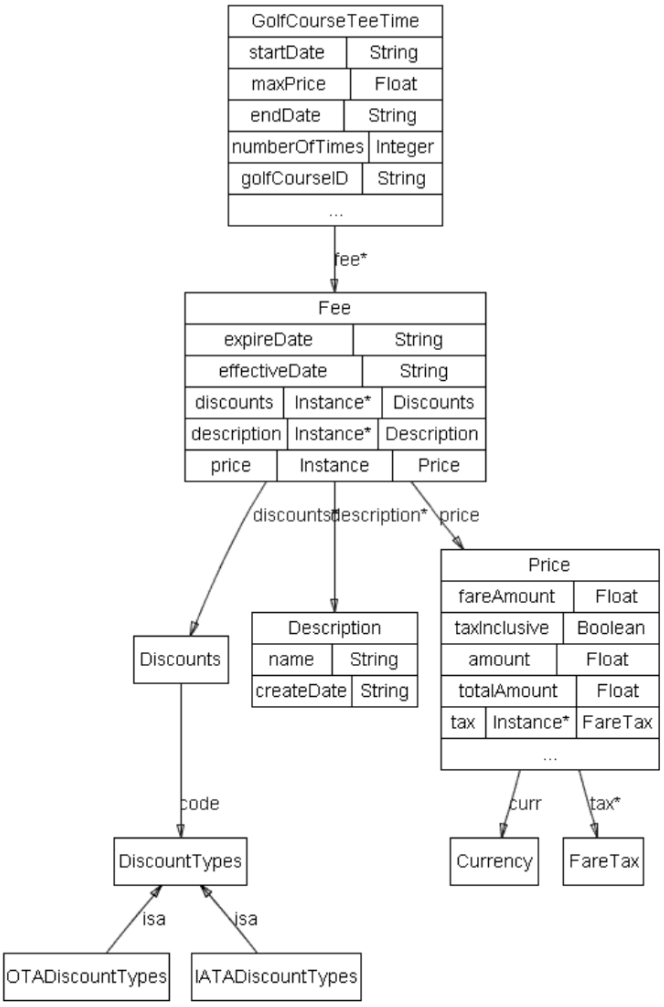

Fig. 6. *Golf Course Tee Time* concept; graphical representation

}

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

```
base:Price a rdfs:Class;
    rdfs:comment ''Description of a price''.
base:amount a rdf:Property;
    rdfs:domain base:Price;
    rdfs:range xsd:float.
base:taxInclusive a rdf:Property;
   rdfs:domain base:Price;
   rdfs:range xsd:boolean.
base:totalAmount a rdf:Property;
   rdfs:domain base:Price;
   rdfs:range xsd:double;
   rdfs:comment ''Total amount (taxes incl.)''
base:fareAmount a rdf:Property;
   rdfs:domain base:Price;
   rdfs:range xsd:double;
   rdfs:comment ''Fare amount (taxes excluded)''
base:tax a rdf:Property;
   rdfs:domain base:Price;
   rdfs:range tax:FareTax;
   rdfs:comment ''Information about taxes''
base:curr a rdf:Property;
   rdfs:domain base:Price;
   rdfs:range cur:Currency;
   rdfs:comment ''The currency information.''
```

Fig. 7. *Price* concept; proposed *Price* class

5.1 Implementing message translations

To be able to complete translations summarized in Table 4, the *TA* utilizes classes generated by the *Castor* [24] and the *Jastor* [18] software. Let us look into their utilization in some detail.

Utilization of Castor. Castor is an Open Source data binding framework for Java. Its *Source Code Generator* creates a set of Java classes which represents an object model for an *XMLSchema*, and its input file is an *XSD* file. We used Castor to generate classes for all six OTA messages (see Table 2). Furthermore, Castor generates classes, not only for messages but also for their attributes. For instance let us consider a snippet of the XMLSchema file for the *OTA_GolfCourseSearchRQ* message:

```
<?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>
```

Here, within the *OTA_GolfCourseSearchRQ* 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:attributeGroup name="CriteriaGroup">
<...appropriate headers come here...>
<xs:attribute name="Name" type="StringLength1to32" use="required">
</xs:attribute>

<xs:attribute name="Value" type="StringLength1to16" use="required">
</xs:attribute>

<xs:attribute name="Required" type="xs:boolean" use="required">
</xs:attribute>

<xs:attribute name="Operation" type="StringLength1to16" use="optional">
</xs:attribute>
</xs:attributeGroup>
```

Taking this as an input, Castor generates a class for the *Criterion* with methods *get* and *set*. The resulting class would have the following form (fragment):

```
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()
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()
}
```

In the class generated for the *OTA_GolfCourseSearchRQ* there are methods to *get* and *set* used to obtain and specify 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* used to convert Java classes to XML and to transform that XML back into Java code. Specifically, method *marshal* converts an instance of a class to XML. Note that by using the method *marshal* we can transform only instances of a class, not the class itself. In the process we instantiate (or obtain from a factory or from another instance-producing mechanism) that class to give it a specific form. Next, we populate fields of that instance with the actual data. Obviously that instance is unique; it has the same structure as other instances of the same class, but contains distinctive data. For example, 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 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);
}
```

Afterwards, 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 the following 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>
```

On the other hand, method *unmarshal* converts XML to an instance of a Java class. For example, let us assume that we want to extract information from the *OTA_GolfCourseSearchRQ* XML message. Then the method that does it has the following general form:

```
Reader reader = new StringReader(text);
try {
         OTA_GolfCourseSearchRQ ota = (OTA_GolfCourseSearchRQ)
                          OTA_GolfCourseSearchRQ.unmarshal(reader);
} catch(MarshalException e) {
...
} catch(ValidationException e) {
...
}
```

Here, the *unmarshal* method is invoked with the extracted information as its parameter. As a result, an instance of the *OTA_GolfCourseSearchRQ* class is created. Now, we can use the *get* method to get data from this class (from XML).

Utilization of Jastor The second generator Jastor is used to generate classes for ontologies (similarly to the way that Castor does 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 generated classes. Jastor generates Java interfaces, implementations, factories and listeners for ontologies. For instance, for the *GolfCourse* concept, 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, let us consider concept *GolfCourseTeeTime*, which has parameters: *golfCourseId* (*String*), *amount* (*float*), *currencyCode* (*String*), *startDate* (*String*), *endDate* (*String*), *maxPrice* (*float*), *numberOfHoles* (*integer*), *numberOfTimes* (*integer*), *list of fees* (*Fee*). For this concept, Jastor generates the interface *GolfCourseTeeTime* with methods *get/set* for properties, and the 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;
/**
* Gets the 'numberOfTimes' property value
* @return        {@link java.math.BigInteger}
* @see  #numberOfTimesProperty
*/
public java.math.BigInteger getNumberOfTimes()
        throws com.ibm.adtech.jastor.JastorException;

/**
* Sets the 'numberOfTimes' property value
* @param         {@link java.math.BigInteger}
* @see  #numberOfTimesProperty
*/
public void setNumberOfTimes(java.math.BigInteger numberOfTimes)
                throws com.ibm.adtech.jastor.JastorException;

/**
* Gets the 'currencyCode' property value
* @return        {@link java.lang.String}
* @see  #currencyCodeProperty
*/
public java.lang.String getCurrencyCode()
throws com.ibm.adtech.jastor.JastorException;

/**
* Sets the 'currencyCode' property value
* @param         {@link java.lang.String}
* @see  #currencyCodeProperty
*/
public void setCurrencyCode(java.lang.String currencyCode)
throws com.ibm.adtech.jastor.JastorException;

/**
* Gets the 'amount' property value
* @return        {@link java.lang.Float}
* @see  #amountProperty
*/
public java.lang.Float getAmount()
throws com.ibm.adtech.jastor.JastorException;
```

```
/**
* Sets the 'amount' property value
* @param          {@link java.lang.Float}
* @see  #amountProperty
*/
public void setAmount(java.lang.Float amount)
throws com.ibm.adtech.jastor.JastorException;

/**
* Gets the 'numberOfHoles' property value
* @return         {@link java.math.BigInteger}
* @see  #numberOfHolesProperty
*/
public java.math.BigInteger getNumberOfHoles()
throws com.ibm.adtech.jastor.JastorException;

/**
* Sets the 'numberOfHoles' property value
* @param          {@link java.math.BigInteger}
* @see  #numberOfHolesProperty
*/
public void setNumberOfHoles(java.math.BigInteger numberOfHoles)
throws com.ibm.adtech.jastor.JastorException;

/**
* Gets the 'numberOfGolfers' property value
* @return         {@link java.math.BigInteger}
* @see  #numberOfGolfersProperty
*/
public java.math.BigInteger getNumberOfGolfers()
throws com.ibm.adtech.jastor.JastorException;

/**
* Sets the 'numberOfGolfers' property value
* @param          {@link java.math.BigInteger}
* @see  #numberOfGolfersProperty
*/
public void setNumberOfGolfers(java.math.BigInteger numberOfGolfers)
throws com.ibm.adtech.jastor.JastorException;

/**
* Gets the 'maxPrice' property value
* @return         {@link java.lang.Float}
* @see  #maxPriceProperty
*/
public java.lang.Float getMaxPrice()
throws com.ibm.adtech.jastor.JastorException;

/**
* Sets the 'maxPrice' property value
* @param          {@link java.lang.Float}
* @see  #maxPriceProperty
*/
public void setMaxPrice(java.lang.Float maxPrice)
throws com.ibm.adtech.jastor.JastorException;

/**
* Gets the 'startDate' property value
* @return         {@link java.lang.String}
* @see  #startDateProperty
*/
public java.lang.String getStartDate()
throws com.ibm.adtech.jastor.JastorException;

/**
* Sets the 'startDate' property value
* @param          {@link java.lang.String}
* @see  #startDateProperty
```

```
*/
public void setStartDate(java.lang.String startDate)
throws com.ibm.adtech.jastor.JastorException;

/**
* Gets the 'endDate' property value
* @return        {@link java.lang.String}
* @see  #endDateProperty
*/
public java.lang.String getEndDate()
throws com.ibm.adtech.jastor.JastorException;

/**
* Sets the 'endDate' property value
* @param         {@link java.lang.String}
* @see  #endDateProperty
*/
public void setEndDate(java.lang.String endDate)
throws com.ibm.adtech.jastor.JastorException;

/**
* Get an Iterator the 'fee' property values.  This Iteartor
* may be used to remove all such values.
* @return        {@link java.util.Iterator} of {@link
* com.ibm.adtech.jastor.Thing}
* @see  #feeProperty
*/
public java.util.Iterator getFee()
throws com.ibm.adtech.jastor.JastorException;

/**
* Adds a value for the 'fee' property
* @param         The {@link com.ibm.adtech.jastor.Thing} to add
* @see  #feeProperty
*/
void addFee(com.ibm.adtech.jastor.Thing fee)
throws com.ibm.adtech.jastor.JastorException;

/**
* Adds an anonymous value for the 'fee' property
* @return        The anoymous {@link com.ibm.adtech.jastor.Thing} created
* @see  #feeProperty
*/
public com.ibm.adtech.jastor.Thing addFee()
throws com.ibm.adtech.jastor.JastorException;

/**
* Adds a value for the 'fee' property.  This
* method is equivalent constructing a new instance of
* {@link com.ibm.adtech.jastor.Thing} with the factory
* and calling addFee(com.ibm.adtech.jastor.Thing fee)
* The resource argument have rdf:type
*http://www.w3.org/2000/01/rdf-schema#Resource.
*That is, this method
* should not be used as a shortcut for creating new objects in the model.
* @param         The {@link om.hp.hpl.jena.rdf.model.Resource} to add
* @see  #feeProperty
*/
public com.ibm.adtech.jastor.Thing addFee(
                        com.hp.hpl.jena.rdf.model.Resource resource)
throws com.ibm.adtech.jastor.JastorException;

/**
* Removes a value for the 'fee' property.  This method should not
* be invoked while iterator through values.
*In that case, the remove() method of the Iterator
* itself should be used.
* @param         The {@link com.ibm.adtech.jastor.Thing} to remove
```

```
* @see #feeProperty
*/
public void removeFee(com.ibm.adtech.jastor.Thing fee)
throws com.ibm.adtech.jastor.JastorException;
...
}
```

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 generates classes for the ontologies (like Castor does for XMLSchema). Next, we work with instances of these classes. We can convert an instance of a class generated by Jastor to an 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 the *TA* uses classes generated by Castor and Jastor. Thus the *TA* has only to take values from the object of one class and put it to the object of another class.

6 Concluding remarks

The aim of this chapter was three-fold. First, we have outlined our vision of the future of the world of travel. We have argued, that it will consist of three main groups of stakeholders, that will utilize their own ways of storing and processing data. Therefore, for further development of this area, efforts like the OTA messaging standardization are o particular value. We have used this as a backdrop against which we have shown how we have reverse engineered an *OTA golf ontology* out of *OTA golf messages*. Finally, we have presented an in-depth description of translations that have to take place if a system is to to utilize just proposed *OTA golf ontology* and at the same time utilize *OTA golf messages* to communicate with other travel-related entities. Not only the general approach was discussed, but also implementation details have been presented. We believe that approach like the one presented here is needed also for all remaining OTA-defined standards and we plan to proceed in this direction.

References

1. Jena—RDF persistency engine. `http://jena.sourceforge.net/`.
2. Open travel alliance. `http://www.opentravel.org/`.
3. Ota registration program. `http://www.opentravel.org/MembersOnly/RegistrationProgram.aspx`.
4. Ota specifications. `http://www.opentravel.org/Specifications/Default.aspx`.
5. Sparql—RDF query language. `http://www.w3.org/TR/rdf-sparql-query/`.
6. Travel support system, software repositories. `http://www.e-travel.sourceforge`.

7. A. Cieślik, M. Ganzha, and M. Paprzycki. Developing open travel alliance-based ontology of golf. In *Proceedings of the 2008 WEBIST conference*, 2008. to appear.
8. A. Cieslik, M. Ganzha, and M. Paprzycki. Utilizing open travel alliance-based ontology of golf in an agent-based travel support system. In L. R. et. al., editor, *Artificial Intelligence and Soft Computing—ICAISC 2008*, LNAI, pages 1173–1184, Berlin, 2008. Springer.
9. D. Fensel. *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
10. J. Fink and A. Kobsa. User modeling for personalized city tours. *Artificial Intelligence Review*, (18):33–74, 2002.
11. 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.
12. M. Gawinecki, M. Gordon, N. T. Nguyen, M. Paprzycki, and M. Szymczak. Rdf demarcated resources in an agent based travel support system. In M. G. et. al., editor, *Informatics and Effectiveness of Systems*, pages 303–310, Katowice, 2005. PTI Press.
13. 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.
14. M. Gawinecki, M. Kruszyk, and M. Paprzycki. Ontology-based stereotyping in a travel support system. In *Proc. of the XXI Fall Meeting of Polish Information Processing Society*, pages 73–85. PTI Press, 2005.
15. 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.
16. M. Gordon and M. Paprzycki. Designing agent based travel support system. In *ISPDC'2005:Proc. of the ISPDC 2005 Conference*, pages 207–214, Los Alamitos, CA, 2005. IEEE Computer Society Press.
17. J. Hagel III and J. F. Rayport. The coming battle for customer information. Technical report, 1997.
18. `http://jastor.sourceforge.net/`.
19. A. Kobsa, J. Koenemann, and W. Pohl. Personalized hypermedia presentation techniques for improving online customer relationships. *The Knowledge Engineering Review*, (16:2):111–155, 2001.
20. P. Maes. Agents that reduce work and information overload. *Commun. ACM*, 37(7):30–40, 1994.
21. `http://www.cyc.com/`.
22. `OTA_MessageUserGuide2006V1.0`, 2006.
23. 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, 2006.
24. `http://www.castor.org/`.
25. 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*, pages 101–105. Josef Stefan Institute Press, 2006.
26. 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, Piscatway, NJ, 2006. IEEE.