The ontology engineering process

Johannes Kinzig

Frankfurt University of Applied Sciences
johannes_kinzig@icloud.com

Abstract. When Tim Berners-Lee started to develop the semantic web in the 1990s the general aim was that computers should "understand" the meaning or sense behind the content they were delivering. The overall purpose was to deliver the content which was requested by the user, faster and with a higher accuracy regarding the search terms. This was achieved by defining meta-data for contents and linking the meta-data. In the former described scenario - the semantic web - it was mainly meta-data for webpages. When the user was searching for a specific topic the search engine was able to parse the meta-data and also resolve the references which were given inside the meta-data (which linked to other corresponding meta-data). Then the user was provided with more relevant and significant content for his search.
Defining descriptive content is named *ontology* and was then adapted for describing other processes, applications or domains. The term *ontology engineering* then was established and describes the process for making up ontologies and complete ontology graphs for processes and applications. Thereby ontology engineering is not limited to technical processes or a special kind of data.
This paper describes the engineering process when developing an ontology or a complete knowledge system. The *ontology engineering process* will then also be compared with the technical engineering- and the software engineering lifecycle.
In addition, this paper will focus on the methods of gathering knowledge from different sources and on some modelling techniques. Software tools will be introduced shortly at the end but will definitely not form the main part of this scientific paper.

1 Introduction

The terms *ontology* and *semantics* were mainly affected in the 1990s when Tim Berners-Lee came up with the idea of categorising information on the world wide web in a way that machines were able to "understand" the context and content of the information they were delivering.
The *world wide web* as it is known today was not yet existing as the www and information was mainly stored internally by several companies and institutes. The storage format used by the companies was often not compatible and therefore disallowed an easy exchange of documents and data between companies. Therefore the need for a global and structured information database was discovered. Information was electronically available and users were able to retrieve the

information but the content was not interlinked to each other. The experienced user was able to understand the content and could link it to it's needs but the computer – which was delivering it – didn't. Therefore a search on an information database or network was complicated and time consuming because information was not automatically sorted and presented regarding its relevance for the search terms. This meant being confronted with lots of information, relevant and irrelevant information in the same manner and with the same probability. The user then had to go manually through the search results and decide which article or topic is relevant in his search context and which is not. [5, p. 10, 11]
From this time on categorising and interlinking information and content on the world wide web was needed. Possible solutions for achieving this could be

1. kind of artificial intelligence
2. defining descriptive methods for the content

The idea behind (1) was to use methods from artificial intelligence which scans the information and performs a categorisation and linking between the information. The AI would then build a network of information and interlinking them in a way a computer can "understand" and a human can read. Then providing the user with relevant information regarding its search terms seems to be far easier part. The network can be parsed by the computer and regarding the users search terms the information is displayed according to the "network of information".
Until now the sciences of artificial intelligence makes more and more progress but scanning a human readable content and classifying like a human would do still seems to be an unsolvable problem. [5, p. 11]
(2) proposes a classification by the user or rather the author of information. The author then classifies his information and interlinks it to other data sources whose authors already performed the classification and linking in a similar way. The storage of the descriptive information for a corresponding content is then also performed in a structure or a graph which machines can read and transform in a way such that a human can read and interact with it. This method is nowadays very common, the payload is described by data in a special way which is named *meta-data*. This method was developed in the the 1990s and was named *Semantic Web*, because at this time it was mainly used for the world wide web. Nowadays this method is used in much more domains than just the web to classify and categorise data or information. Additional examples where "semantic web" technologies are used is setting up and maintaining knowledge systems, cognitive intelligent systems, machine learning, etc. [5, p. 12]

1.1 Applications making use of semantic web technologies - Flickr

Describing the content or payload of data is nowadays a common task and is mainly performed automatically. For instance the social media platform for photographers *Flickr* (`https://www.flickr.com`) relies on this method. When the photographer is taking a picture with a digital camera, the camera takes the foto

and stores it with additional meta-data like camera type and model, aperture, zoom level, what kind of lens was used, time and date, geographical location etc. When the photographers are publishing their photos on *Flickr*, the platform is parsing the meta-data and suggests fotos to other users based on e.g. the camera type or geographical location.
The scenario described above is just a general example how social media platforms make use of the semantic web technologies and just describes the basic workflow roughly visible to the user. For developers *Flickr* offers APIs, protocols and interchange data formats to retrieve meta-data for pictures to allow application programmers to make use of and link to *Flickr's* database. [1, p. 80]

1.2 Prerequisites for semantic web applications

As already described above the prerequisites for developing or using semantic web applications is a meta-data format which is readable by machines. Additionally it is necessary to provide consistent and open standards because this is responsible for exchanging information between several applications or platforms. Beside this, the standard should be closed for modification and open for extension to allow the homogeneous integration with future applications or standards. This aim was achieved by the *W3C* (World Wide Web Consortium, `https://www.w3.org`) which developed standards like RDF(S) and OWL as languages with the main aspect for specifying and interlinking data. These kind of languages are known as "ontology languages". [5, p. 11]

2 Ontology Engineering

In section 1 was described that ontologies and semantic web technologies are nowadays used in far more applications than just on the *WWW*. Modelling processes or knowledge systems can be done in nearly every science. In medicine it can be modelling a disease, allergy or in juristic sciences modelling cases and their dependencies seems to be a common task. This leads to the point that someone is needed who can engineer models for a special semantic when required by a certain science. This is named "ontology engineering" and is definitely not equal to "Software Engineering" or "Technical Engineering" but the engineering processes between the engineering disciplines can be seen as quite similar. They share the ability to create a complex system which needs to work reliably. So looking at the process an engineer needs to follow seems to be good starting point for diving into ontology engineering. [6, p. 307]

2.1 Software Engineering Process

As discussed above - before diving into ontology engineering - a short introduction of the software engineering process seems to be a good start. Software engineering is today a very advanced engineering discipline. Depending on the purpose of the software which is going to be developed a special development

mode and afterwards model should be chosen. In this case - as it only acts as short introduction - the steps occurring in the *waterfall model* are introduced. This allows to get an overall impression of the software engineering process. The *waterfall model* is used when developing safety-critical-systems and demands a relativ static approach. The opposite models are agile models which are mainly consisting of the same steps but are used in a more cyclic manner. (Compare steps described in [4, p. 90] as a static model with steps described in [4, p. 104] as an agile model. Both procedures have different approaches but containing steps are quite similar.) [4, p. 88, 90]
The main steps which have to be covered are:

1. System Requirements
2. Software Requirements
3. Analysis
4. Program Design
5. Coding
6. Testing & Integration
7. Operations

The mentioned steps (1), (2) and (3) mainly concentrate on doing a requirement analysis. At first about the overall system, then the derived system requirements for the software which then become the software requirements. This is followed by the analysis about how to implement and designing the software. Steps (4) and (5) are the consequence of the previous steps, (4) is about making up an architecture for the software module and (5) is the technical realisation, the actual software coding.
The then followed step is (6) testing the system software and integrating the system into its context. The last step (7) is the usage of the system. [4, p. 91, 92]

2.2 Ontology Engineering Process

In this section the engineering process for a knowledge system will be described. It is important to mention that before the actual requirement analysis can take place some preconditions have to be decided. This is the modelling formalism, such as the technical storage and the organisation of the knowledge. Should a semantic approach be used or rather a relational or object-oriented database? If a semantic approach is used then a decision about the language must be made (RDFS, OWL, OWL2). If the amount of data to be stored is large compared to the expressive formalism, RDF (Resource Description Framework) seems to be a good idea, if it is the opposite case (less information to be stored but more expressive means of representation required) OWL seems to be the right choice. This procedure is also described in figure 1. After deciding about the modelling formalism, the requirements can be specified.

Preconditions and Requirement Analysis As already described above, the aim of ontology engineering is the transformation and storage an a computer "understandable" form. Therefore it must be clarified where the information can be taken from and based on this source the formalisation process is different.

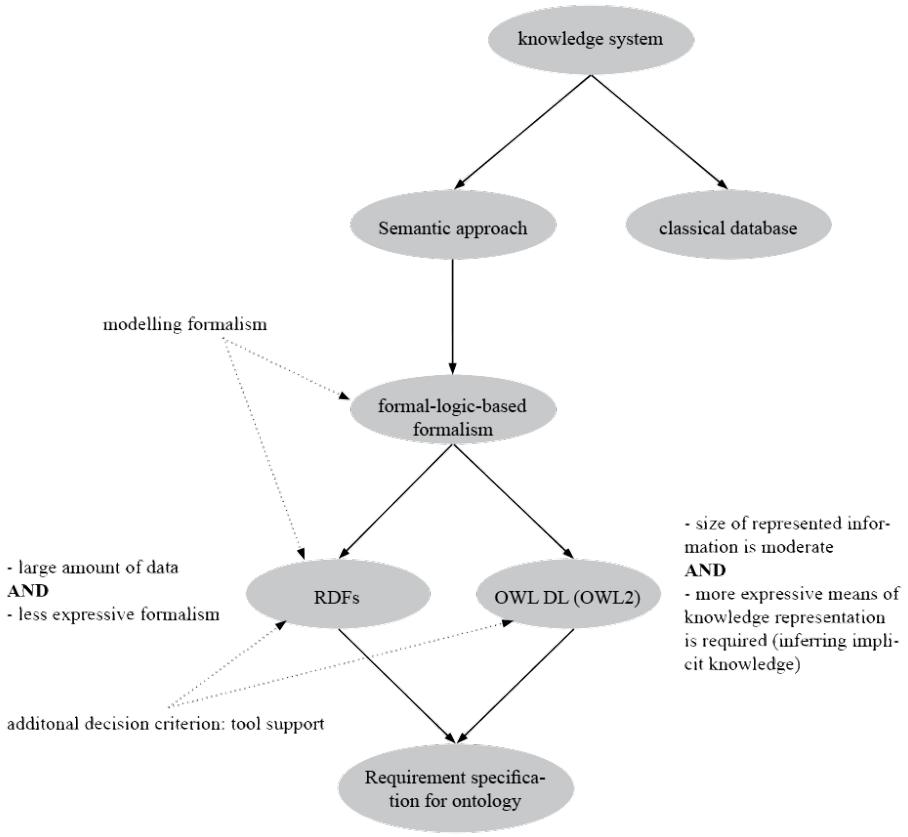

Fig. 1. Pre-analysis for ontology engineering [6, p. 308, 309]

Generally the source of information can be distinguished between "human", "unstructured", "semi-structured" and "structured".
Basically, there is not a "straight-way-to-follow" when modelling a system to build an ontology but there exist some modelling patterns which can improve the workflow. They will be described in section 2.3

Gathering knowledge from human sources A "domain expert" is a person which is an expert in the domain which is to be modelled for the knowledge system. The main issue is that the domain expert most likely will be unable to formulate his knowledge in a format that can directly be used for formal information representation (or the kind of formal representation the modelling language requires). This is the reason why a "knowledge engineer" is required. The knowledge engineer has excellent communication skills and knows the formal representation the "knowledge" has to be stored in. He will then perform interviews with the domain expert to gather all the necessary information and

transforms it into the representation required by the modelling language. To reduce the risk of misunderstandings between the expert and the engineer, feedbacks and double-checking is required to ensure a high-quality knowledge system. Another possibility would be to use machine learning technologies or interactive methods where the expert is actively asked to categorise or classify data. [6, p. 310, 311]

Gathering knowledge from textual resources - unstructured sources
Another method to gather knowledge is using textual resources such as books, magazines and professional journals, etc. The main issue which the ontology engineer has to cope with is the fact that texts, written in natural languages, are accessible by humans only. The interpretation of a text is often dependent on the grammatical structure of the sentences and mostly requires background information to understand the logical meaning correctly.
Automatically extracting information out of texts which are written in a natural language is still a hard problem which cannot be solved initially. Therefore methods exists which allow automatic text analysis. Based on the outcome of these methods the logical meaning and information can be closely reconstructed and then transformed into the representation which is required by the knowledge system. One method is statistical analysis of a text. The grammar is not taken into account, word occurrences are counted and other frequencies are measured. This gives only a rough information what a text is about, but there are cases where this is far enough.
Another approach is "parsing" which implies that a text is grammatically analysed in a specific procedure of steps such as part-of-speech tagging, named entity recognition, chunking, word-sense disambiguation. When finishing such an analysis the outcome is a structured representation of the grammatical composition, such as a parse tree. Depending on the parser the parse tree is different and may contain errors because the meaning in a natural language is mainly dependent on pronouns. [6, p. 312, 313]
The next step is the "formalisation", the linguistic structure is transformed into a logical description. The main conception is based on the assumption that the meaning of a sentence (in a natural language) is based on the meaning of its components. Additionally the grammatical structure of the sentences gives the information about how to combine the partial meanings of the "words" to a global meaning of the sentences. Therefore it seems plausible to infer from the grammatical structure to a logical interdependence. This is technically performed by converting the parse tree of a sentences by applying transformation rules. The outcome then can be (dependent on the kind of formalism needed) an OWL statement, an expression in first order logic (FOL) or something similar which describes a formal relationship between entities. Nevertheless it is likely that the meaning is not flawlessly extracted because some pronouns have ambiguous meanings or the tenses of verbs may have big influence on the logical interpretation of the sentence. To circumvent these issues it would be possible to use a *controlled language* which allows only normalised words or verbs in its

infinitive form. The temporal information is then lost but there exists no best practice to store these information in an ontology language anyway.
Additionally it is important to integrate the knowledge with lexical background information. Communication in a natural language is quite trivial because everybody has some basic lexical background knowledge like the "existence of family structures and the relationship between the family members". When automatically parsing a text a flawless interpretation is possible only when explicitly providing the lexical background knowledge. This can be realised by using *thesauri* like *WordNet* which is a popular database for this purpose `http://wordnet.princeton.edu`. [6, p. 314, 315]

Gathering knowledge from the www - semistructured sources Extracting knowledge out of *www* sources is a comparatively simple task because the information is already linked somehow. Hyperlinks between resources like articles or webpages provide a general structure about the information pieces. This structure can directly be transformed in the desired formal representation (OWL, RDF, etc.). Information is captured mainly on a meta-level: which file types are just, who is the creator, the publisher of a specific content. *mp3* files offer *id3tags* for meta data storage, the same do jpg files (compare section 1.1) and a lot more file types. Additionally some of the content of the "human-read-only" resources can nowadays be analysed automatically like face recognition on fotos or music title identification (`https://www.shazam.com`). [6, p. 315, 316]

Gathering knowledge from databases - structured sources The knowledge which is already stored in a structure requires less work to integrate in its own knowledge system. Information from a relational database can easily be transferred into an OWL or RDF resource. There it is only important to analyse the database beforehand and to decide how to transform a row/table to an OWL statement. The same applies for the schema information which may deliver useful relationship information and can then be used for generating terminological axioms.
Another resource for knowledge are other already existing ontologies. These may be reused partially or fully depending on the grade of granularity. Using and extending an already existing ontology is also far easier than starting from scratch. [6, p. 316, 317]

2.3 Modelling ontologies

Modelling the ontology can be a quite challenging task because there exist no "right" or "wrong" way to model. There exist some rules and conventions the engineer is advised to follow; these will be discussed in the following.
The general approach when beginning the modelling or checking an existing model is to verify the model against the "real world", that is the domain the ontology was developed for. Are the consequences logically following from the reality or are they contradictory. When this is decided one can use a *Reasoner*

to do a model checking. The purpose and the functionality of a reasoner will be discussed in section 3.2. Of course the before mentioned approaches can only be used in the context of a real application but there exist some techniques that can be initially followed.

1. Logical Criteria
2. Structural and Formal Criteria
3. Accuracy Criteria
4. Disjointness
5. Quantification and Quantifiers
6. Part and Subclass Identity
7. Subclasses and equivalent classes
8. Translate loosely from natural languages

(1) deals with the characteristics which can be directly checked on a logical level based on the model.
Inconsistency: An ontology is called *inconsistent* if there is not a correct mapping between the "real world" and the model. The issue is that an inconsistent ontology follows statements as logical consequences but cannot be used for automated deduction. (A class is called inconsistent/unsatisfiable if it is interpreted as an empty set in the model. This is not an issue as long as no instance is added to the class.)
Coherency: An ontology does not contain unsatisfiable classes. A *consistent* ontology can be *incoherent* but a *coherent* ontology cannot be *inconsistent*. Software tools offer automatic checking for these flaws. Incoherency and inconsistency can be prevented by not constraining the model enough. [6, p. 317, 318]

(2) indicates another modelling problem, kinds of taxonomic cycles inside the structure. This behaviour may occur when using classes which have a similar meaning in semantic, but this is rather seldom. The following example shows a taxonomic circle:

$$Architecture \sqsubseteq Faculty$$
$$Faculty \sqsubseteq University$$
$$University \sqsubseteq Building$$
$$Building \sqsubseteq Architecture$$

To prevent such a behaviour the ontology can be checked for rigidity.
Rigidity: A class is called rigid if every member of it cannot stop being a member without loosing existence. A class can be *rigid*, *anti-rigid*, or none-of both, additional characteristics might be *Identity*, *Unity* and *dependency*. [6, p. 319, 320]
Tools exist for automated ontology checking, such as reasoning tools which can determine the set of axioms which are responsible for the inconsistency. Further information about a *semantic reasoner* will be given in section 3.2.

(3) covers the accuracy and granularity of the model for the ontology; mainly the model is verified against its "real-world-domain"; of course this cannot be checked automatically. Therefore it can be verified by a second ontology engineer who can measure some characteristics like number of domain related statements,

number of classes and additionally random testing against some test specification.

(4) demonstrates that is important to disjoint classes when needed. As an example for a possible flaw regard the following model:

$$Woman \sqsubseteq Human, Human \sqsubseteq Man \sqcup Woman, Man \sqsubseteq Human$$
$$Woman(Anna), Man(Steve)$$

Now assume the following statement: ¬Man(Anna) There is no logical reason why "Anna" cannot be a man and a woman but in "real life logic" this makes not much sense of course. To prevent this the classes "Man" and "Woman" have to become *disjoint* which means that there is no individual existing in both classes. [6, p. 321, 322]
(5) shows the issue when it comes to a quantification of classes and individuals. For the ontology it is often very important to express a quantification in a "has-a" relation like "a car has wheels and a motor".

The **existential quantifier(∃)** is used more often than the *universal quantifier (∀)*. An indication in natural language for use of the **universal quantifier∀)** can be seen when statements occur like *nothing but*, *only*, *exclusively*. A misapprehension can easily occur when looking at the following example. One wants to express that "a car has wheels", then the correct translation should be $Car \sqsubseteq \exists has.Wheels$. Often the following and **wrong** translation is performed $Car \sqsubseteq \forall has.Wheels$ but this implies that the car **has only wheels (or nothing)**. This implication is – indeed – wrong.
As a rule of thumb the following two implications can be used in most cases:

- By default the **existential quantifier (∃)** should be used
- The **universal quantifier (∀)** does not guarantee the existence of a respective rule (for all or nothing)

[6, p. 322, 323]

(6) describes the issue which may occur when mixing or not explicitly isolating subclasses and parts. The following example ([6, p. 323]) shows an example where this misconception happened.

$$Finger \sqsubseteq Hand, Hand \sqsubseteq Arm, Arm \sqsubseteq Body$$
$$Toe \sqsubseteq Foot, Foot \sqsubseteq Leg, Leg \sqsubseteq Body$$
$$Arm \sqcap Leg \sqsubseteq \bot$$
(Arm and Leg are disjoint)

This model allows to deduce $Arm(myLeftThumb)$ because the thumb is not only a finger it is also a hand and an arm (at least when following the above mentioned model). In this case the subclass relation *partOf* was used mistakenly. In this case the reason could have been that both subclasses share the property

of "belonging to something". In this case the leads to a logical inconsistence but this can be prevented by introducing a new role e.g. *partOf*. This then leads to the following corrected model:

$$Finger \sqsubseteq \exists partOf.Hand, Hand \sqsubseteq \exists partOf.Arm, Arm \sqsubseteq \exists partOf.Body$$
$$Toe \sqsubseteq \exists partOf.Foot, Foot \sqsubseteq \exists partOf.Leg, Leg \sqsubseteq \exists partOf.Body$$
$$Arm \sqcap Leg \sqsubseteq \bot$$

As the rule of thumb a subclass X (of the parent class Z) can be introduced iff the statement "every X is always a Z" is true. [6, p. 324]

(7) is the counter part to (6) when it is hard do decide wether a class is a subclass or an equivalent class of another. Subclassing can be used when trying to express some characteristic about members of a class. A subclass $LivingInWater$ can express that a member $LivingInWater(fish)$, in this case the fish, does live in water. It can be seen as a *necessary criterion* for being a fish. Now it is important to not imply a *sufficient criterion* out of this characteristic because it is not sufficient for beeing a fish when living in water (such as a moray or plankton also live in water but are no fish). Equivalence statements can be used when iff a class description is **necessary and sufficient**. This relation could look like this $Winner \equiv Player \sqcap \forall hasCompleteCollection.Spades$, a player can only be a winner iff he is playing a game and is holding a whole collection of a spades.

(8) describes a modelling incoherency which may occur when translating to verbally from a natural language. The simplest example can be a misunderstanding when using "and"; it does not always mean an *intersection* between to characteristics or properties. When trying to model the statement: "Staff members and students of the university will get a login account". The "and" will be translated into a **union** $StaffMember \sqcup Student \sqsubseteq \exists have.LoginAccount$ and not into an **intersection** $StaffMember \sqcap Student \sqsubseteq \exists have.LoginAccount$. The **intersection** would express that only those will have a login account who are students and staff members. The difference is visible in figure 2. When unsure about the correct modelling in this position rephrasing and testing can help getting out of the misery. [6, p. 325, 326]

2.4 Conclusion: Engineering Ontologies

In a nutshell it becomes clear that the ontology engineer is not only responsible for the technical approach, the domain specific analysis such as domain and scope, term enumeration, property definition is as important as taking care for a reusable ontology or knowledge system.
Beside, it is remarkable that the engineering processes (introduced in section 2.1 and 2.2) seem to have some similarities but when diving into the topic and deciding about a procedure both disciplines evolve to be diverging. Clearly, both disciplines have to follow a certain (given) procedure and the tasks for the procedures can clearly be defined but from the technical point of view the workflow

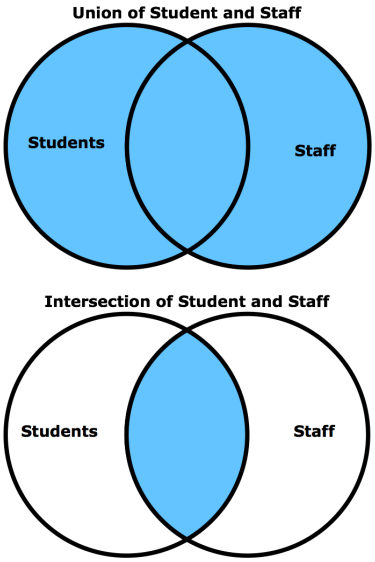

Fig. 2. Difference between intersection and union – students and staff members

is different.
When closely looking at section 2.3 one can notice that these recommendations can be seen as patterns (like design patterns in software engineering). This shows that the sciences of *ontology engineering* evolves (slowly) to a field like software engineering where fixed and important principles and design recommendations are as important as the technical implementation.

3 Software and Tools

In this section some software tools will be introduced which are helpful when engineering an ontology and developing a knowledge system. The aim is not to provide the reader with some tools and describing the pros and cons but to introduce tools which support the developer by acting as a toolchain. For use in a productive environment the tools may need to be specially configured but in this context the prototyping and development is in the foreground so the standard configuration seems to be appropriate.
However, the tools described below are not limited to "develop" an ontology they can of course be used to publish the ontology in a productive environment.

3.1 Protégé

Protégé is an open source application which supports the engineers and developers when editing an ontology. It is available as a desktop application (known

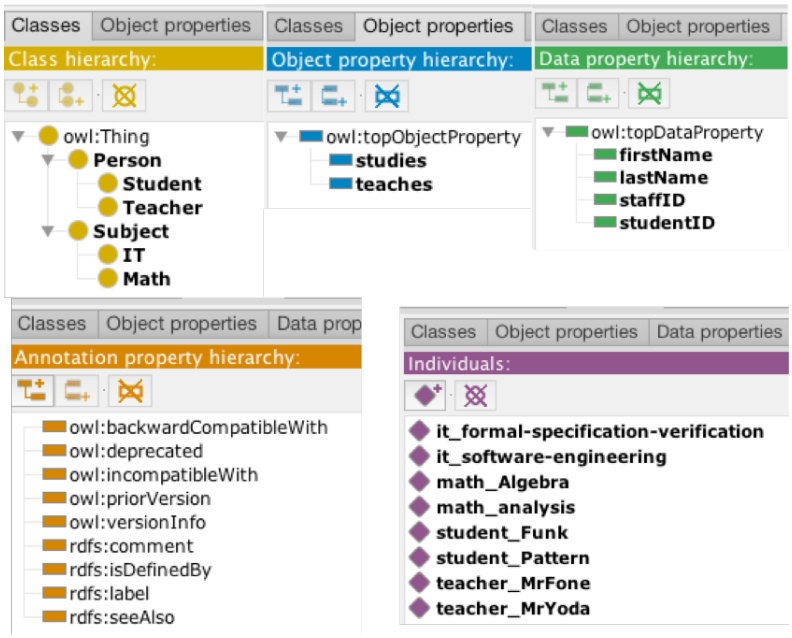

Fig. 3. Excerpt from the Protégé Desktop application showing different ontology object hierarchies

as *Protégé Desktop*) or a web application (known as *WebProtégé*). Protégé supports the OWL 2 Web Ontology Language and allows to export the ontology for further processing or publishing. Protégé is written in Java and offers a GUI and graphical tools for simplifying the engineering process. Figure 3 shows excerpts of the graphical editor, which displays "classes, object properties, data property hierarchies, annotation property hierarchies and individuals" of a sample ontology. [7]

Protégé also includes a *Semantic Reasoner* named *HermiT*. More information about what a reasoner is and how it works will be given in the following section, 3.2.

3.2 Semantic Reasoner

A semantic reasoner is a software or an algorithm which checks an ontology for logical consequences from a given set of facts or axioms. The logical consequences are given by interference rules which can be applied to or taken from individuals and relations of the ontology. Most reasoners use first-order-logic to check the ontology.

The reasoner is able to parse the ontology and apply the rules to the ontology and its parts, relations, classes, individuals, etc. If the reasoner recognises that an individual follows more than one behaviour which are not compatible to each

other (e.g. an individual "Subject:Peter" is at the same time his father and hisself) than the reasoner will throw an error. The ontology has to be changed, the engineer has to eliminate this misconception.

Protégé includes a reasoning engine, named *HermiT* which performes the tasks described above. [3]

3.3 Apache Jena

The *Apache Jena* is an open source framework for publishing and editing ontologies on the web. The software is written in Java and includes several tools:

- RDF - create and read RDF graphs, export the ontology by using RDF/XML or Turtle
- SPARQL - querying and requesting information from the ontology
- TDB - triple store database
- Fuseki - publish the ontology over http/s, access using SPARQL, using web browser or REST-API
- Ontology API - using RDFS to add extra semantics to the ontology
- Interference API - model checking, reasoning engine

The *Apache Jena* project can be seen as a powerful ontology server for distributed environments, it can index, store and query data up to millions of triples. It can be installed as a standalone server application or as a web app into Apache Tomcat. Interaction with the published ontologies is mainly performed via the web browser or by using custom applications through the offered REST-API. [2]

4 Summary and Conclusion

As seen in the sections before, developing an ontology or a knowledge system is not a trivial task. Not just because of the overall complexity, additionally because no easy-to-follow rules (like design patterns in software engineering) exist. Anyhow the introduced rules in section 2.3 can be seen as a guideline to make up an ontology but depending on the size and complexity of the knowledge system the whole engineering task becomes very complex. Nevertheless some good software and tools exist which not only help the engineer while developing, at the same time these tools offer possibilities to publish the ontologies and to allow others to interact and extend the published knowledge system.

Protégé seems to be a helpful tool because it offers graphical interaction with the user. Errors or misconceptions can be detected earlier and can be fixed instantly. The included reasoner *HermiT* allows model checking and seems to use a quite efficient algorithm (compare [3]).

The *Apache Jena* project then can be used to publish the ontology and allow others to interact with it. The included triple store and SPARQL support are just a few features which characterise the *Apache Jena* project. Ontologies are

often offered through a distributed environment, the included REST-API for querying SPARQL commands allow to use one or mote ontologies in customised applications.

All in all ontology engineering not only requires an ontology engineer with good skills, also the tools have to be appropriate and have to be carefully chosen to meet the project requirements.

A lot more tools than described above exist, but the introduced seem to offer a good starting point when diving into the ontology development. Graphical tools and interactions with other ontologies make life much more easier and allow a smooth start into the topic.

References

1. Elena Simperl Ioan Toma Dieter Fensel, Federico Michele Facca. *Semantic Web Services*. Number 978-3-642-19193-0. Springer Berlin Heidelberg, 2011.
2. The Apache Software Foundation. Apache jena. https://jena.apache.org/index.html, 2016.
3. University of Oxford Information Systems Group. Hermit owl reasoner. http://www.hermit-reasoner.com.
4. Marco Kuhrmann Manfred Broy. *Projektorganisation und Management im Software Engineering*. Number 978-3-642-29290-3. Springer-Verlag Berlin Heidelberg, 2013.
5. Sebastian Rudolph Pascal Hitzler, Markus Krötzsch. *Semantic Web*. Number 978-3-540-33994-6. Springer, 2008.
6. Sebastian Rudolph Pascal Hitzler, Markus Krötzsch. *Foundations of Semantic Web Technologies*. Number 978-1-4200-9050-5. CRC Press, 2010.
7. Stanford Center for Biomedical Informatics Research Protege Community, Stanford Labs. Protege desktop user documentation. http://protegewiki.stanford.edu/wiki/Protege4UserDocs, 2016.