6 Protégé

Objective:

Protégé is the most famous tool in the Semantic Web community to build ontologies. This chapter will discuss in detail, the method to create an ontology using Protégé, and the ways to explore the concept of classes and individuals in the ontology.

6.1 Introduction to Protégé

Protege is an open-source tool developed at Stanford Medical Informatics. It has a community of thousands of users. Although the development of Protege has historically been mainly driven by biomedical applications, the system is domain-independent and has been successfully used for many other application areas as well. Like most other modeling tools, the architecture of Protege is cleanly separated into a 'model' part and a 'view' part. Protege's model is the internal representation mechanism for ontologies and knowledge bases. Protege's view components provide a user interface to display and manipulate the underlying model. The protege's model is based on a simple, yet flexible metamodel, which is comparable to object-oriented and frame-based systems. Basically, it can represent ontologies consisting of classes, properties (slots), property characteristics (facets and constraints), and instances.

Protege provides an open Java API to query and manipulate models. An important strength of Protege is that the Protege metamodel itself is a Protege ontology, with classes that represent classes, properties, and so on. For example, the default class in the Protege base system is called STANDARD-CLASS, and has properties such as :NAME and :DIRECT-SUPERCLASSES. This structure of the metamodel enables easy extension and adaption to other representations. For example, this metamodel is extended to handle UML and OWL. Using the views of Protege's user interface, ontology designers basically create classes, assign properties to the classes, and then restrict the properties' facets at certain classes. Using the resulting ontologies, Protege is able to automatically generate user interfaces that support the creation of individuals (instances). For each class in the ontology, the system creates one form with editing components (widgets) for each property of the class.

For example, for properties that can take single string values, the system would, by default, provide a text field widget. The generated forms can be further customized with Protege's form editor, where users can select alternative user interface widgets for their project. In addition to the predefined library of user interface widgets, Protege has a flexible architecture that enables programmers to develop custom-tailored widgets, which can then be plugged into the core system. Another type of plugin supports full-size user interface panels (*tabs*) that can contain arbitrary other components.

In addition to the collection of standard tabs for editing classes, properties, forms, and instances, a library of other tabs exists that perform queries, access data repositories, visualize ontologies graphically, and manage ontology versions. Protégé currently can be used to load, edit and save ontologies in various formats, including CLIPS, RDF, XML, UML and relational databases. Recently, support for OWL was added. Since ontologies played an important role in Semantic Web applications, it was straightforward to take an existing ontology development environment as a starting point. Extensions to Protege can benefit from the generic services provided by the core platform, such as an event mechanism, undo capabilities, and a plugin mechanism. By basing the OWL Plugin on top of Protege, we could also reuse Protege's client-server-based multi-user mode that allows multiple people to edit the same ontology at the same time. Protege also provides a highly scalable database back-end, allowing users to create ontologies with hundreds of thousands of classes.

Also, there is already a considerable library of plugins which can be either directly used in OWL or adapted to OWL with little effort. Furthermore, the fact that Protege is open-source also encourages plugin development. Last but not least, Protege is backed by a large community of active users and developers, and the feedback from this community proved to be invaluable for the development of the OWL Plugin. Our decision to base the OWL Plugin on Protege also had some risks. In order to be able to reuse as much of the existing Protege features as possible, we had to create a careful mapping between the Protege metamodel and OWL that maintains the traditional Protege semantics where possible. Furthermore, none of the generic Protege widgets and tabs is optimized for OWL, and not all of the editing metaphors for frame based systems are appropriate for OWL. In particular, OWL's rich description logics features such as logical class definitions required special attention. The following sections will show how we have addressed these issues.

6.1.1 How to develop Ontology?

Developing ontology in theoretic terms is already discussed in the previous chapter. The following steps will help to develop ontology practically:-

1. Define classes in the ontology
2. Arrange the classes in a subclass-superclass hierarchy
3. Define slots and describe allowed values for these slots
4. Fill-in the values of slots for instances

6.2 Files in Protégé:

When you use Protégé to create and edit ontologies you will generate at least two files:

- **Project file**
  A project file has an extension of .pprj. The project file stores information related to any interface customizations or editor options you have selected. In many cases, you don't have to send this file to colleagues with your ontology unless you have been customizing forms with the Forms Tab. Project files created with older versions of Protégé-OWL (especially prior to version 3.0) may not be compatible with your current version, in such case you may have to build a new project from the scratch.

- **Source file**
  A source file has an extension of .owl, .rdfs, or .rdf. This file is where your ontology classes, individuals and properties are defined. There may be several source files depending on how the ontology has been defined. If it is modular and has been created properly, Protégé-OWL will find and load all of the appropriate source files.

- **Classes**
  Ontology classes are very similar to classes in an object oriented program. Just like object oriented programming, classes in ontologies also form an hierarchy. You can view this hierarchy in the right-hand side panel, labeled as Asserted class hierarchy. The root class which is at the top of the inheritance hierarchy is Thing (this is true of all OWL ontologies).

6.2.1 How are Ontology Classes different from Object-oriented Classes?

While they are similar in many ways, one important difference to remember is that, an individual of ontology can belong to zero or more classes, in addition to any inherited class. There are ways to limit the classes an individual of one class can belong to but if such explicit information is absent then individuals can be members of, as few or as many classes as the ontology creator wants them to be.

- **Equivalent classes**
  This section describes other classes or groups that are equivalent to the selected class.

- **Superclasses**
  Superclasses are the parent class/classes of the selected class (since you can have multiple classes you can also have multiple superclasses).

- **Members**
  Members represent individuals which are members of a particular class. They can be added explicitly, or inferred later through reasoning.

- **Disjoint classes**
  They allow you to explicitly select the classes that members of the selected class cannot belong to. Most of the time, you do not have to explicitly mark classes as disjoint but it can be helpful if you are using some external reasoning or application to make the class definitions as explicit and exact.

  Browse through the classes provided and try to get a feel of the concepts they are trying to describe. After you are done looking at the classes go to the top menu and select Reasoner -> Fact++. This will turn on the reasoner to infer additional facts about the classes.

- **Object and Data properties**
  While attaching properties to classes, it makes sense to immediately provide statements about the domain and range of these properties. There is a methodological tension here between generality and specificity. It is useful to define the domain and range as narrowly as possible, so that it will be easy to detect potential inconsistencies and misconceptions in the ontology by spotting domain and range violations.

- **OWL properties represent relationships between two objects.**

  There are two main properties:

  1. Object properties link object to object.
  2. Data-type properties link object to XML Schema datatype or rdf:literal

Some other important properties are:

Annotation properties

This property can be used to add annotation information to classes, individuals, and properties.

Inverse properties:

1. Every object property may has a corresponding inverse property.
2. If some property links individual A to individual B, then its inverse property will link individual B to individual A.

Functional properties:

1. If a property is functional for a given individual, then only one individual can be related via this property.
2. Functional properties are also known as single valued properties.

Inverse functional properties:

If a property is inverse functional, then its inverse property is functional.

Transitive properties:

If a property P is transitive, and the property relates individual A to individual B, and also individual B to individual C, then we can infer that individual A is related to individual C via property P.

Symmetric properties:

If a property P is symmetric, and the property relates individual A to individual B, then individual B is also related to individual A via property P.

6.2.3 Some important terms

Cardinality:

Cardinality must be specified for as many properties as possible in order to indicate whether they are allowed or required to have a certain number of different values or not. Often, occurring cases are 'at least one value' (i.e., required properties) and 'at most one value' (i.e., single-valued properties).

Required values:

Often, classes are defined by virtue of a certain property having particular values, and such required values can be specified in OWL, using owl:hasValue. Sometimes, the requirements are less stringent, for instance-

A property may have some values from a given class (and not necessarily a specific value, owl:someValuesFrom).

Relational characteristics:

It is concerned with the relational characteristics of properties, i.e., symmetry, transitivity, inverse properties, functional values.

Property domain and ranges

Properties link individuals from the domain to individuals from the range. OWL uses domain and range as axioms in reasoning.

Property restrictions:

In OWL, properties are used to create restrictions. Restrictions are used to restrict the individuals that belong to a class.

There are five restrictions:

- Quantifier restrictions
- Existential quantifier
- Universal quantifier
- Cardinality restrictions
- hasValue restrictions

6.3 Instances

We use ontologies to organize sets of instances and there is a separate step to fill the ontologies with instances. Typically, the number of instances is larger than the number of classes in the ontology.

Ontologies vary in size from a few hundred classes to tens of thousands of classes; the number of instances varies from hundreds to hundreds of thousands, or even larger. Because of these large numbers, populating ontology with instances is typically not done manually. Often, instances are retrieved from legacy data sources such as databases. Another, often used technique is the automated extraction of instances from a text corpus.

6.3.1 Creating a sample project

Note:

There are two main ways of modeling ontologies:

- Frame-based
- OWL

Each has its own user interface.

1. Protege Frame editor: It enables users to build and populate ontologies that are frame-based, in accordance with OKBC (Open Knowledge Base Connectivity Protocol). It consists of –
   - Classes
   - Slots for properties and relationships
   - Instances of class

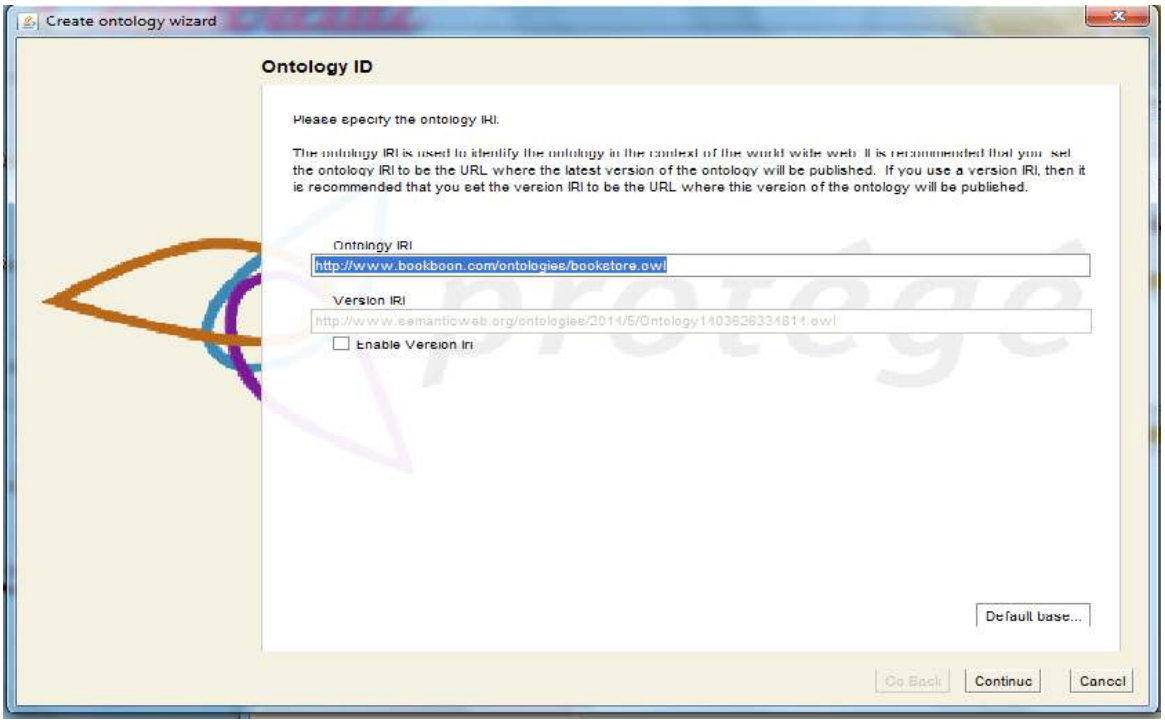
2. Protege OWL editor:
   It enables users to build ontologies for the Semantic Web. It consists of-
   - Classes
   - Properties
   - Instances
   - reasoning

Tool used: Protégé

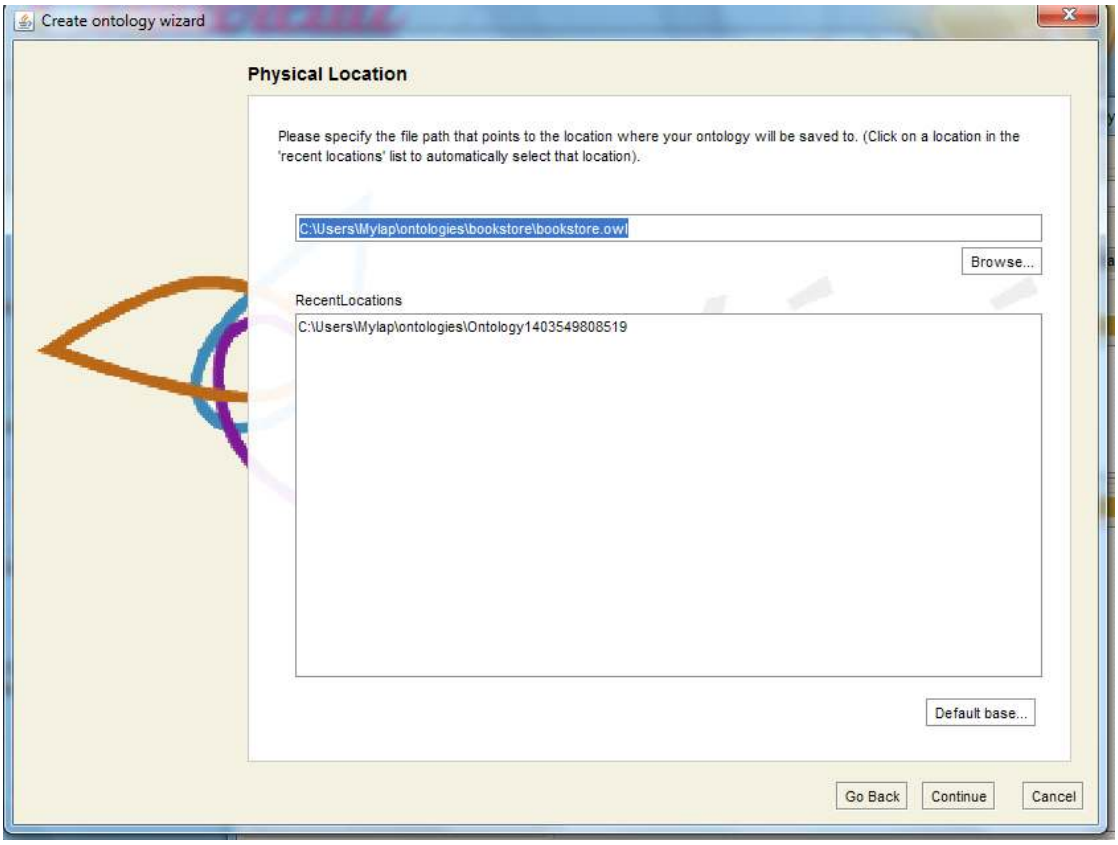
Step 1:

1. Start protégé
2. Go to File
3. Choose New
4. Enter the Ontology URI (**http://www.bookboon.com/ontologies/bookstore.owl**)
5. Then choose RDF/XML
6. Choose Properties View

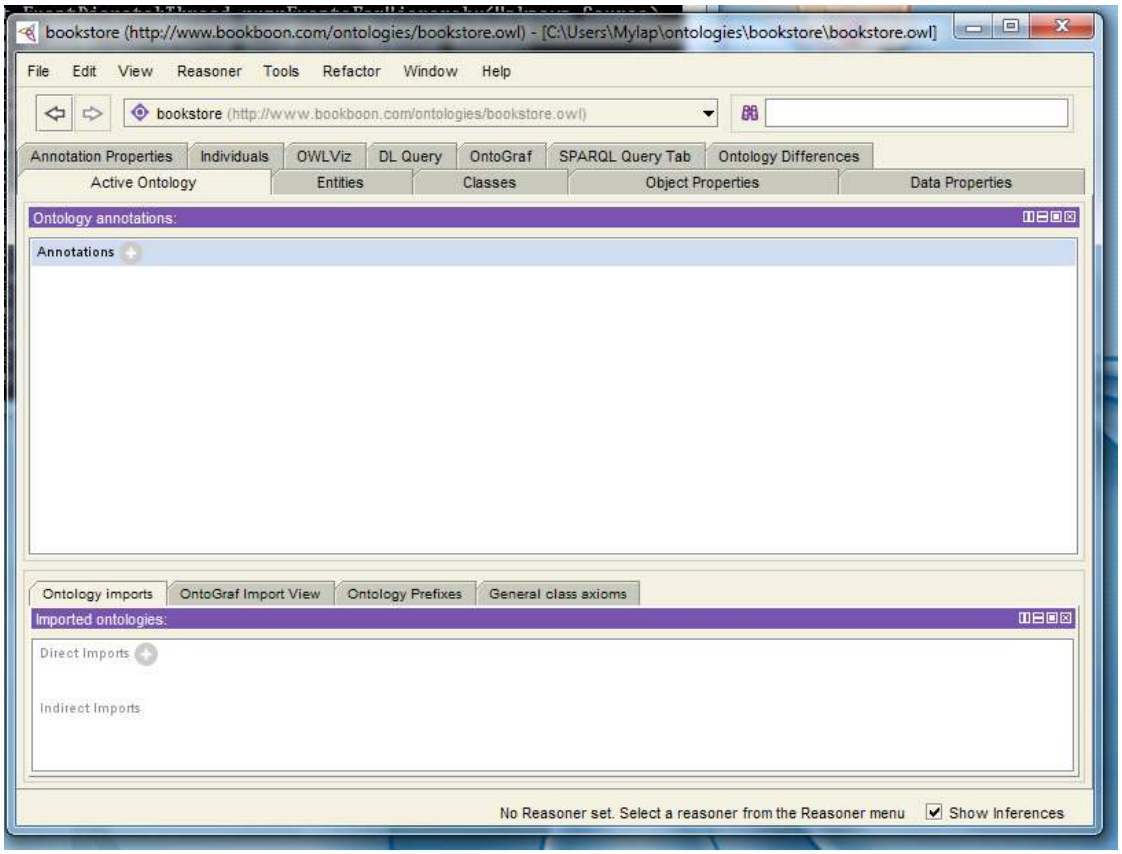
A new, empty Protégé-project has been created. Save it in your local file as bookstore.owl. Refer the following screenshots for better understanding:



Screenshot 6.1: Create Ontology Wizard



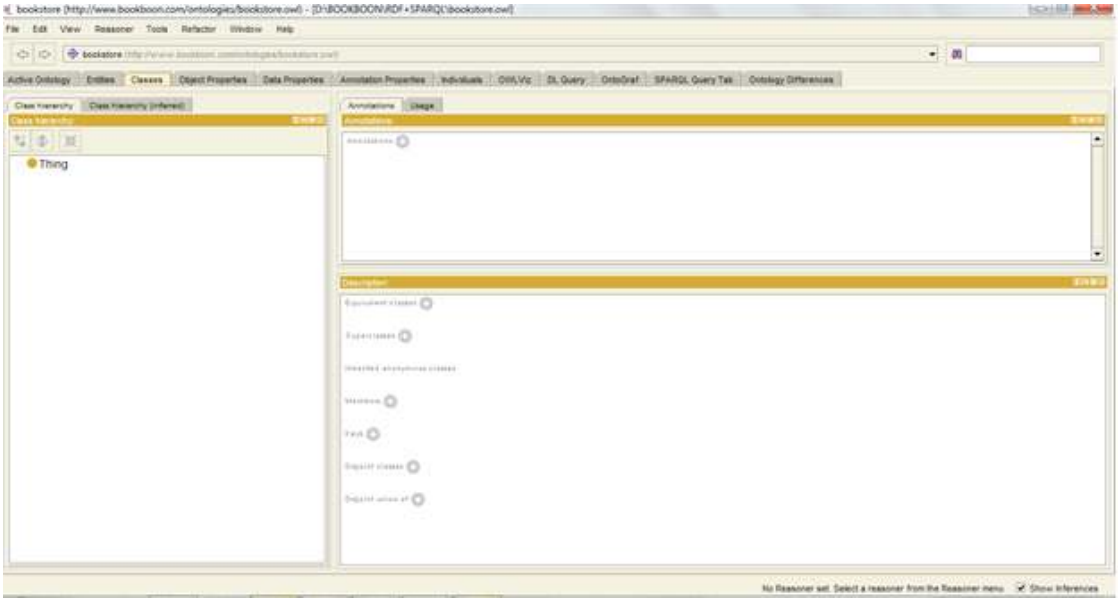
Screenshot 6.2: Create Ontology Wizard – Specifying the location to store the .owl file



Screenshot 6.3: Protégé interface

Step 2:

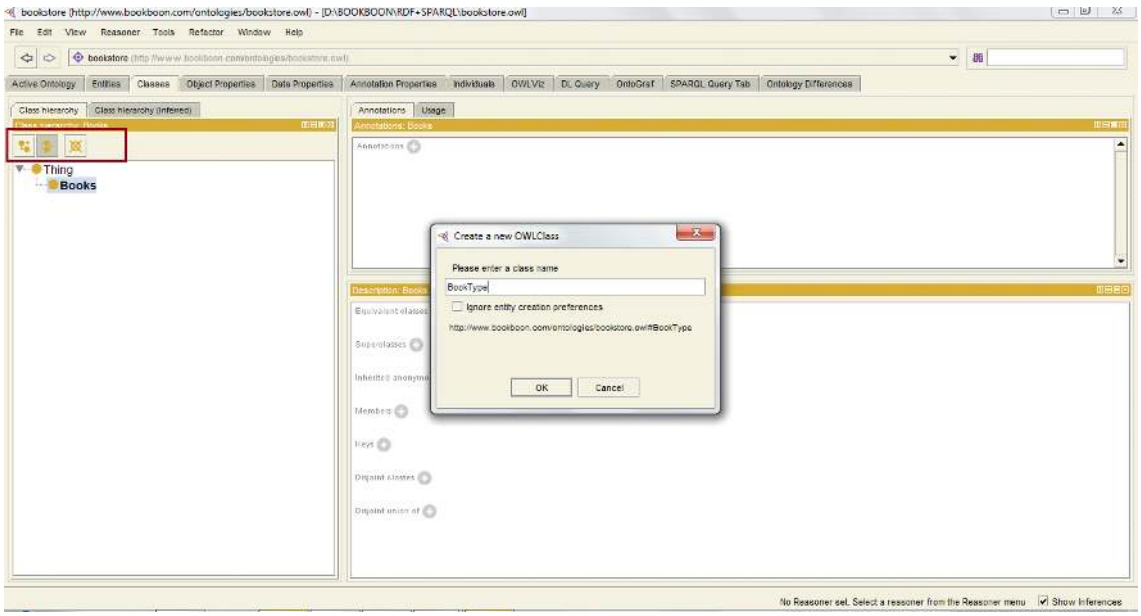
Go to Classes tab. You will find that the empty class tree contains one class called owl:Thing, which is the superclass of everything.



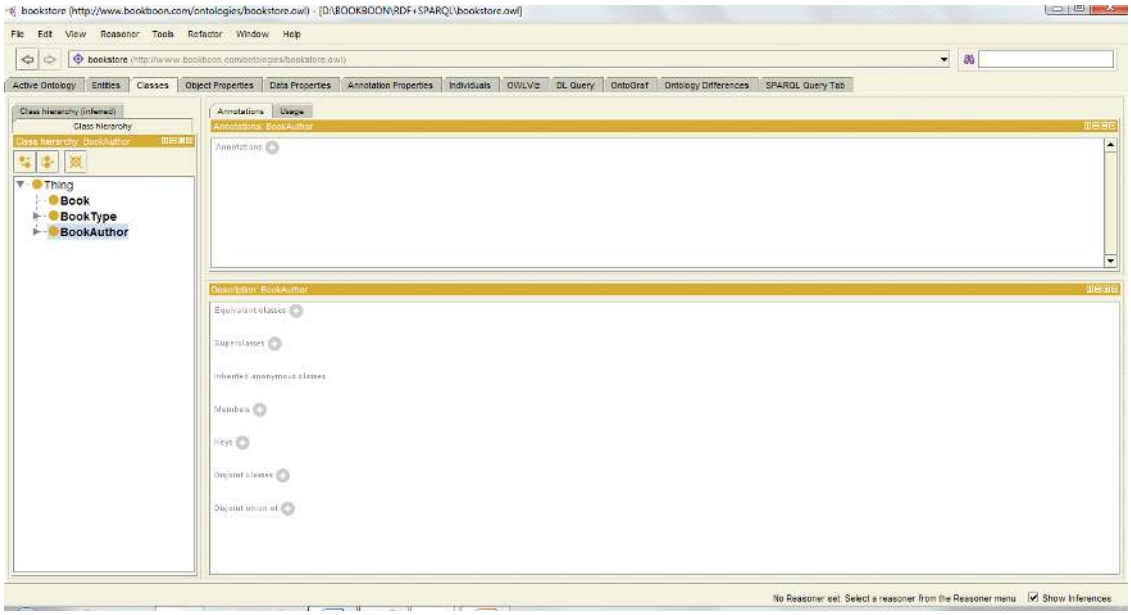
Screenshot 6.4: Protégé-classes tab

Create subclasses:

Sub-classes to be constructed are- Books, BookType and BookAuthor. They are subclasses of owl:Thing. Though there is no special naming convention, it's important that we maintain consistency. Refer to the part highlighted in the below screenshot, to know the icon used to create a sub-class.



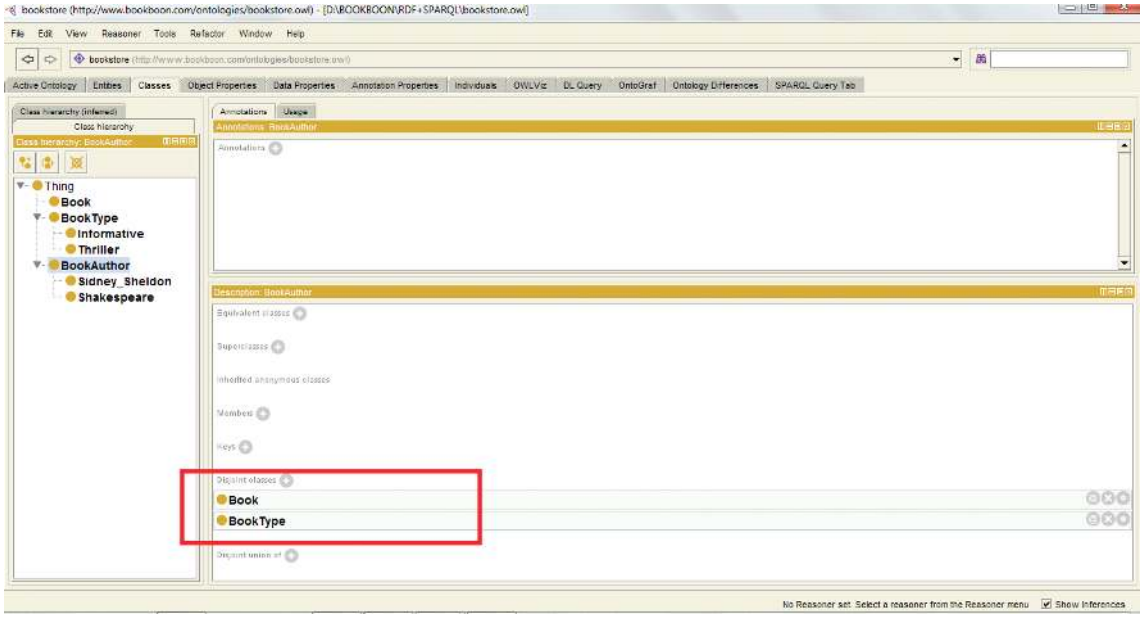
Screenshot 6.5: Creating sub-class



Screenshot 6.6: Sub-classes

Step 3:

To specify that Book, BookType and BookAuthor are disjoint classes. Add both the sub-classes to the disjoint class panel by selecting them. Refer to the below screenshot.



Screenshot 6.7: Disjoint classes

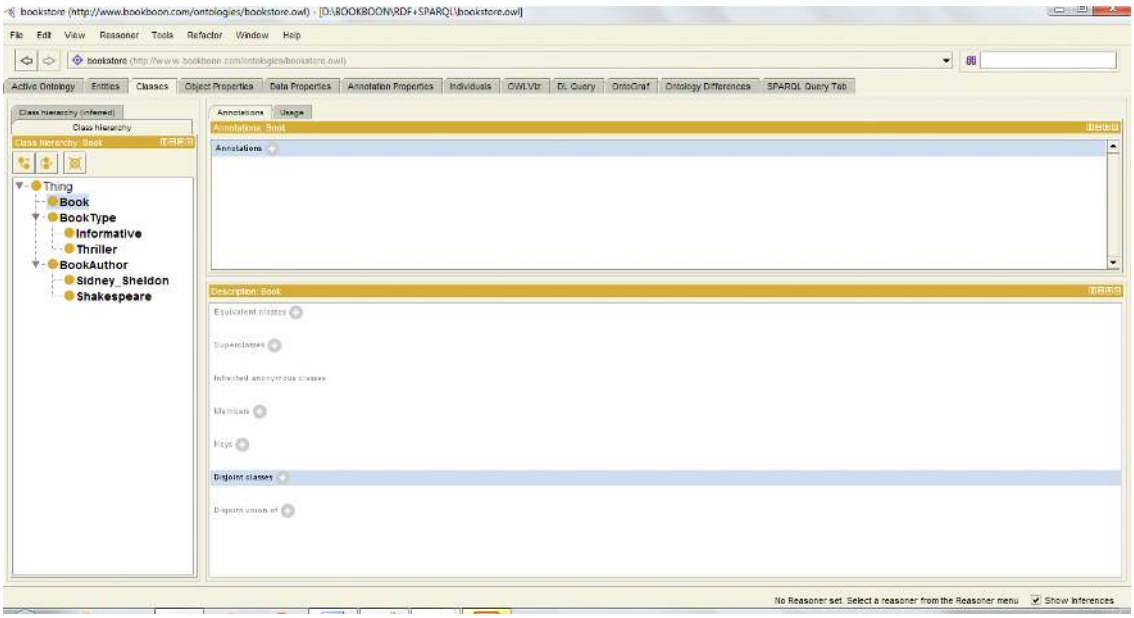
Step 4:

Create sub-classes for BookType and BookAuthor.

For example,
BookType = (Romance, Thriller)

BookAuthor = (Shakespeare, Sidney Sheldon)

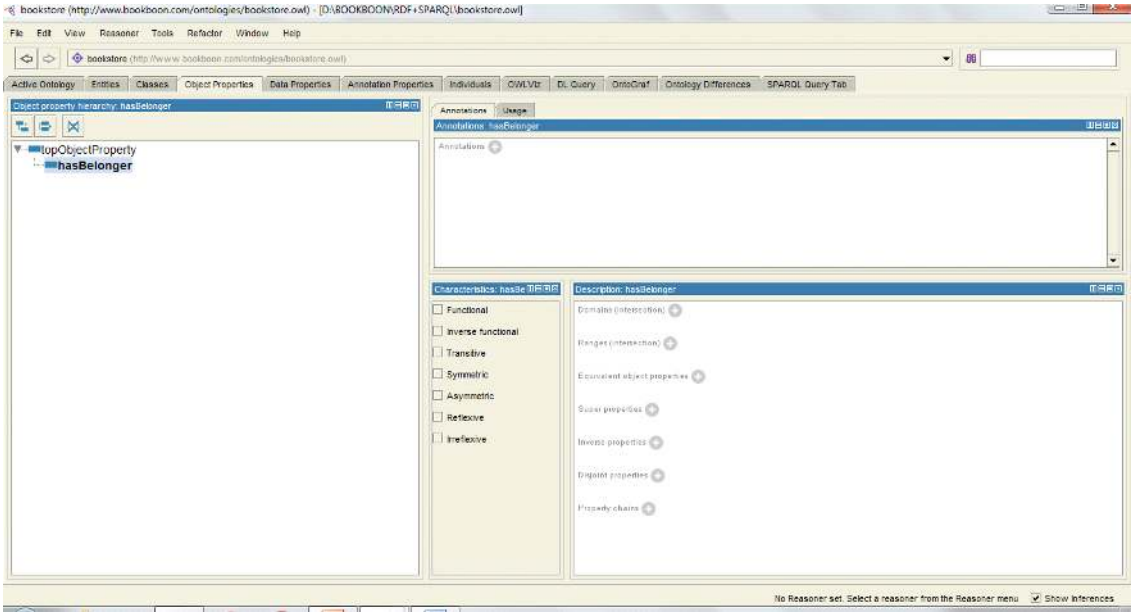
Follow the wizard to create these disjoint classes.



Screenshot 6.8 : Creating sub-classes

Step 5:

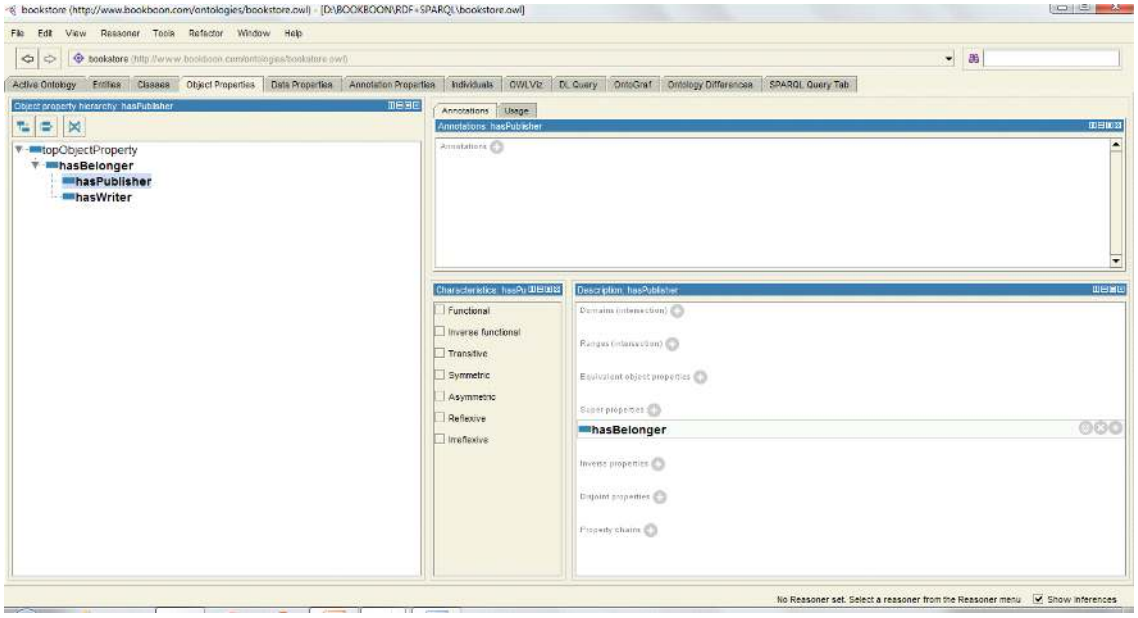
- Go to the Object properties tab.
- Click on “create Object properties”.
- Create an Object property “hasBelonger”.



Screenshot 6.8: Creating Object Properties

Step 6:

Now, select the "hasBelonger" property and add sub-properties to it. Let's create sub-properties "hasWriter" and "hasPublisher".

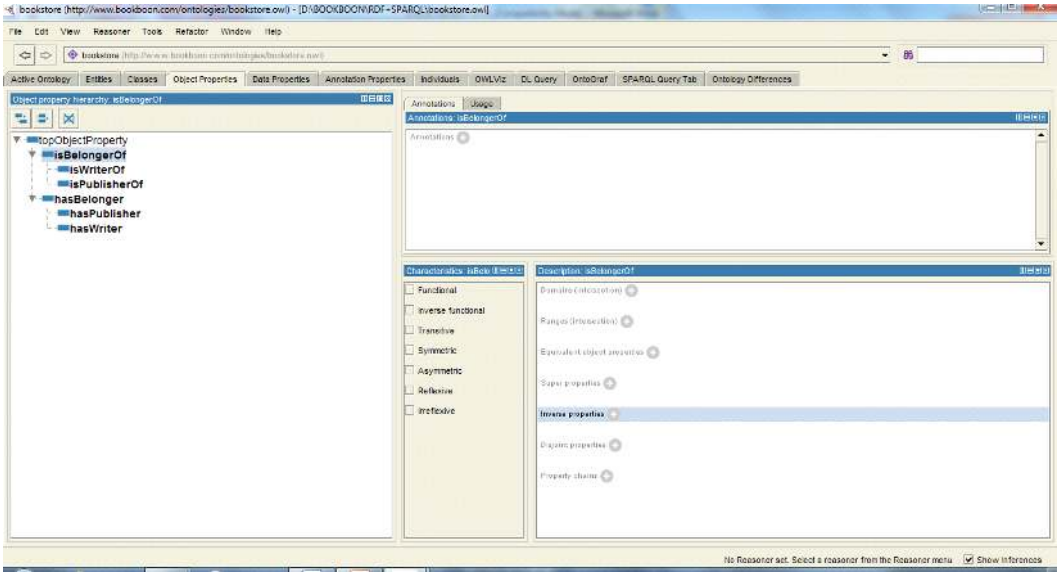


Screenshot 6.9: Creating sub-properties.

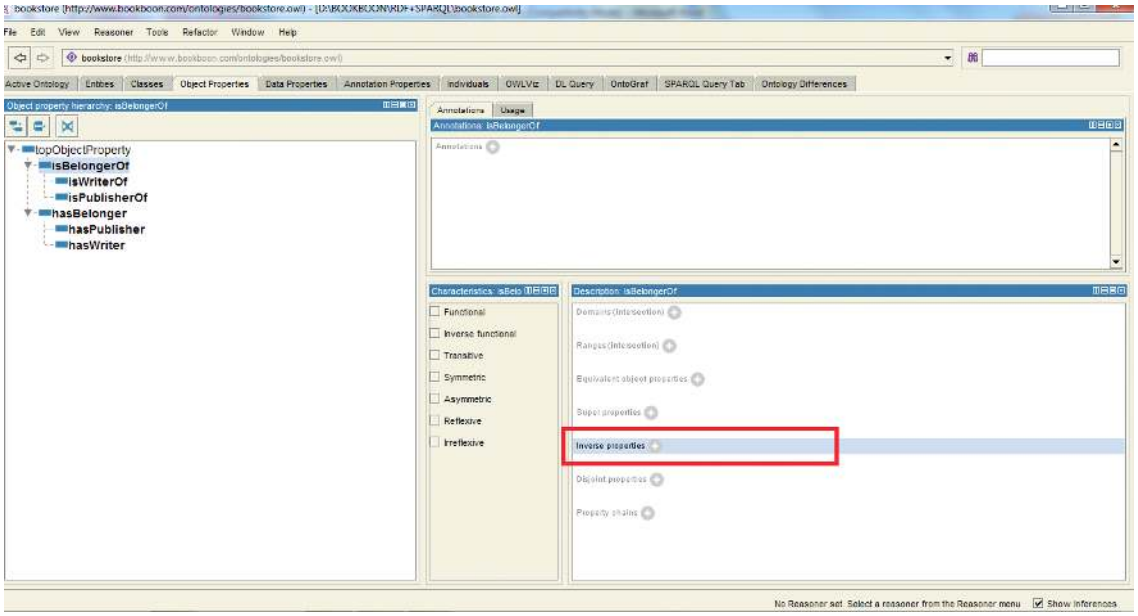
Step 7:

Create inverse properties.

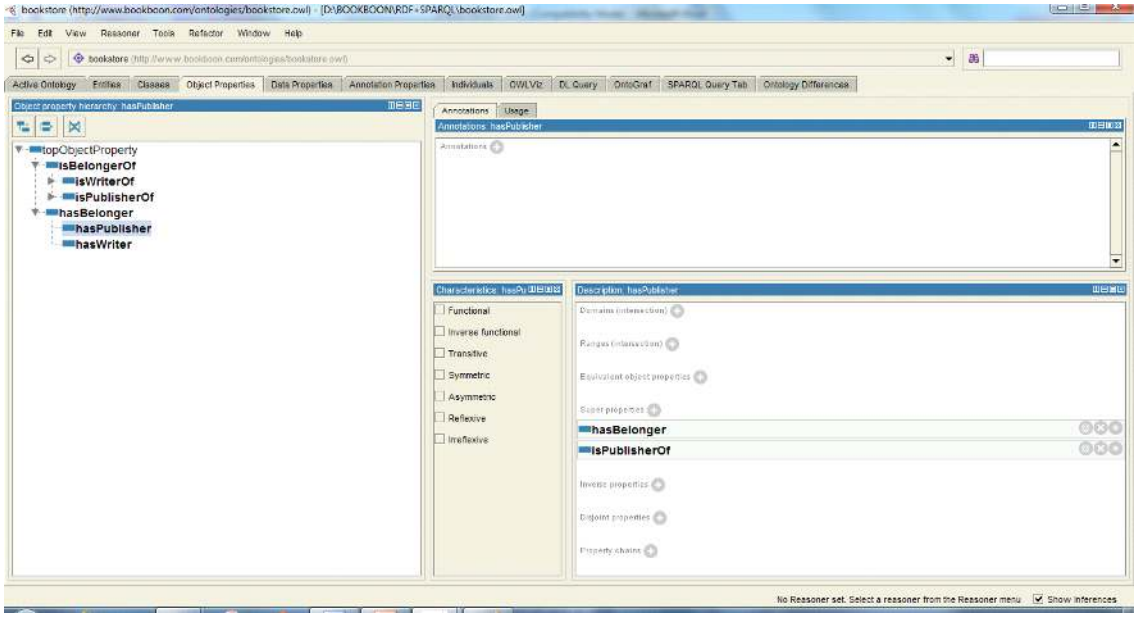
- Create a new object property called isBelongerOf.
- Press the "Set inverse property" button.
- Select "hasBelonger".
- The inverse relation has been set up.
- Select hasPublisher
- Create the isPublisherOf as the inverse property of hasPublisher.
- Hence, isPublisherOf is the subproperty of isBelongerOf.
- Select hasWriter
- Create isWriterOf as the inverse property.
- Hence, isWriterOf is the subproperty of isBelongerOf.



Screenshot 6.10: Creating inverse properties.



Screenshot 6.11: Adding inverse properties.

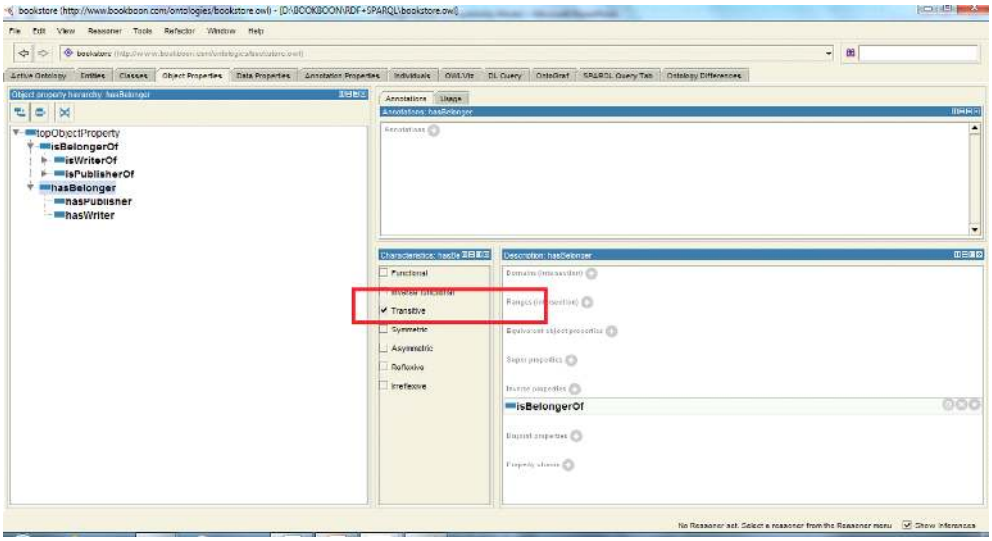


Screenshot 6.12: After completely adding inverse properties.

Step 8:

Create a transitive property.

- Select the "hasBelonger" property.
- Choose/Tick the transitive tick box.
- Select the "isBelongerOf" property, make sure that the transitive tick box is ticked.

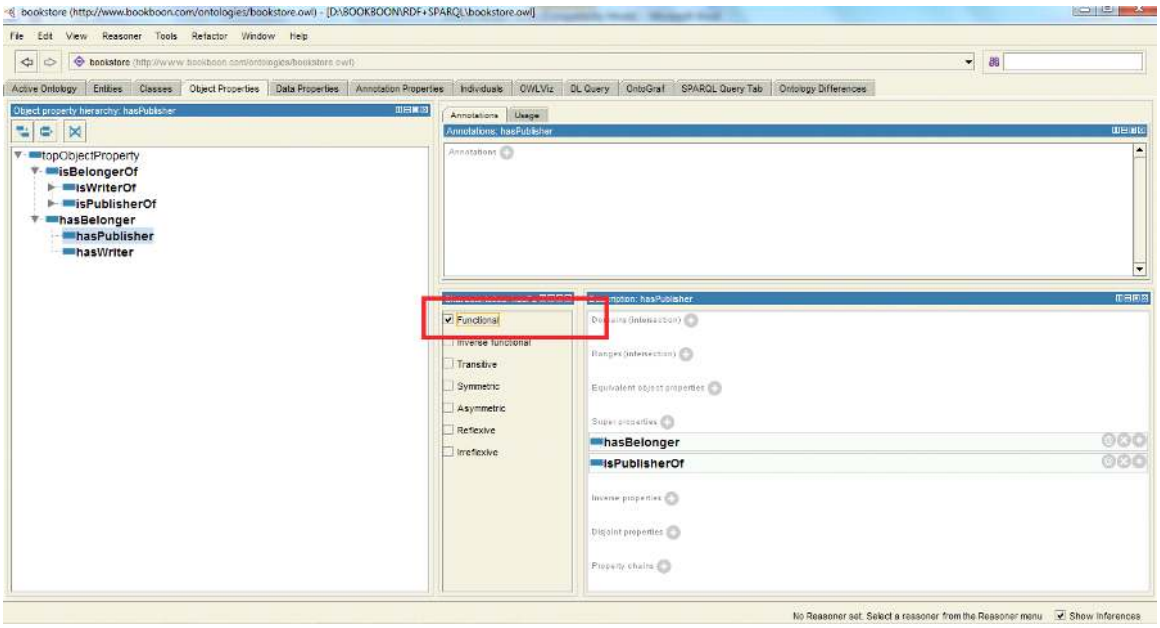


Screenshot 6.13: Creating transitive properties.

Step 9:

Create functional properties.

- Select the “hasPublisher” property
- Tick the “functional” tick box
- OWL-DL does not allow datatype properties to be transitive, symmetric or have inverse properties.

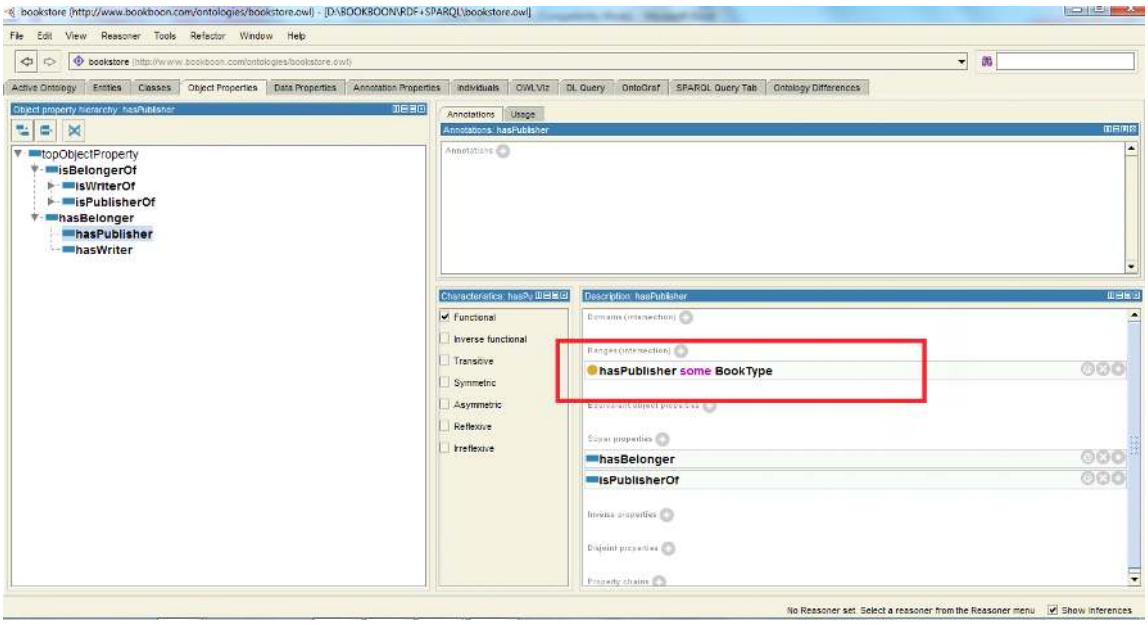


Screenshot 6.14: Creating functional properties.

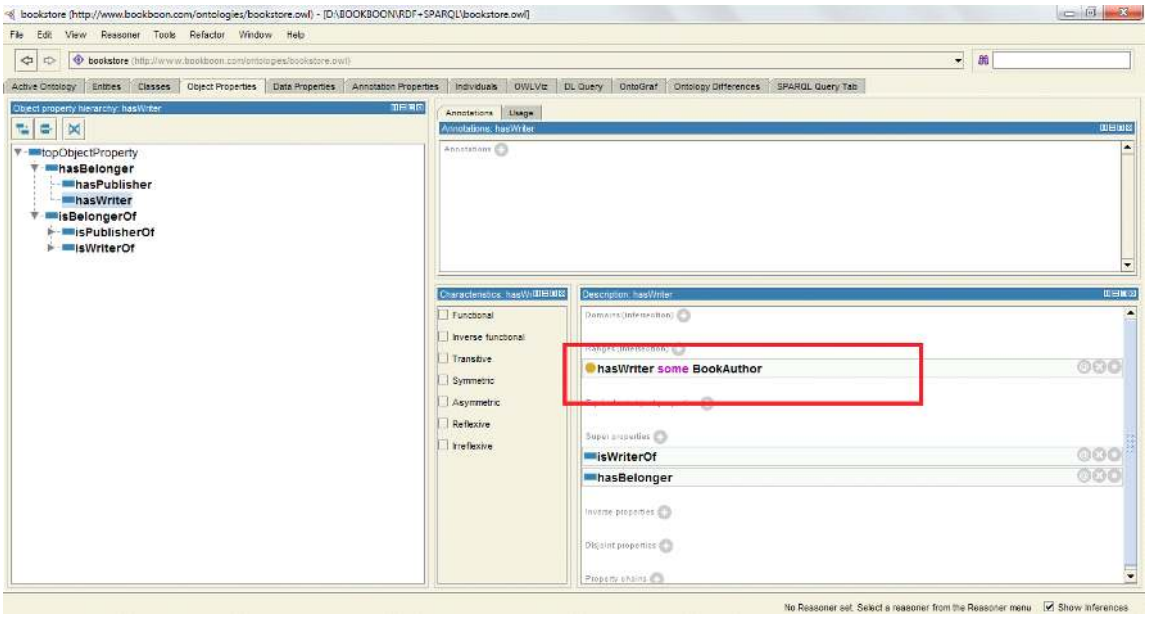
Step 10:

Specify the ranges.

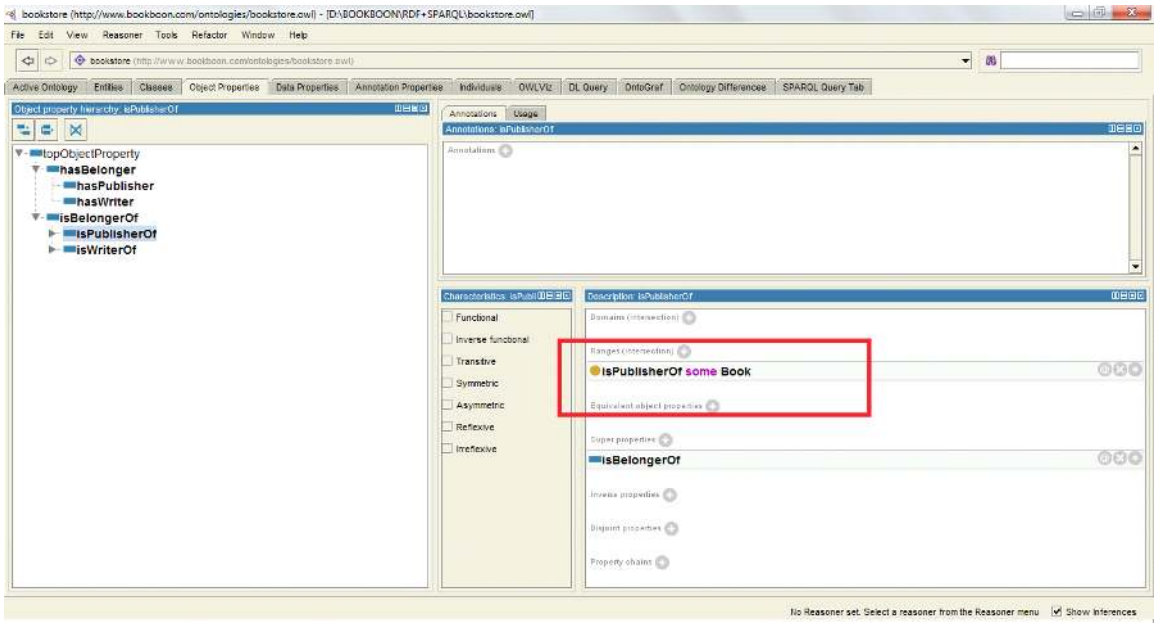
- Select hasPublisher.
- Press the range button.
- Select BookType.
- Press OK button.
- BookType must be displayed in the range list.
- When multiple classes are added to the range, they represent the union of all classes.
- Select the isPublisherOf property.
- Set the domain of the isPublisherOf property to BookType.
- Set the range of the isPublisherOf property to Book.
- The above steps will now be repeated for hasWriter.
- Select the hasWriter property.
- Specify the range as BookAuthor.
- Select the isWriterOf property.
- Specify the range as Book.



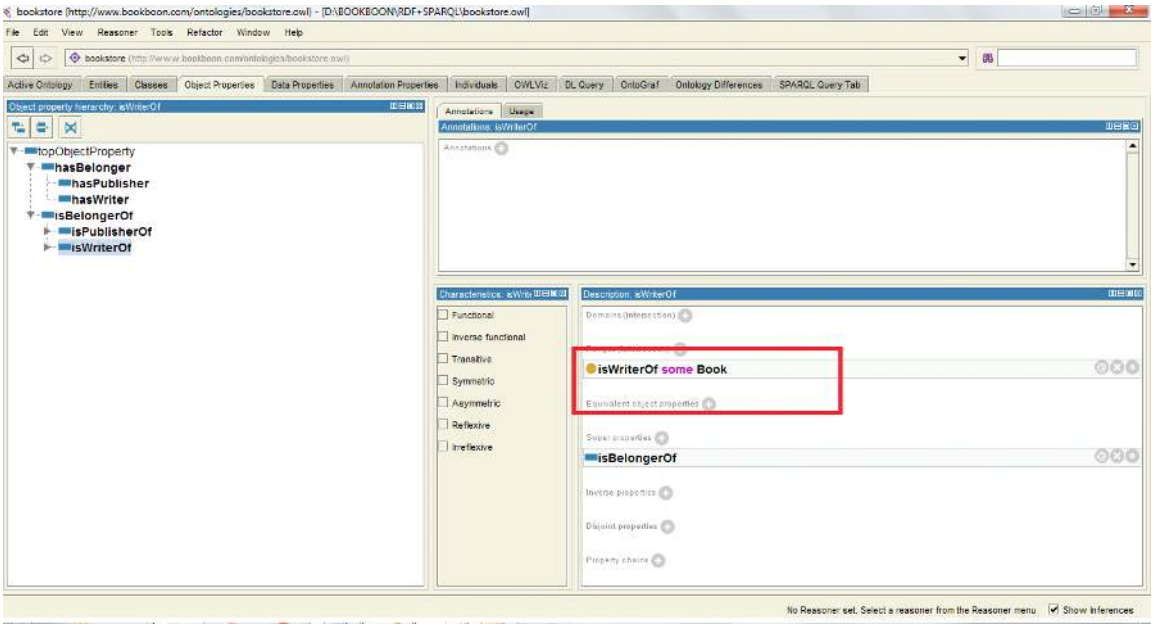
Screenshot 6.15: Specifying the ranges for hasPublisher.



Screenshot 6.16: Specifying range for hasWriter.



Screenshot 6.17: Specifying the range for isPublisherOf.

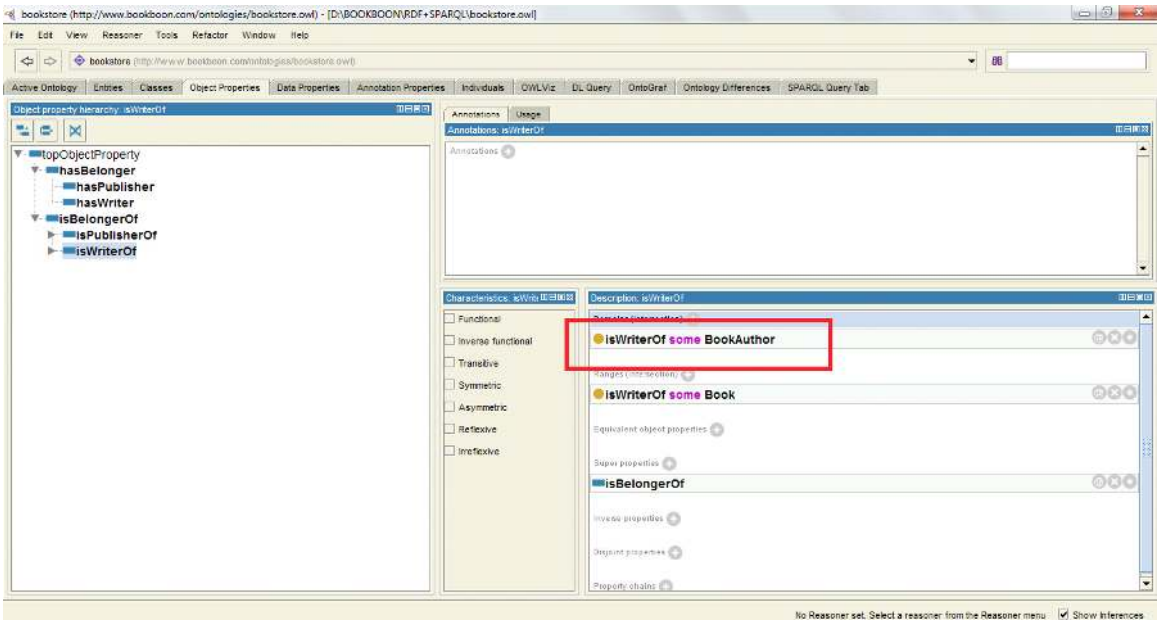


Screenshot 6.18: Specifying the range for isWriterOf.

Step 11:

Specify the domain

- Select hasPublisher property.
- Press add domain button.
- Select Book.
- Press OK.
- Book is displayed in the domain list.
- When multiple classes are added as domain, they represent the union of these classes.
- Set the domain of the isPublisherOf property to BookType.
- Select the hasWriter property.
- Specify the domain as Book.
- Select the isWriterOf property.
- Specify the domain as BookAuthor.



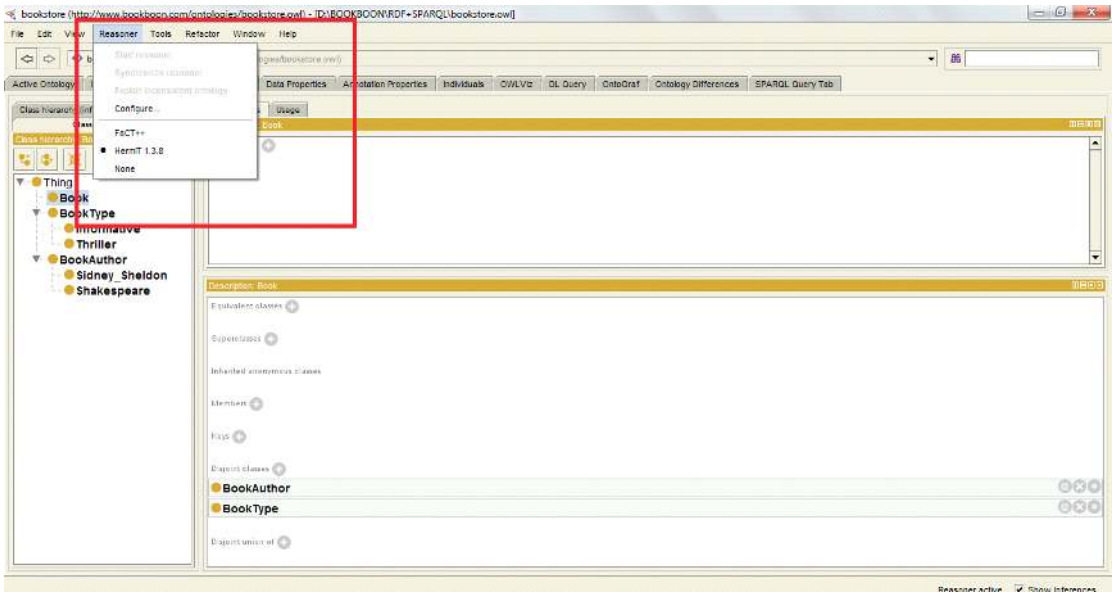
Screenshot 6.19: Specifying the domain.

Step 12:

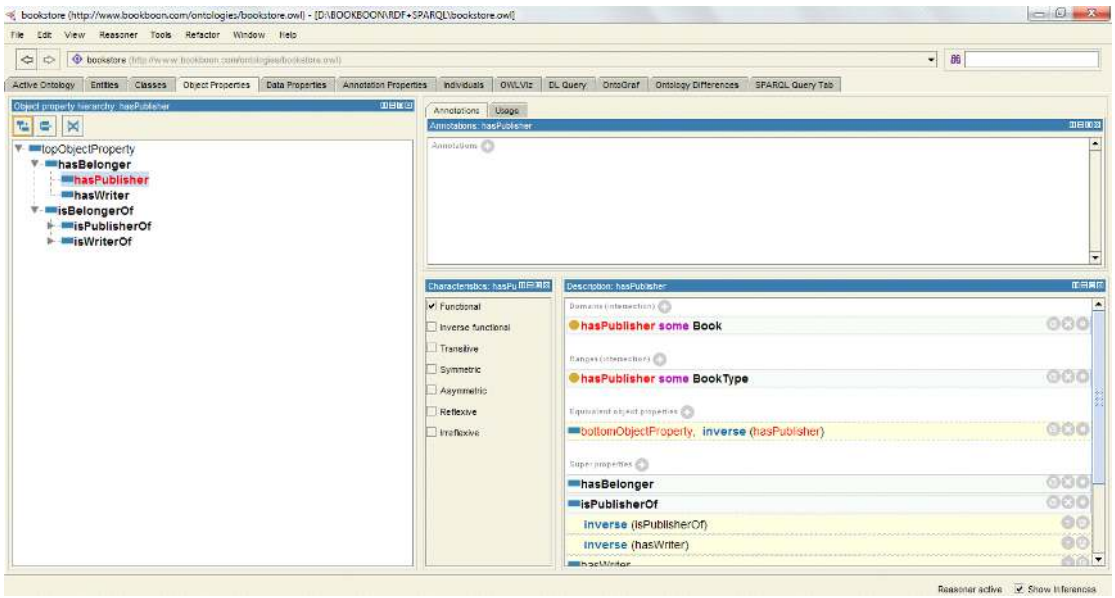
Invoke the reasoner.

Ontology described in OWL-DL can be processed by a reasoner. Go to owl —>preference, to make sure that OWL-DL is selected. The main service offered by a reasoner is to test whether or not one class is a subclass of another class. By performing such tests on all of the classes, it is possible for a reasoner to compute the inferred ontology class hierarchy.

Another reasoning service is consistency checking, i.e., to check whether or not it is possible for the class to have any instances. A class is deemed to be inconsistent if it cannot possibly have any instances.



Screenshot 6.20: Starting a reasoner.



Screenshot 6.21: Inconsistencies identified by Reasoner.

Once identified the inconsistencies are removed appropriately to ensure consistency.