# TRANSFORMATION OF THEMATIC CARTOGRAPHY DOMAIN ONTOLOGY INTO JAVA INTERFACES AND CLASESS

# TRANSFORMACE DOMÉNOVÉ ONTOLOGIE TEMATICKÉ KARTOGRAFIE NA JAVA ROZHRANÍ A TŘÍDY

*Tomáš PEŇÁZ* [1], *Radek DOSTÁL* [2]

[1] *Ing., Ph.D., Department of Geoinformatics, Faculty of Mining and Geology, VSB-Technical University of Ostrava,17. listopadu 15/2172, Ostrava Poruba, tel. (+420) 597 325 458 e-mail tomas.penaz@vsb.cz*

[2] *Ing., Department of Geoinformatics, Faculty of Mining and Geology, VSB-Technical University of Ostrava, 17. listopadu 15/2172, Ostrava Poruba e-mail radek.dostal@gmail.com*

**Abstract**

The article deals with the transformation of an experimental ontology, classifying selected declarative knowledge for the domain of thematic cartography, into classes and interfaces of the Java language. The reason for this transformation is to transfer the declarative knowledge from the field of thematic cartography into the form of a program code in the Java programming language. The resulting program code containing declarations of interfaces and classes will be further used for creating a software application for an intelligent system for the interactive support of thematic map creation. The upcoming pilot project of this knowledge system is designed for the users without necessary cartographic knowledge, which will allow them to create interactively thematic maps and provide them with the support. The purpose of the use of such a tool is to prevent the users from deviating from established cartographic rules and avoid the occurrence of gross errors in resulting maps.

A properly compiled knowledge ontology facilitates the design of the prepared intelligent cartographic application, as the use of cartographic knowledge is enabled based on the automated transformation into the program code in the Java language. The generated program code contains declarations of basic concepts of thematic cartography, their structuring into classes corresponding to the source structures described in the ontology. The code also contains descriptions vertical and horizontal relations between the declared classes and also the interface for access to these classes and relations. The automated transformation of ontology into the Java code is not completely lossless. The examples of a transformation of individual components of ontology (classes, individuals, and object and datatype properties) into interfaces and classes in Java show the differences occurred during the transfer of the declarative knowledge into the program code. From these examples it is clear that the elements of ontology component description are or are not transformed into the Java code in full. The article proposes recommendations how to optimize the knowledge description on the part of ontology in order to minimize possible losses of the transformation of knowledge into the Java code.

**Abstrakt**

Článek pojednává o transformaci experimentální znalostní ontologie, soustřeďující vybrané deklarativní kartografické znalosti pro doménu tematické kartografie, na třídy a rozhraní jazyka Java. Důvodem transformace je přenos deklarativních znalostí z oblasti tematické kartografie, do podoby programového kódu v jazyce Java. Vzniklý programový kód, obsahující deklarace rozhraní a tříd, bude dále využit pro tvorbu programové aplikace inteligentního systému pro interaktivní podporu tvorby tematických map. Připravovaný pilotní projekt tohoto znalostního systému je určen pro uživatele bez potřebných kartografických znalostí, kterým umožní interaktivně vytvářet tematické mapy a poskytne jim při tom podporu. Smyslem využití takového nástroje je zamezit uživateli odchýlení od zavedených kartografických pravidel a zabránit vzniku hrubých chyb ve výsledné mapě.

Vhodně sestavená znalostní ontologie usnadňuje návrh připravované inteligentní kartografické aplikace, neboť využití kartografických znalostí je umožněno na základě automatizované transformace do podoby programového kódu v jazyce Java. Vygerovaný programový kód obsahuje deklaraci základních pojmů tematické kartografie, jejich strukturování do tříd, které odpovídají zdrojovým strukturám popisovaným v ontologii. Kód obsahuje rovněž popis vertikálních a horizontálních vztahů mezi deklarovanými třídami a dále též rozhraní pro přístup k těmto třídám a vztahům. Automatizovaná transformace ontologie na Java kód však není zcela bezztrátová. Na příkladech transformace jednotlivých komponent znalostní ontologie (tříd, jedinců, objektových a datotypových vlastností) na rozhraní a třídy v jazyce Java, jsou ukázány odlišnosti při přenosu deklarativních znalostí do programového kódu. Z uvedených příkladů je patrné, které prvky popisu komponent ontologie jsou či nejsou transformovány do Java kódu plnohodnotně. V článku jsou navržena doporučení optimalizace popisu znalostí na straně ontologie s cílem minimalizovat případné ztráty při transformaci znalostí do kódu v jazyce Java.

**Key words:** declarative knowledge, intelligent system, Java, ontology, OWL, taxonomy, thematic cartography, thematic map.

## 1 INTRODUCTION

One of the possibilities of using artificial intelligence in cartography is the development of intelligent systems to support users in creating interactive maps. A simple but effective way of integrating intelligence into the environment of tools for interactive creation of maps is the implementation of optimization algorithms. An example is the algorithm implementing the Jenks' optimization method (Dent, 2009; Slocum, 2009), known from the environment of ArcGIS Desktop, Quantum GIS, and other products. The method allows the users the automated design of classes during classifying a statistical ensemble according to selected quantitative characters. From a series of classification methods disposed by the products, the Jenks' optimization method is offered to users as a default classification method in ArcGIS.

A different task of intelligent system may be the mediating of access to a cartographic knowledge base to the users without corresponding cartographic education (Brus, 2009). An intelligent system, complementing the software tool for creating maps, can influence the user creating a map based on the ongoing interaction in different ways. The simplest method of communication is to provide users with simple tips and simultaneous input of related requirements through a wizard. This article is based on the experience gained in solving the research project **Intelligent System for Interactive Support of Thematic Map Design** (hereinafter referred to as the project).

The design and preparation of an intelligent system cooperating interactively with the user needs to work with the technical vocabulary used in (thematic) cartography. A set of used terms must be sufficiently rich, apposite and sophisticated, in order to cover the corresponding part of issues included in the functionality of the intelligent system. Knowledge ontologies that are a subject of ontology modelling may be used as a knowledge database for implementing intelligent systems in various fields of human activity (domains). The use of ontologies facilitates the design and implementation of knowledge systems (Uschold, 1996). Knowledge ontologies are used as an environment suitable for the static description of a concept which includes declarative knowledge (Apt, 1988), related to the concepts used in the domain for the identification and description of objects and phenomena. The declarative knowledge includes a list of definitions of concepts, the classification of the concepts into groups, called *classes*, based on their similarity according to common characteristics, and the arrangement of the concepts and classes into a class hierarchy. Finally, it is important to express relations between the classes based on object properties, containing certain concepts, and the expressions of disjoint of sibling classes included in a superclass. The declarative knowledge includes also the expression of equivalence between classes and properties and the description of property restrictions.

During the analysis of the project issue, we did not manage to find any suitable domain ontology of thematic cartography, which could be used for the practical implementation of the intelligent system. Based on the search activity, we could not get relevant information how to approach to the design and preparation of the ontology of thematic cartography. The exception is some published information (Iosifescu-Enescu, 2005; Smith, 2010; Dobešová, 2011), which, however, has only a character of a general overview of the issue, or retrieval, and thus cannot be a sufficient basis for developing this concept. Due to the fact that the activities aimed at full formalization of domain knowledge of thematic cartography have not progressed so far, we have designed and built an own experimental ontology (Peňáz, 2010) for this field. In designing the knowledge ontology, we followed general recommendations (Gruber, 1993; Noy, 2001; Svátek, 2009) to obtain the best and perfect description of the issue of the thematic cartography domain. For processing the ontology we used the OWL-DL language (Patel-Schneider, 2004; Smith, 2004; Bechhofer, 2004). Although we had originally intended to use the new version of the language OWL 2 and the corresponding ontology editor Protégé 4.1, we omitted this goal and created the ontology interactively in the editor Protégé 3.4.4 (Horridge, 2004). The reason was the fact that there are no appropriate tools in the ontology editor Protégé 4.1 for the subsequent transformation of OWL into Java.

In order the declarative knowledge, concentrated in the knowledge ontology, to be available to programmers for further development, we had to convert the experimental ontology into a program code in Java. The obtained program code should include the declarations of classes and interfaces required for their use in the further development of the software application. However, the conversion of ontology components into the Java code is not completely lossless. The following chapters provide the findings we gained during this transformation. At the same time, they offer the description of the procedure of transforming the contents of the knowledge base into Java interfaces and clasess.

It is also to be noted at the beginning of the article that in publications (Studer, 1998; Kalyanpur, 2004; Gobin, 2010; Frenzel, 2011) also the phrase *ontology mapping* is used in connection with the described transformation of ontology into the Java code. The original meaning of the word *mapping*, however, comes from cartography, and therefore in connection with the contents of the article the term *transformation*, which is used mainly in this context, is more appropriate.

The article's topic is on the border of several disciplines, including in particular knowledge engineering and cartography. The article is primarily addressed to the professional community of specialists in geoinformatics and GIS application determined for digital cartography.

## 2 METHOD AND TOOLS FOR TRANSFORMATION OF ONTOLOGY COMPONENTS

The aim of the described transformation is to create a program code in Java, which includes collections of Java interfaces and classes based on the prepared experimental ontology of (thematic) cartography. The interfaces and classes should be structured hierarchically, but need not necessarily exactly correspond with OWL under all conditions (Puleston, 2008). An instance of a Java class should represent as accurately as possible an instance of a single ontology class with most of its properties (Kalyanpur, 2004). The transformation should therefore ensure the transfer of an important part of declarative knowledge from the ontology (knowledge base) to the program code of a knowledge application (Fig. 1). In (Kalyanpur, 2004), a notice of the existence of major semantic differences between the description of knowledge formalized by means of description logics (Baader, 2003, 2007) used in ontology and Object Oriented Systems can be found. The Object Oriented Systems include Java. The experience with transforming OWL into a Java code is also stated by (Stevenson, 2011; GOBIN, 2010). The OWL specification is available on the website of the W3C consortium (Patel-Schneider, 2004; Smith, 2004).

Generating Java classes and interfaces is an automated or partially automated operation and can be implemented using the standard function *Generate* of the Protégé-OWL Java Code, which is available in the editor Protégé 3.4.4.

The OWL editor Protégé (Knublauch, 2004) is a software tool of the open-source category, developed at the Stanford Center for Biomedical Informatics Research. It is the environment for development of knowledge-based systems that has been developed for almost twenty years (Gennari, 2003). The Protégé model is based on a simple but flexible metamodel (Noy, 2000), which is comparable with object-oriented and frame-based models. In fact, it may represent ontologies consisting of classes, properties, characteristics of properties (aspects and limitations) and individuals (instances). The Protégé editor provides an open Java API to query and handle models. Protégé also offers a highly scalable interface allowing users to create hundreds of thousands of ontology classes (Knublauch, 2004).

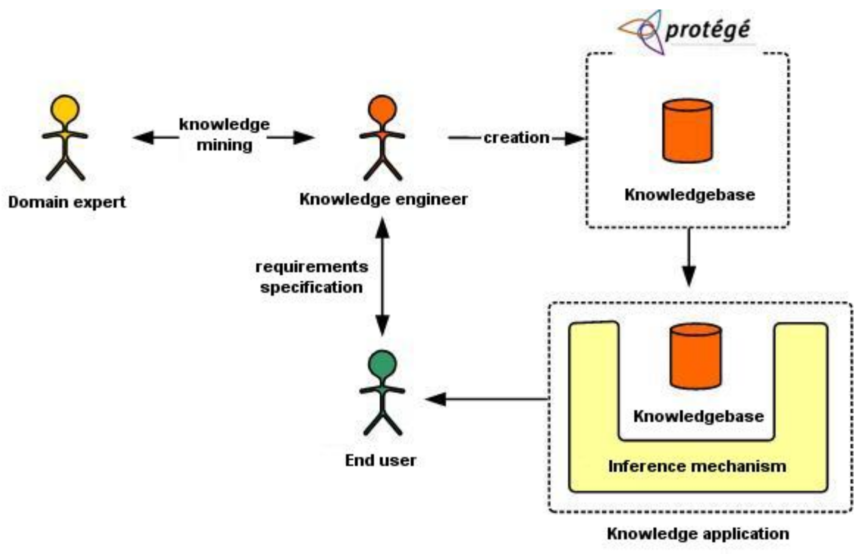


**Fig. 1**Protégé and the knowledge base (adjusted according to Husáková, 2006)

An important feature of the transformation is the ability of the function *Generate* of the Protégé-OWL Java Code to ensure the correct syntax of the resulting Java code. If the creator of ontology, as a result of ignorance of the Java language syntax, uses a hyphen in the name of an OWL component (which Java does not enable in key words) the conversion program ensures correct writing of the Java code, by replacing the hyphen with an underscore. The automated corrections on the part of the transformation function can be avoided by excluding the hyphen character when giving names to ontology components.

## 3 TRANSFORMATION OF CLASSES

The purpose of the transformation of classes declared in ontology is to create interfaces, and also classes that will implement these interfaces in the Java language. When performing the transformation, the rule is valid that for each *named class* of OWL just one interface and just one class is created in Java. Anonymous classes, which OWL allows to create as a result of set operations over the named or anonymous classes (Bechhofer, 2004), cannot be transformed into the Java code.

For a class in OWL, holding its position at the highest level in the hierarchy of classes, the interface is derived from the interface *OWLIndividual* (`public interface DomainConcept extends OWLIndividual {}`) and the appropriate class is derived from the class AbstractCodeGeneratorIndividual (`public class DefaultDomainConcept extends AbstractCodeGeneratorIndividual implements DomainConcept {}`).

Other interfaces and classes are derived depending on the location in the hierarchy recorded in OWL. It is vital just in order some of the information carried by the OWL classes to be transferred into the Java classes. Unlike the classes in Java, the OWL classes do not dispose of any methods. However, they can have defined datatype properties (Chapter 4 Transformation of datatype properties), which are inherited in the hierarchy of OWL by a class from a superclass. The transformation of OWL into the Java code ensures each datatype property to be transformed into a set of methods of Java classes. These derived Java classes then possess the methods of their ancestors. Examples can be the methods MapName, `Legend`, `MappedArea`, `Scale`, `Imprint` and possibly other (Brewer, 2005) that were inherited by the class `ThematicMapPage` from the superclass `MapPage`.

Based on experiments during the transformation with the standard feature `Generate Protégé-OWL Java Code` we verified that all the generated Java classes are named equivalently to the original classes, as defined in OWL. At the same time, the names of the generated Java interfaces are derived from the names of Java classes.

We cannot forget the fact that from this way generated classes it is not possible to create their instances directly by using the operator *new*. Protégé, in addition to all of Java interfaces and classes, creates even the so-called *factory class*. Just this class is used to create objects, thus instances of individual classes. In the object-oriented terminology of design patterns, this type of class is the implementation of the design pattern `Factory`. In (Pecinovský, 2007), `factory` is described as a design pattern, which declares an interface with the method for obtaining the object. However, it lets their descendants, i.e. overlapping versions of the declared method, to make their own decision on the specific type of returned object. As indicated in (Dietrich, 2005), there are abstract (AbstractFactory) and concrete (ConcreteFactory) classes. The `Factory` class name is one of the possible parameters in the dialogue for generating the Java code (Chapter 6 **Configuration of the procedure for generating Java classes**) and can be specified as required; otherwise the default name is used.

Important concepts that can be applied in the design of taxonomy of classes in OWL are a *disjoint axiom* and *covering axiom* of a class. By means of the disjoint axiom, which is not expressed in OWL implicitly, the disjoint of sibling classes is explicitly determined. By contrast, the disjoint of classes that have a common ancestor, is expressed in Java implicitly. The transformation of the explicit expression of the disjoint axiom from OWL to the Java code thus becomes meaningless. The simultaneous use of the disjoint axiom among sibling classes and the set operation of union of sibling classes is applied in OWL to express the class covering axiom. When converting OWL into the Java code, the information about using this axiom is lost.

Although the classes in OWL can have a defined *cardinality constraint* (Bechhofer, 2004; Athanasiadis, 2007), the fact is not reflected in any way in the generated Java classes in case of applying this type of constraint. However, the situation can be solved, especially when the creating of instances of classes is permitted only through the aforementioned class `Factory`. One way of the solution could be the implementation by means of static attributes of those Java classes that have declared the cardinality constraint in OWL. In this case, however, this functionality had to be contained in the Protégé editor. Another option is to create an own class that would be able to obtain the knowledge from the source OWL file and subsequently process it in a needful way for further use in the Java code.

## 4 TRANSFORMATION OF PROPERTIES

*Properties* are important components of ontology, as they represent binary relations between two individuals (Smith, 2004). The transformation of properties takes place in a different way for each of the three types of properties (Patel-Schneider, 2004; Smith, 2004), which may occur in the OWL ontology:

- object properties,
- datatype properties,

- annotation properties.

The article, however, deals only with the transformation of object and datatype properties that are relevant with regard to the further development of the knowledge-based application.

### 4.1 Transformation of object properties

The function `Generate Protégé-OWL Java Code` will transform the object properties by creating a set of declarations of methods in an interface and a set of definitions of methods in classes. Each declared method of an interface and each method definition in a class - *subject*, describes the relation to another declared class - *object* (Hong, 2008). The term subject thus corresponds to the concept *D(f) property domain*. The concept object corresponds to the term *H(f) property range*.

An object property thus establishes a relation *between a subject and an object*. An example is the object property `isMapElementOf`, which establishes the relation between the individuals of the class `MapElements` and the individuals of the class `MapPage`. Then for any individual, such as `Legend`, of the class `MapElements` a relation exists to any individual, for example `ThematicMapPage`, of the class `MapPage`:

`Legend isMapElementOf ThematicMapPage`

When transforming the object property `isMapElementOf`, the transformational procedure `Generate Protégé-OWL Java Code` created a set of declarations of methods in interfaces and a set of definitions of methods in classes that are able to describe these relations.

If it is the object property `isMapElementOf`, characterized as *functional*, then in the Java code the relation between the individual from the domain of definition and maximum one individual from the range of definition was created by the transformation. Just as for datatype properties, also here the differences are valid in the created methods depending on the choice of characteristics - *functional*, or *inversely functional* characteristics (Fig. 2) or without these characteristics.

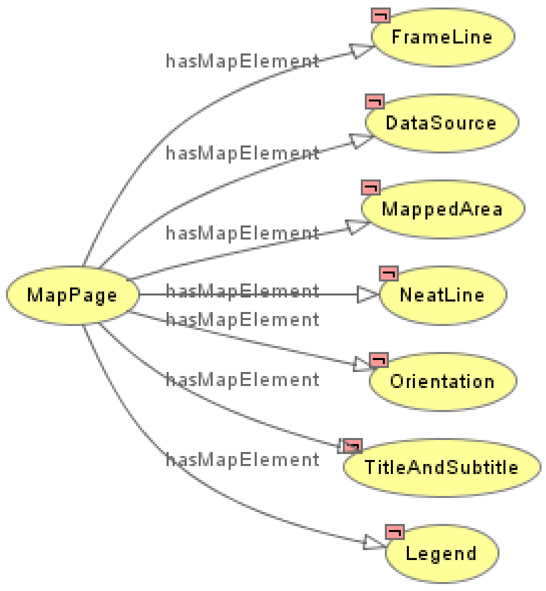


**Fig. 2** Scheme of the object property `hasMapElement` inverse to the object property `isMapElementOf`

The selected illustration defines for this example the classes `ThematicMap` and `GeneralReferenceMapFeatureDataset`. The class `ThematicMap` represents a thematic map, the class `GeneralReferenceMapFeatureDataset` represents elements of the map base. If within this ontology there are the following mutually inverse object properties:

`ThematicMap hasGeneralReferenceMapFeature GeneralReferenceMapFeatureDataset`

`GeneralReferenceMapFeatureDataset isGeneralReferenceMapFeatureOf ThematicMap`

then assuming that the object properties `hasGeneralReferenceMapFeature` and `isGeneralReferenceMapFeatureOf` will not be identified in the ontology as *functional* and *inverse functional* characteristics, the following interfaces will be created:

```
public interface ThematicMap extends Map {
Collection getHasGeneralReferenceMapFeature();
RDFProperty getHasGeneralReferenceMapFeatureProperty();
boolean hasHasGeneralReferenceMapFeature();
Iterator listHasGeneralReferenceMapFeature();
void addHasGeneralReferenceMapFeature(GeneralReferenceMapFeatureDataset
newHasGeneralReference MapFeature);
void removeHasGeneralReferenceMapFeature(GeneralReferenceMapFeatureDataset
oldHasGeneralReferenceMapFeature);
void setHasGeneralReferenceMapFeature(Collection newHasGeneralReferenceMapFeature);
}
public interface GeneralReferenceMapFeatureDataset extends DomainConcept {
Collection getIsGeneralReferenceMapFeatureOf();
RDFProperty getIsGeneralReferenceMapFeatureOfProperty();
boolean hasIsGeneralReferenceMapFeatureOf();
Iterator listIsGeneralReferenceMapFeatureOf();
void addIsGeneralReferenceMapFeatureOf(ThematicMap newIsGeneralReferenceMapFeatureOf);
void removeIsGeneralReferenceMapFeatureOf(ThematicMap oldIsGeneralReferenceMapFeatureOf);
void setIsGeneralReferenceMapFeatureOf(Collection newIsGeneralReferenceMapFeatureOf);
}
```

If the object property `hasGeneralReferenceMapFeature` has only the *functional* characteristic, the object property `isGeneralReferenceMapFeatureOf` will automatically obtain the *inverse functional* characteristic only. In this case, the object property `GeneralReferenceMapFeatureDataset` remains unchanged, but the interface `ThematicMap` will undergo significant changes:

```
public interface ThematicMap extends Map {
GeneralReferenceMapFeatureDataset getHasGeneralReferenceMapFeature();
RDFProperty getHasGeneralReferenceMapFeatureProperty();
boolean hasHasGeneralReferenceMapFeature();
void setHasGeneralReferenceMapFeature(GeneralReferenceMapFeatureDataset
newHasGeneralReferenceMapFeature);
}
```

The last option takes into account the fact that both mentioned object properties were assigned functional and inverse functional characteristics in the ontology. The generated interface `ThematicMap` will then be the same as in the previous case and the interface `GeneralReferenceMapFeatureDataset` will look like this:

```
public interface GeneralReferenceMapFeatureDataset extends DomainConcept {
ThematicMap getIsGeneralReferenceMapFeatureOf();
RDFProperty getIsGeneralReferenceMapFeatureOfProperty();
boolean hasIsGeneralReferenceMapFeatureOf();
void setIsGeneralReferenceMapFeatureOf(ThematicMap newIsGeneralReferenceMapFeatureOf);
}
```

Object properties are therefore a very important component of the ontology, by the transformation of which into the Java code only the part of knowledge describing horizontal relations between classes can be transferred. The described conversion, however, failed to transfer the knowledge about constraints of object properties, which allowed specifying in the ontology the semantic meaning of relations between individuals and classes very precisely.

## 4.2 Transformation of datatype properties

Datatype properties can be transformed directly into data types of the Java language with corresponding data types (e.g., the properties of type `xsd:String` to the field of type `String[]`etc.) (Kalyanpur, 2004). They can be considered as relations between the instances of classes and RDF literals (Huiji, 2009).

The data types of ontology datatype properties are transformed into data types in Java according to Tab.1.

**Tab. 1** Corresponding data types in Java and OWL

| Data type in OWL | Data type in Java |
|---|---|
| boolean | boolean |
| float | float |
| int | int |
| string | String |
| date | RDFSLiteral |
| dateTime | RDFSLiteral |
| time | RDFSLiteral |
| any | Object |

The significant differences in the generated Java code of these data attributes are caused by a switch, by which the datatype property can be marked as a *functional* characteristic in the design of ontology. If the datatype property is marked as *functional*, the generation of interfaces with the following declaration of methods will occur during the transformation of the ontology (in the ontology, the existence of the datatype properties named `Size` and *int* data type is expected):

```
int getSize();

RDFProperty getSizeProperty();

boolean hasSize();

void setSize(int newSize);
```

In the event that the datatype property is not designated as *functional*, the following declarations of methods are created in the interface:

```
Collection getSize();

RDFProperty getSizeProperty();

boolean hasSize();

Iterator listSize();

void addSize(int newSize);

void removeSize(int oldSize);

void setSize(Collection newSize);
```

Additional programming works that will use the described declarations of methods will vary significantly.

The options of transforming datatype properties of classes of the OWL ontology to data types in the Java code are very interesting from the viewpoint of knowledge sharing during the development of a knowledge-based application. The creator of ontology has a way available how to describe declared classes for purposes of subsequent development of an application in Java. The programmer obtains the names of datatype properties of classes and their data types after converting the ontology into the Java code.

## 5 TRANSFORMATION OF INDIVIDUALS

When transforming the ontology components, denoted as *individual*, we recorded the smallest loss of information, expressing the datatype properties of individuals. The instances represented by individuals themselves are not directly transformed into the Java programming code. Only the classes whose instances are these individuals are transformed. These classes include then the methods developed on the basis of the used datatype properties. The logic of the declarations of methods in interfaces and the definitions of methods in classes is analogous to the transformation of datatype properties described in the preceding section. A key role is played again by the characteristic of properties, known as *functional*. For example, by the transformation of the datatype property `dimension_height` (of type *float)* of the individual A4_Portrait of the class `PaperFormat` characterized as *functional*, the following declarations of methods in the interface will be created:

```
float getDimension_height();

RDFProperty getDimension_heightProperty();

boolean hasDimension_height();

void setDimension_height(float newDimension_height);
```

If this property is not characterized as *functional*, then the generated declarations of methods will have the following form:

```
Collection getDimension_height();
RDFProperty getDimension_heightProperty();
boolean hasDimension_height();
Iterator listDimension_height();
void addDimension_height(float newDimension_height);
void removeDimension_height(float oldDimension_height);
void setDimension_height(Collection newDimension_height);
```

The specific values of the datatype properties of selected individuals can be obtained programmatically using the Java code. The following example demonstrates the possibility of obtaining the value `dimension_height` of the individual `A4_Portrait` of the class `PaperFormat`.

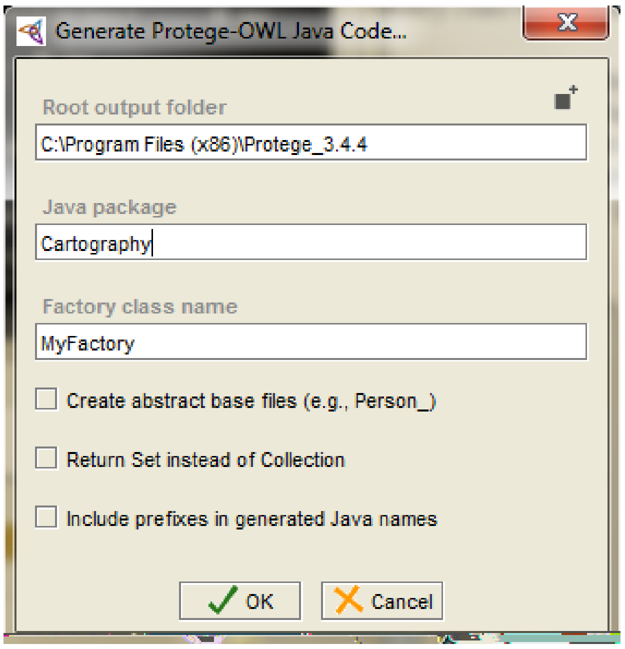
```
...
File owl = new File("TcOntology.owl");
OWLModel model = ProtegeOWL.createJenaOWLModelFromURI(owl.toURI().toString());
TcFactory tcF = new TcFactory(model);
System.out.println(tcF.getPaperFormat("A4_Portrait").getDimension_height());
...
```

In the described example, the Java programmer has an option to obtain specific page dimensions of the standard A4 size, oriented vertically. The prerequisite is, however, the presence of these specific values in the source knowledge ontology.

## 6 CONFIGURATION OF THE PROCEDURE FOR GENERATING JAVA CLASSES

The optional configuration item *Java package* can contain a Java package name. For example, the name *Cartography* is then taken into account in the generated interfaces such as the command *package Cartography* and in classes as *package Cartography.impl*. As the name suggests, another optional item *Factory class name* is intended to indicate the name factory of a class.

If not specified, the default name *MyFactory* is used. Using the option *Create abstract base files* (e.g. Person_) the next level (of inheritance) in interfaces and classes will be created. For example, according to Fig. 3, the interface *public interface Map_ extends DomainConcept* (the interface name is Map with underscore) is created in this case. This interface contains declarations of methods. In addition to this interface, the interface *public interface Map extends Map_* is created too that does not declare any own methods any more. Similarly, it is also in the case of classes.



Root output folder: C:\Program Files (x86)\Protege_3.4.4

Java package: Cartography

Factory class name: MyFactory

☐ Create abstract base files (e.g., Person_)

☐ Return Set instead of Collection

☐ Include prefixes in generated Java names

**Fig. 3** Configuration dialogue of parameters to generate the Java code

Collection can optionally be replaced, if necessary, by a Set. For that purpose the configuration parameter named *Return Set instead of Collection* is used.

The last item, which can be affected by the form of resulting interfaces and classes, is the parameter *Include prefixes in generated Java names*. If the name of a class in OWL starts with a prefix followed by a colon, an interface and a class is created by default, named only by the name without any prefix and colon. Using this option, the name of the interface and the class will consist of a prefix; the colon will be replaced with underscore followed by the name of the interface or class itself.

## 7 DISCUSSIONS

Creating software applications, which use from the beginning only a "manual" record of the Java code in a traditional development environment, is currently still very popular. This established approach may seem faster at first glance, as it does not take into account the time required to work with some of the tools for the visual design of classes. On the contrary, a considerable disadvantage is the lack of a compact view of the structure of classes, their inheritance, and types of relations. Another very substantial disadvantage of the "manual" record of the Java code is obvious in case of a requirement for subsequent refactoring. Modern development environments like Eclipse or NetBeans, however, allow programmers to solve this problem.

The use of visual design tools (e.g. Visual Paradigm for UML, ArgoUML, etc.) offers a compact view of the structure of classes, their inheritance, types of relations. The advantage of the visual design is evident in the situation where the designer returns to the project after a long time. Almost immediately he can see the current status and bindings and can start editing. In practice, an analyst or a designer is very often in charge of the design of classes, or their properties and declarations of methods. The code of methods or the definition of methods is often up to the programmer. It is quite difficult to imagine that in a more complex project, which counts tens, hundreds or even thousands of classes, the designer could keep the structure of classes in the code entered in traditional development tools only.

The knowledge ontology designed and built with the intent to provide declarative knowledge to be shared in the Java programming code of an upcoming application is a very interesting environment for designing classes and their properties and mutual relations. The ontology editor Protégé 3.4.4 is the environment in which the design of knowledge ontology is performed. The resulting ontology, prepared to be shared in the development of a knowledge application, is a knowledge database, whose significant part can be transformed into Java interfaces and classes. As shown by experiments, the automated transformation of ontology leads to loss of some knowledge, which was recorded in the ontology by relevant constructs. The reason is probably the difference between the OWL conception as an ontological language, where the set operations over classes are essential, and the Java programming language, where these relations have no meaning in the object design. Java is not so strict and lacks language constructs, which would be able to define these relations. It is then entirely at the programmer's discretion and abilities in which way he will use these classes in his application; whether he will not consider the relations of classes at all or create his own code, by means of which he will implement them in some way.

Unlike the modern tools for visual design, the functionality of Protégé 3.4.4 is limited to visualizing the knowledge ontology designed using one of the specialized tools (OntoViz, OLS2OWL, OntoSphere3D), which can be used as a plug-in. These tools enable to visualize the ontology as a whole or to visualize selected branches of the hierarchical structure. Using a series of parameters, it is possible to confine only to the view of the hierarchical structure of taxonomy of classes or, conversely, to make visible object properties of classes (i.e., horizontal relations between classes). The view of some restrictions of object properties is provided in a simple form. After possible editing an ontology component, it is possible to prepare the view again, this time for the changed conditions.

The main importance of a knowledge ontology, as an environment for the formalization of declarative knowledge, and the importance of an ontology editor (e.g. Protégé 3.4.4) lies in the ontology classification. Thus it is possible to gain new knowledge derived from existing findings, which were deposited into the ontology before classification. These knowledge ontologies and the editor of ontologies are also a suitable environment for testing a designed ontology, where inconsistencies in the existing ontology can be automatically detected.

## 8 CONCLUSIONS

Based on the up to now activities that we have made in connection with the design of an experimental domain ontology of (thematic) cartography, we managed to realize the transformation of this ontology into Java interfaces and classes. Fragments of the code in the Java language occurred, which contain a significant portion of declarative knowledge listed in the original knowledge ontology. We tried also the possibility of further use of these fragments of the Java code for the development of a knowledge-based application designed for the domain of thematic cartography.

For the transformation of ontology into the Java code, we used a standard procedure from the menu of Protégé 3.4.4 under the name Generate Protégé-OWL Java Code. As the functionality of the used transformation procedure was not entirely obvious in generating the Java code based on the available sources of information, we could not estimate the transformation losses in advance and to adapt the design of the experimental ontology. The partial goal of the experiments was to verify the possible loss to be expected due to the transformation. Based on the performed tests of the transformation course and results we gained valuable experience and could adapt the approach to the ontology design so that the transformation is loss-making as few as possible. In addition to changes in the ontology design, we verified also the behaviour of the transformation procedure for different settings of transformation parameters. We managed to find that an important part of the declarative knowledge of cartography can be expressed through the language OWL-DL so that this knowledge is then automatically transferred into the generated Java code fragments.

Transforming all three types of ontology components (classes, properties and individuals), Java interfaces and classes occurred, named in accordance with the URI that was used in the ontology. The created classes respect the original hierarchy arrangement of the taxonomy of classes and inheritance as well. Similarly, the object properties are transferred, preserving the hierarchical structure and inheritance in the Java code. Although it is possible to define in the ontology datatype properties for classes namely by their name (by the URI identifier), data type and range of values, only the name and data type of the properties is transmitted into the Java code.

During the transformation of the ontology we recorded significantly smaller losses of declarative knowledge when transferring the description of individuals of the ontology. In the Java code, individuals are declared as instances of the class transferred from the original ontology. The programmer gets the possibility to use quite easy specific values, describing the datatype properties of individuals of the ontology. An example is the individual `A4_Portrait` of the class `PaperFormat`, for which the datatype properties `dimension_height` and `dimension_width` of type *float* were stored in the ontology. The generated Java classes and interfaces allow the programmer to work with a specific size of page of a map sheet - A4 format with a vertical orientation.

The aim of subsequent experiments will be to verify the creation of a software application to integrate the database of declarative knowledge and the database of procedural knowledge. At the same time, we will carry out further experiments with the transformation of ontology into the Java program code in order to further reduce the loss of the transmitted knowledge. For this purpose, we will try to use one of the alternative conversion tools, whose application has a positive acclaim in publications in the field of knowledge engineering (Stevenson 2011).

It is therefore possible to search more information in the following period on the Web site (http://cartoexpert.comuf.com/) of the GA CR project no. 205/09/1159.

## ACKNOWLEDGEMENT

This material is part of the project "Intelligent System for Interactive Support of Thematic Map Design" supported by the Czech Science Foundation under Grant No. 205/09/1159. The support of the Czech Science Foundation is gratefully acknowledged.

## REFERENCES

[1] APT, K. R.; HOWARD, A. B. WALKER, A.: *Towards a Theory of Declarative Knowledge.* Morgan Kaufmann Publishers Inc. San Francisco, CA, USA ©1988, ISBN: 0-934613-40-0, pp. 89-148.

[2] ATHANASIADIS, I. N.; VILLA, F.; RIZZOLI, A. E.: *Ontologies, JavaBeans and Relational Databases for enabling semantics programming.* Computer Software and Applications Conference, 2007. SOMPSAC 2007. 31st Annual International. 24-27 July 2007. Volume 2, pp. 341-346. ISSN: 0730-3157. ISBN: 0-7695-2870-8.

[3] BAADER, F.; CALVANESE, D.; MCGUINNESS, D.; NARDI, D.; PATEL-SCHNEIDER, P. F. eds: *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003, ISBN: 9780521781763.

[4] BAADER, F.; HORROCKS, I.; SATTLER, U.: *Chapter 3 Description Logics*. In: Harmelen van Frank, Vladimir Lifschitz, and Bruce Porter, eds., Handbook of Knowledge Representation. Elsevier, 2007. Available online: (cit. 30 August 2010):

http://www.comlab.ox.ac.uk/people/ian.horrocks/Publications/download/2007/BaHS07a.pdf

[5] BECHHOFER, S. et al.: *OWL Web Ontology Language.* Reference. W3C Recommendation 10 February 2004. Document Status Update, 12 November 2009. Available online: (cit. 2 June 2011): http://www.w3.org/TR/owl-ref

[6] BREWER, C.A.: *Designing Better Maps: a Guide for GIS Users.* ESRI Press 2005

[7] BRUS, J.; DOBEŠOVÁ, Z.; KAŇOK, J. (2009): *Utilization of expert systems in thematic cartography.* In: proceedings International Conference on Intelligent Networking and Collaborative Systems INCoS 2009, Barcelona, Spain IEEE Computer Society Press, pp. 285-289. ISBN: 978-0-7695-3858-7.

[8] DENT, B.D.; TORGUSON, J.S.; HODLER, T.W.: *Cartography: Thematic Map Design* (6th Ed.), ISBN 0072943823, McGraw-Hill, August 2009, 204-224.

[9] DIETRICH, J.; ELGAR, C.: *A Formal Description of Design Patterns Using OWL*. Software Engineering Conference, 2005. Proceedings. 2005 Australian. 29 March – 1 April 2005. Pages 243-250. ISSN: 1530-0803. ISBN: 0-7695-2257-2.

[10] DOBEŠOVÁ, Z.; BRUS, J.: *Coping with cartographical ontology.* Conference Proceedings SGEM 2011, 11th International Multidisciplinary Scientific GeoConfrence. STEF92 Technology Ltd., Sofia, Bulgaria, pp. 377-384, ISSN 1314-2704, DOI:10.5593/sgem2011

[11] FRENZEL, CH.; BIJAN, P.; SATTLER, U.; BAUER, B.: *Mooop – A Hybrid Integration of OWL and Java.* Lecture Notes in Business Information Processing. Volume 83 LNBIP, 2011, Pages 437-447. ISSN: 18651348. ISBN: 978-364222055-5.

[12] GENNARI, J.H.; MUSEN, M.A.; FERGERSON, R.W.; GROSSO, W.E.; CRUBEZY, M; ERIKSSON, H.; NOY, N.F.; TU, S.W.: *The Evolution of Protégé: An Environment for Knowledge-based System Development.* International Journal of Human Computer Studies. Volume 58, Issue 1, January 2003, Pages 89-123. ISSN: 10715819.

[13] GOBIN, B.A.; SUBRAMANIAN, R.K.: *Mapping Knowledge Model onto Java Codes.* Proceedings of World Academy of Science, Engineering and Technology. Volume 61, 2010, Pages 140-145. ISSN: 2010376X.

[14] GRUBER, T.R.: *A Translation Approach to Portable Ontology Specifications,* Knowledge Acquisition, vol.5, issue 2, Academic Press Ltd., ISSN: 1042-8143, 199-220. June 1993.

[15] HONG, T.-P.; DONG, J.-S.; LIN, W.-Y.: *An integrated OWL data mining and query system.* Systems, Man and Cybernetics, 2008. SMC 2008. IEEE International Conference. 12-15 Oct. 2008. Pages 251-255. ISSN: 1062-922X. ISBN: 978-1-4244-2383-5.

[16] HORRIDGE, M.; KNUBLAUCH, H. et al.: *A Practical Guide to Building OWL Ontologies Using the Protégé-OWL Plugin and CO-ODE Tools* (Edition 1.0). The University of Manchester, 27 August 2004. Available online:

http://owl.cs.manchester.ac.uk/tutorials/protegeowltutorial/resources/ProtegeOWLTutorialP3_v1_0.pdf

[17] HUIJI, Z.; ZHILI, W.; ZHIPENG, G.; WENJING. L.: *Design and implementation of mapping rules from OWL to relational database.* 2009 WRI World Congres and Computer Science and Information Engineering, CSIE 2009. Vol. 4, 2009, Article number 5170964, Pages 71-75. ISBN: 978-076953507-4.

[18] HUSÁKOVÁ, M.: *Znalostní technologie I.* 2006, On-line (cit. September 2011): http://lide.uhk.cz/fim/ucitel/fshusam2/lekarnicky/zt1/zt1_index.html

[19] IOSIFESCU-ENESCU, I.; HURNI, L.: *Towards cartographic ontologies or "how computers learn cartography",* In: Proceedings 23rd International Cartographic Conference, 4 - 10 August 2007, Moscow, Russia.

[20] KALYANPUR, A.; PASTOR, D.J.; BATTLE, S.; PADGET, J.A.: Automatic mapping of OWL ontologies into Java. In Maurer F.; Ruhe, G: Proceedings of the 17th Int'l *Conference on Software Engineering {&} Knowledge Engineering* (SEKE'2004), June 2004, pp. 98-103.

[21] KNUBLAUCH, H.; FERGERSON, R.W.; NOY, N.F.; MUSEN, M.A.: *The Protégé OWL Plug-in: An Open Development Environment for Semantic Web Applications.* S.A. McIlraith et al. (Eds.): ISWC 2004, LNCS 3298, pp. 229-243, 2004. Springer-Verlag Berlin Heidelberg 2004.

http://www.bell-labs.com/project/classic/papers/ClassTut/ClassTut.html

[22] MEPHAM, W.; GARDNER, S.: *A Software Framework for Translating ECA Sequences from OWL-DL into Java. Web Intelligence and Intelligent Agent Technology,* 2008. WI-IAT '08, IEEE/WIC/ACM International Conference. 9-12 Dec. 2008. Volume 1, Pages 540-543. ISBN: 978-0-7695-3496-1.

[23] NOY, N.F.; MCGUINNESS, D.L.: *Ontology Development 101: A Guide to Creating Your First Ontology.* Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, March 2001. Available online (cit 25. 5. 2011): http://protege.stanford.edu/publications/ontology_development/ontology101.pdf

[24] PATEL-SCHNEIDER, P.F.; HAYES, P.; HORROCKS, I.: *OWL Web Ontology Language Semantics and Abstract Syntax*. 10 February 2004. Available online: (cit. 20 Jul 2011):

http://www.w3.org/TR/2004/REC-owl-semantics-20040210.

[25] PECINOVSKÝ, R.: *Návrhové vzory.* Computer Press, a.s., 2007, ISBN: 978-80-251-1582-4.

[26] PEŇÁZ, T.: An Ontological Model Building for Application Use of Knowledge in Thematic Cartography Domain. In: *Sborník XXII. sjezdu České geografické společnosti*. Ostrava, 31.8.-2.9.2010.

[27] PULESTON, C.; PARSIA, B.; CUNNINGHAM, J.; RECTOR, A.: *Integrating object-oriented and ontological representations: A case study in Java and OWL.* Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Volume 5318 LNCS, 2008, Pages 130-145. ISSN: 03029743. ISBN: 3540885633; 978-354088563-4.

http://ica.ign.fr/BDpubli/moscow2007/Regnauld-ICAWorkshop.pdf

[28] SLOCUM, T.A.; MCMASTER, R.B.; KESSLER, F.C.; HOWARD, H.H.: *Thematic Cartography and Geovisualization* (3rd Ed.). Prentice Hall, Upper Saddle River, NJ, 2009, ISBN 978-0-13-229834-6, 188-210.

[29] SMITH, M.K.; WELTY, C.; MCGUINNESS, D.L.: *OWL Web Ontology Language Guide. W3C Recommendation* 10 February 2004. Available online: (cit. 30 May 2011): http://www.w3.org/TR/owl-guide/

[30] SMITH, R.A.: Designing cartographic ontology for use with expert systems. In proceedings: *A special joint symposium of ISPRS Technical Commission IV & AutoCarto in conjunction with ASPRS/CaGIS 2010 Fall Specialty Conference*. November 15-19, 2010, Orlando, Florida.

[31] SVÁTEK, V.; ZAMAZAL, O.; PRESUTTI, V.: Ontology Naming Pattern Sauce for (Human and Computer) Gourmets. In: Proceedings *WOP 2009*, Washington, 21. 10. 2009. ISSN 1613-0073, 171-178.

[32] STEVENSON, G.; DOBSON, S.: *Sapphire: Generating Java runtime artefacts from OWL ontologies.* Lecture Notes in Business Information Processing. Volume 83 LNBIP, 2011, Pages 425-436. ISSN: 18651348. ISBN: 978-364222055-5.

[33] STUDER, R.; BENJAMINS, V.R.; FENSEL, D.: *Knowledge Engineering: Principles and Methods.* Data and Knowledge Engineering. Volume 25, Issue 1-2, March 1998, pp. 161-197, ISSN: 0169023X.

[34] USCHOLD, M.; GRUNINGER, M.: *Ontologies: principles, methods and applications.* The Knowledge Engineering Review, Cambridge University Press, Vol.11:2, 1996, pp 93-136. ISSN: 02698889.

## RESUMÉ

Článek podává základní informace o možnostech využití znalostní ontologie pro usnadnění návrhu a vývoje znalostního systému pro tematickou kartografii. Existující doménová ontologie pro tematickou kartografii musí být pro tento účel tranformována na fragmenty Java kódu, který obsahuje deklarativní znalosti přenesené z ontologie. Vytvořený kód, obsahující Java rozhraní a třídy, je využíván pro další vývoj znalostní aplikace, nazvané *Inteligentní systém pro interaktivní podporu tvorby tematických map*. Výhodou popisovaného řešení je skutečnost, že základní množina deklarativních znalostí, využívaných ve znalostní aplikaci, je připravena v prostředí OWL editoru Protégé 3.4.4, který je vhodným grafickým prostředím pro návrh ontologie.

Ontologie zahrnující odborné pojmy, členění těchto pojmů do tříd na základě společných vlastností, dále jejich hierarchické uspořádání do taxonomie, vyjádření vzájemných vertikálních a horizontálních vztahů mezi třídami a jedinci, slouží jako databáze pro sdílení znalostí. V textu jsou naznačeny existující problémy, které plynou z faktické neexistence ontologie pro doménu tematické kartografie, která by byla použitelná pro vývoj znalostních aplikací. Článek vysvětluje potřebu existence doposud chybějící znalostní ontologie pro doménu tematické kartografie. Doménová ontologie, použitá pro potřeby aktivit popsaných v článku, je vytvářena jako experimentální. Požadavkem je následné využití ontologie jako databáze deklarativních znalostí pro potřeby pilotního projektu. Článek shrnuje zkušenosti autorů, které získali při transformaci ontologie na Java kód a formuluje některá doporučení pro konstrukci ontologie jejíž komponenty jsou lépe přenositelné do Java rozhraní a tříd.