.15926 Editor

Version 1.4

Volume 4

Patterns and Mapping

Welcome to the .15926 Editor. This free software is distributed by TechInvestLab.ru “as is” without any warranties or obligations, for testing purposes. Use it and enjoy at your own risk. Please send bug reports, usage and licensing questions or suggestions to dot15926@gmail.com .

February 23, 2013

Volume 4. Patterns and Mapping

Contents

Other documentation volumes:

Volume 1. Getting Started
Volume 2. APIs: Scanner and Builder
Volume 3. Extensions

License

Parts of .15926 Editor (built-in extensions and extension samples) are released as a source code under the BSD 2-Clause license.

Other parts of the software (released in binary and in text form) are not covered by the license above and are distributed "as is" and free of charge for evaluation purposes only!

Elephant icon by Martin Berube is used for .15926 software according to terms at http://www.iconarchive.com/show/animal-icons-by-martin-berube/elephant-icon.html

12. Patterns

12.1. Approach to patterns

It is well known that ISO 15926 (and Semantic Web technologies in general) allow for many ways to express the same ontological relationship between things. Part 2 type instances, base or specialised templates of unrestricted complexity, OWL relationships or custom RDF properties – all these constructs can be used in data modelling. Several implementations of ISO 15926 exist already and more are planned.

Part 2 relational instances, corresponding base templates, core, specialized and any number of custom templates can be all used to express the same fact (sometimes among other facts) – that class A is specialization of B, or individual X is part of individual Y, for example. On the other hand, the same concept of "connectedness" or "identification" can be applied to quite diverse things –to classes of classes, to classes of individuals, or to individuals – physical objects, events, activities.

Complex multi-role relationship can be expressed as custom template signature, later it can be expanded (lifted) to expose entities composing template axiom, to base template level, and finally it can be fully expanded to a complex network of Part 2 type instances. The process of template 'contraction' can go in other direction. Data graphs in this sequence can be also considered as representations of the same pattern.

Today slightly different concepts of "pattern" are developed by several ISO 15926 communities to utilize an idea of recurring patterns in various domains and of multiple methods to do the same modelling. Patterns are the cornerstone of PCA modelling methodology (https://www.posccaesar.org/wiki/FiatechJord), patterns are defined for use in mapping in ISO 15926 Information Patterns (IIP) project by members of iRING User Group (http://iringug.org/wiki/index.php?title=ISO_15926_Information_Patterns_%28IIP%29).

Pattern representation technology and tools developed in .15926 project will be able to support patterns in all possible areas: template expansion/contraction, pattern modelling, pattern mapping, etc.

For .15929 Platform pattern is a formally defined list of alternative ways (representations) to express particular ontological relation between two or more entities.

> For example, concept of connection between two things can be expressed for various types of things by usage of six Part 2 type instances (ConnectionOfIndividual, DirectConnection, ClassOfDirectConnection, ClassOfIndirectConnection, ClassOfConnectionOfIndividual, IndirectConnection), six corresponding templates from Part 8 initial template set, or through several templates currently discussed by the Special Interest Group (SIG) Modeling, Methods and Technology (MMT) of POSC Caesar Association (ClassOfDirectConnectionDefinition, etc.), available at http://15926.org). One single pattern "connected to" is based on this list. Two things are considered to form this pattern if they occupy corresponding roles in any of relational entities listed above.

Pattern recognition in data sources is a powerful mechanism of .15926 Platform. Search for patterns in SearchLan.15926 (documented in **Volume 2**) is already supported in the Editor. From version 1.4 the Editor supports mapping to patterns from Excel spreadsheet. Other pattern-based instruments will be developed later, including template expansion for data set transformations.

12.2. Patterns in the Editor

Standardization of ISO 15926 pattern format and mapping rule description language are expected in the future. Today .15926 Editor keeps patterns libraries as Python dictionaries available as open source parts of the software.

Released version of .15926 Editor has an initial set of predefined patterns defined in the library ***pattern_samples.py***. More pattern libraries will be released by TechInvestLab.ru in the future.

User can expand existing patterns and add new patterns as described below. Pattern definitions can be built using Part 2 types, template definitions and reference data items, and custom RDF predicates (properties). Pattern structure should be maintained manually now, pattern management module and GUI for pattern editing will be added in the future versions of the Editor.

Patterns libraries for the Editor are located in Python files with ***.py*** extension, placed in the <*installation_folder*>*/patterns* folder. You can add new ***.py*** files to this folder or modify existing patterns definitions.

Always use new files to record new patterns and rename *pattern_samples.py* file if you are changing definitions in it. An installer may overwrite it while upgrading the software!

Template definitions used in patterns should be present in the project to facilitate pattern search and mapping. There template definition data sources in the project should be assigned predefined specified module names, as documented in the released libraries.

Initial pattern set defined in the library ***pattern_samples.py*** depends on three template data sources:

- Part 8 initial set in a module named ***p7tpl*** (the file ***p7tpl.owl*** accompanying ISO 15926-8 is included with the distribution in folder <*samples*>);
- PCA MMT SIG set in a module named ***projtpl*** (the work-in-progress version of PCA MMT SIG set is included with the distribution as ***templates.owl*** file in folder <*samples*>*pid*);
- IIP template set in a module named ***iiptpl*** available as **IIT Sandbox (templates)** from *Files* menu.

An absence of a registered module or an absence of particular template definition in a module will lead to the inability to recognize presence of some patterns for visualization, querying or mapping.

You can edit pattern definitions without closing the Editor and reload them by File-***Reload patterns (Ctrl+Shift+W)*** menu command. This is useful while testing pattern search or mapping code.

12.3. Patterns definition

Please refer to the file ***patterns_samples.py*** in <*installation_folder*>*patterns* folder while reading this documetation.

Pattern definitions are recorded using Python dictionary and list notations (please refer to any Python tutorial for details).

In pattern format descriptions below:

- *string* value refers to Python strings included in single or double quotation marks;

- *dictionary* value refers to Python dictionaries organized as *key : value* pairs where key is a string;

- *list* value refers to Python lists;

- *element* value refers to elements defined in the project modules as an entity with module name (like **part2.Class** or **p7tpl.Classification**); element values can be written without quotation marks.

12.3.1. Overall pattern structure

Pattern definitions are stored as the Python list in the local variable **patterns**. Use **patterns.append** method to add new patterns to this list.

Each pattern definition in a list is a dictionary with the following predefined keys with values:

key **'name'** with string value – required name of the pattern, any text string without spaces (use CamelCase or underscores);

key **'signature'** with dictionary value – dictionary with *pattern role* names as keys and string description of "inverse" roles as values;

key **'options'** with list value – list of pattern option definitions (pattern building blocks representing alternative sets of entities realizing the pattern).

12.3.2. Pattern roles

Pattern signature is a dictionary containing ***pattern role*** names and textual description of inverse relations corresponding to roles, for use in output forms. Role names are used as keys, textual descriptions as values. To get a sensible output the textual description has to describe relation of other objects to an object in this role, for example:

'subclass': 'is superclass of'
'identifier': 'is identified by'

where **subclass** and **identifier** are ***pattern roles***.

Textual description can be empty: **''**.

12.3.3. Pattern options

Each item in pattern's **options** list contains definition of one specific way the pattern is realized (one particular way the pattern can be represented in dataset). Each option is independent from others. Pattern search can look for all options or for one option particularly (if it is named).

Each option is defined by a Python dictionary with the following keys and values:

Pedefined option description keys:

key **'name'** with string value – optional name of the option, any text string without spaces (use CamelCase or underscores), may be omitted;

key **'expansion'** with string value – optional name of another option of the same pattern which is an expansion for the described option (as axiom for the template signature, or fully expanded template for template axiom);

key **'parts'** with list value – optional list of dictionaries describing individual parts comprising pattern option.

If an option consists of only one part, the dictionary defining this option directly contains ***part description keys*** described below. If there is more then one part in an option, special key **'parts'** with list value is used on option dictionary. The **'parts'** list members are dictionaries describing individual parts with ***part description keys***.

If **'parts'** list is used to define option parts, only keys **'name'**, **'parts'** and **'expansion'** are allowed in the option definition dictionary. Keys **'type'** and **'uri',** ***pattern role*** keys and ***scolem variable*** keys are used in option part definition. Part definition can not have **'name'**, **'parts'** and **'expansion'** keys.

Parts of a pattern option should be interconnected by the use of common pattern roles or common Scolem variables (see below). If pattern is defined in such a way that it can be separated into two independent components – such pattern will be ignored.

Part description keys include ***predefined keys, pattern role keys*** and ***scolem variable keys.***

Predefined part description keys (either 'type or 'uri' key is allowed!):

key **'type'** with element, string or list value – specifying type or classifier class (several classifier classes) for an entity which is a part in focus; classifier may be any Part 2 type or reference data class;

key **'uri'** with string or list value – restricting an entity which is a part in focus to one particular entity or a set of entities.

Pattern role keys with string values – defining mapping of pattern roles to relations properties of option parts.

Scolem variable keys with string values – defining mapping of scolem variables to option parts and their properties.

All keys whose names are not predefined keys or ***pattern role*** keys (defined in the pattern signature) are treated as ***scolem variable*** keys, whose function is to show that the same entity is found to be related in a specified way to some parts of a single option (described as **'parts'**).

Values for **'type'** key can be URI strings, lists of URI strings, or elements (defined in a particular module). If referenced module is not loaded, any option referencing it will be ignored. ***Pattern role*** keys can be also mapped to custom object and annotation properties (*Roles* and *Annotations* defined for a project or a data source) of option parts.

Inheritance reasoning is implemented for searches with pattern with **'type'** key in one or more parts. Trying to find entities restricted by **'type'** key in pattern searches Scanner module will:

- identify classifiers of an entity using all available options of **Classification pattern**;
- identify all superclasses of classes defined at previous step using all available options of **Specialization pattern**.

Values of ***pattern role*** keys and ***scolem variable*** keys can be the following (all represented by strings):

- a role of a relational entity type or of a template in a module;

- an URI of an object or literal property;

- a short name of a custom object or of an annotation property (defined in *Roles* and *Annotations* of a project or of a data source);

- a special value **'self'** signifying that an entity defined in the option part is itself occupying the pattern role or is a value of ***scolem variable***.

12.4. Examples

12.4.1. Simple pattern example

Let's make a pattern for various ways to record subclass-superclass relationship between classes. Start with a blank pattern stencil:

```
patterns.append({
        'name': 'Specialization',
        'signature': { },
        'options': [ ]})
```

We'll call the pattern **Specialization** and define signature with two roles named **subclass and superclass**:

```
'signature': {'subclass': '', 'superclass': ''},
```

If we've found **subclass** for some entity, this entity become superclass of the found entity, and vice versa. So we'll formulate descriptions of "inverse" roles as:

```
'signature': {'subclass': 'is superclass of', 'superclass': 'is subclass of'},
```

We'll define three alternative representations or options by which this pattern can be realized in the data set (of course there are more then three ways to record a fact of subclass-superclass relationship).

The first option to record specialization is to use an instance of base template **p7tpl.Specialization**, defined in ISO 15926-7. We'll call this option **SpecTemplate**. Option definition is very simple. This option description consists of one part (so no **'parts'** key used) – entity-in-focus is template instance, and pattern roles are directly mapped to corresponding roles of this template:

```
{'name' : 'SpecTemplate',
   'type':       p7tpl.Specialization,
   'subclass':    'hasSubclass',
   'superclass':   'hasSuperclass',},
```

The second option to record specialization is on Part 2 level, using instance of **Specialization** type linked to classes in question. This option description also has only one part – entity-in-focus is Part 2 type instance, and pattern roles are directly mapped to corresponding roles of this entity:

```
{'name' : '',
   'type': part2.Specialization,
   'subclass'   : 'hasSubclass',
   'superclass' : 'hasSuperclass'},
```

Notice that what we've described as a second option is the representation of a FOL axiom of **p7tpl.Specialization** template of the first option.

Generally, once we have a pattern defined with a template, we can automatically generate additional option in this pattern definition using a FOL axiom of this template. Such functionality will be added to the Editor in future versions.

Let's make the name of a second option **SpecTemplAxiom** and add key-value pair to the first option:

```
'expansion' : 'SpecTemplAxiom',
```

which will work as a link from template signature to its expansion.

The third option will use an OWL property instead of reified relationship. As we've discussed above, one possible way to record class specialization relationship is usage of **owl:subClassOf** object property. Corresponding option description looks like:

```
{'name' : 'SubClProperty',
   'subclass'   : 'self',
   'superclass' : 'http://www.w3.org/2000/01/rdf-schema#subClassOf'}
```

This pattern option is found if some entity has **http://www.w3.org/2000/01/rdf-schema#subClassOf** property connecting it to some other entity. Pattern role mapping defines that in this case property possessor itself is **subclass** and property value is **superclass**.

Now let's take all **options** together:

```
patterns.append({
    'name': 'Specialization',
    'signature': {'subclass': 'is superclass of', 'superclass': 'is subclass of'},
    'options': [
            {'name' : 'SpecTemplate',
               'type':       p7tpl.Specialization,
               'subclass':   'hasSubclass',
               'superclass':  'hasSuperclass',
               'expansion':      'SpecTemplAxiom',},
            {'name' : 'SpecTemplAxiom',
               'type': part2.Specialization,
               'subclass'   : 'hasSubclass',
               'superclass' : 'hasSuperclass'},
            {'name' : 'SubClProperty',
               'subclass'   : 'self',
               'superclass' : 'http://www.w3.org/2000/01/rdf-schema#subClassOf'},]})
```

12.4.2. Scolem variable example

Look at a pattern option (from **PropertyOfClass** pattern) illustrating the use of Scolem variable in a pattern option with two **'parts'** elements:

```
{'name'  : 'PropertRangeRestrictionOfClassTemplateAxiom',
'parts' : [ {'variable1' : 'self',
        'type' : part2.ClassOfIndirectProperty,
        'range' : 'hasPropertySpace',
        'posessor' : 'hasClassOfPossessor'},
        {'type' : part2.Specialization,
        'variable1' : 'hasSubclass',
        'property' : 'hasSuperclass'}, ]},
```

This option description consists of two parts, with instances of Part 2 types **ClassOfIndirectProperty** and **Specialization** as entities-in-focus. To join role occupiers of these two entities in a pattern an entity-in-focus of a first part **variable1** (not listed in the pattern signature) is mapped to the **hasSubclass** role in a second part which is a **Specialization** instance.

12.4.3. URI usage example

Look at a pattern example illustrating the use of a particular RDL entity in a pattern:

```
{'name' : 'SpecializationFromPUMP',
    'signature' : {'subclass' : ''},
    'options' : [
      {'parts' : [
            {'PUMP': 'self',
            'uri' : 'http://posccaesar.org/rdl/RDS327239'},
            {'type' : part2.Specialization,
            'subclass' : 'hasSubclass',
            'PUMP' : 'hasSuperclass'} ] },
        [   {'PUMP': 'self',
            'uri' : 'http://posccaesar.org/rdl/RDS327239'},
            {'subclass' : self,
            'PUMP' : 'http://www.w3.org/2000/01/rdf-schema#subClassOf'} ] },
      ]}
```

This pattern will be identified for all entities which are specialized from PUMP class (http://posccaesar.org/rdl/RDS327239).

There are two parts in the first option of this pattern. One part is class with URI http://posccaesar.org/rdl/RDS327239 representing PUMP. Another part is an instance of **Specialization** relationship, in which **hasSuperclass** role is mapped to the PUMP class, and occupier of another role is an entity for which this pattern will be identified.

The second option of this pattern has two parts also. The first part is the same PUMP class, and the second part describes that an entity in focus has PUMP class as a value for *rdfs:subClassOf* property.

12.4.4. Class restriction example

Look at a pattern example illustrating the use of a particular RDL entity as classifier restriction in a pattern:

```
{'name' : 'ClassifiedByPUMP',
   'signature' : {
     'classified' : '',
     },
   'options' : [
     {'parts' :  [{'classified' : 'self',
                   'type' : 'http://posccaesar.org/rdl/RDS327239'}, ] },
  }
```

This pattern will be identified for all entities which are members of PUMP class (http://posccaesar.org/rdl/RDS327239). There are no such entities in the PCA RDL, use data file ***sample_lookup.rdf*** from <*samples*>\ folder. Execute query:

show(type=patterns.ClassifiedByPUMP, classified=out)

The search for this pattern will use inheritance rules implemented for pattern search and described above – it will return entities classified by PUMP class, and entities classified by subclasses of Pump.

12.4.4. IIP pattern modelling

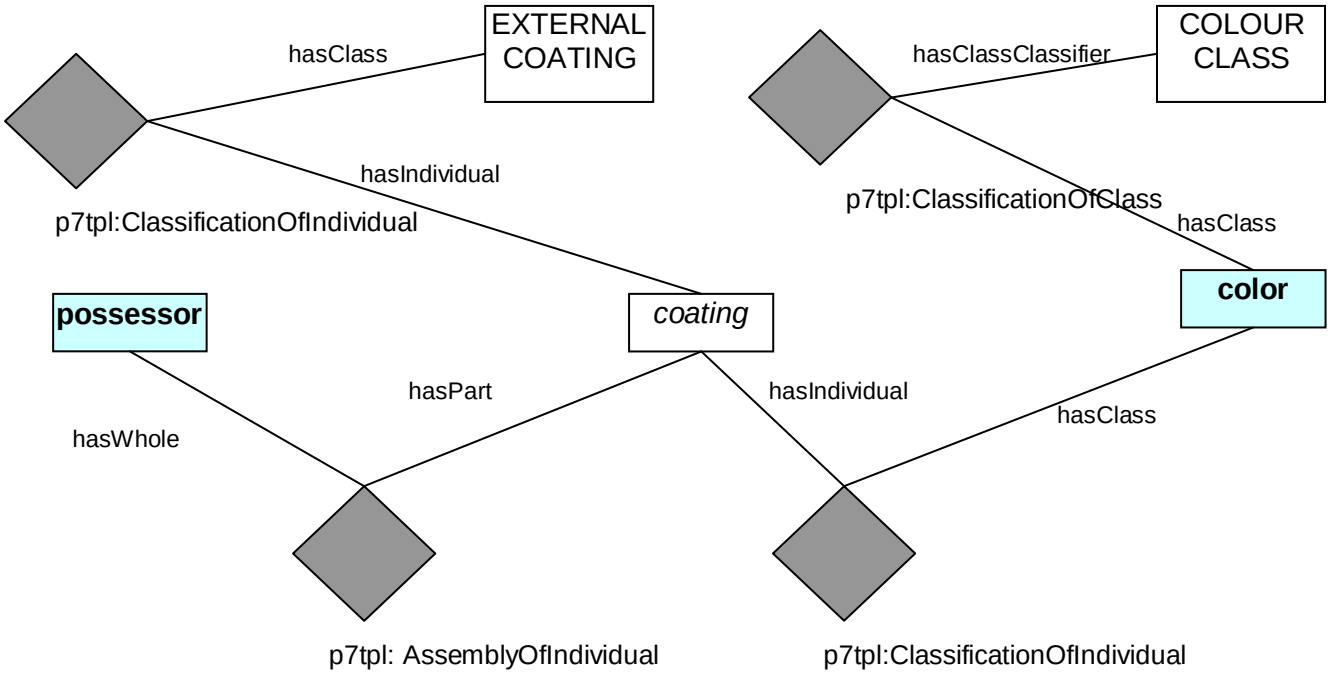
Combining methods of pattern construction described above, let's model a pattern representing one of the patterns proposed in IIP project of iRING User Group (http://iringug.org/wiki/index.php?title=ISO_15926_Information_Patterns_%28IIP%29). Look at the Coating Color pattern described as:

P0003	0	Coating Color	The color name for the coating	EXTERNAL COATING	AssemblyOfIndividual
	1			COLOUR CLASS	ClassificationOfIndividual

We can use this pattern to identify relationships in ISO 15926 dataset for import into some engineering database which has a color list for objects. Or we can use it to create ISO 15926 data set as export from such database. Depending on the goal we'll use pattern search functionality of Scanner.15926 (available already), or pattern writing functionality of Builder.15926 (see spreadsheet adaptor description below).

To identify (or create) this pattern for an entity-in-focus *possessor* (individual), we've to find (or create) another individual (*coating*) classified by EXTERNAL COATING class and connected to our entity-in-focus by an instance of AssemblyOfIndividual template, and then look for (or create) classification of *coating* individual by a class which is a member of COLOUR CLASS.

Selection of templates for classification has to be agreed for pattern modeling, in this example we'll use base templates only, specialized to the most specific entity types available (thus ClassificationOfClass and ClassificationOfIndividual, not Classification).



We'll have two roles for our pattern: **possessor** and **color**. Other parts of a pattern will be representing two classifier classes and four template instances connecting them in a pattern.

If you look at other parameters of Coating Color pattern you will notice that it is applicable to restricted list of several commodity types: EQUIPMENT, INSTRUMENT, PIPING NETWORK SYSTEM, PIPING NETWORK SEGMENT, PIPING SPECIALTY COMPONENT, PIPE SPOOL and VALVE. These are restrictions on the type of **possessor** role, and are given as a list of restricting classes with the **'type'** key. Some of the restricting classes exist in the PCA RDL, others can be found only in the iRING Sandbox, this a mix of URIs in two different namespaces.

The pattern is:

```
{'name' : 'P0003CoatingColor',
    'signature' : {
      'possessor' : 'is coating color of',
      'color' : 'has coating color',},
    'options' : [
      {'parts' : [
            {'posessor': 'self',
       'type' : ['http://rdl.rdlfacade.org/data#R98505918404',
       'http://posccaesar.org/rdl/RDS292589',
       'http://rdl.rdlfacade.org/data#Rd9c631e5-543f-4b98-8684-901e710f953f',
       'http://posccaesar.org/rdl/RDS270359',
       'http://posccaesar.org/rdl/RDS267704',
       'http://rdl.rdlfacade.org/data#R27982435055',
       'http://rdl.rdlfacade.org/data#R85188970528']},

          {'colorclass': self,
          'uri' : 'http://posccaesar.org/rdl/RDS38249'},

          {'type' : p7tpl.ClassificationOfClass,
          'color' : 'hasClass',
          'colorclass' : 'hasClassClassifier'},
```

```
            {'extcoatclass': 'self',
            'uri' : 'http://posccaesar.org/rdl/RDS736495131'},

            {'type' : p7tpl.ClassificationOfIndividual,
            'extcoatclass' : 'hasClass',
            'coating' : 'hasIndividual'},

            {'type' : p7tpl.ClassificationOfIndividual,
            'color' : 'hasClass',
            'coating' : 'hasIndividual'},

            {'type' : p7tpl.AssemblyOfIndividual,
            'possessor' : 'hasWhole',
            'coating' : 'hasPart'},
            ] },
        ]}
```

The same pattern can be represented in other form, using IIP templates and using more general 'type' restriction instead of Classification templates. This example uses only RDS/WIP URIs.

```
{
'name': 'CoatingColor',
'signature': { 'possessor' : 'is coating color of',
               'color' : 'has coating color'},
'options': [{'parts':
              [{'possessor': 'self',
                'type': ['http://rdl.rdlfacade.org/data#R98505918404',
                         'http://rdl.rdlfacade.org/data#R97295617945',
                        'http://rdl.rdlfacade.org/data#Rd9c631e5-543f-4b98-8684-901e710f953f',
                         'http://rdl.rdlfacade.org/data#R19192462550',
                         'http://rdl.rdlfacade.org/data#R58701958436',
                         'http://rdl.rdlfacade.org/data#R27982435055',
                         'http://rdl.rdlfacade.org/data#R85188970528']},

               {'coating': 'hasPart',
                'possessor': 'hasWhole',
                'type': iiptpl.AssemblyOfIndividualTemplate},

                {'color': 'self',
                'type': 'http://rdl.rdlfacade.org/data#R28965418667'},

               {'coating': 'self',
                'type': 'http://rdl.rdlfacade.org/data#R22385076353'},


                {'color': 'hasClass',
                 'coating': 'hasIndividual',
                 'type': iiptpl.ClassificationOfIndividual}]
}],
}
```

Always use new files to record your patterns and rename *pattern_defs.py* file if you are changing definitions in it. An installer may overwrite it while upgrading the software!

13. Spreadsheet mapper

Spreadsheet mapper is a powerful adapter designed for the transformation of Excel spreadsheets to ISO 15926 compliant RDF.

To get an ISO 15926 representation of a spreadsheet we need a data model of the spreadsheet. This data model should be is described by the pattern in one of the pattern libraries of the Editor, using reference data and templates from some RDL(s). The mapping is defined between pattern roles and spreadsheet columns. Then the adapter imports spreadsheet data into the local data source in RDF format.

Spreadsheet mapper is included into the .15926 Editor as an open source extension released under the **BSD 2-Clause License**. **Other parts of the software (released in binary and in text form) are not covered by this license.**

13.1. Mapping basics

The adapter works only with Microsoft Excel installed and running on the computer, using Excel VBA access. It is tested with Microsoft Excel starting from 2002 version.

The adapter can work with files in *.xls* and *.xlsx* formats only. The adapter uses many specific features of these formats, storing mapping data both in open and hidden data fields and attributes of the spreadsheet. Saving spreadsheet in other formats will most probable destroy the mapping data and will require a new mapping process.

Spreadsheet adapter allows mapping of only one pattern for a worksheet at a time! You can save mappings to files and load them from files to make several imports from one worksheet sequentially with different patterns, but only the last mapping will be stored in the worksheet itself.

The adapter is called via *Import – Build patterns from Excel* menu command. Make sure that spreadsheet for data import is open in Excel before you call the adapter! The spreadsheet should have all the needed worksheets present and named and all necessary column names should be already defined in Row 1 of each worksheet. Column names should start from a letter or a numbers, avoid use of other symbols as mapper will add more columns to your spreadsheet with names starting with "!" and "$".

If you find out that you need to insert a worksheet, rename it or insert more columns for your data – you have to close the adapter panel, make necessary changes and call adapter again, in some cases you'll be obliged to redefine the mapping after that.

You can add or change data in the spreadsheet, change mapping and repeat import, search data source and browse other data sources in the Editor – all without closing the adapter panel.

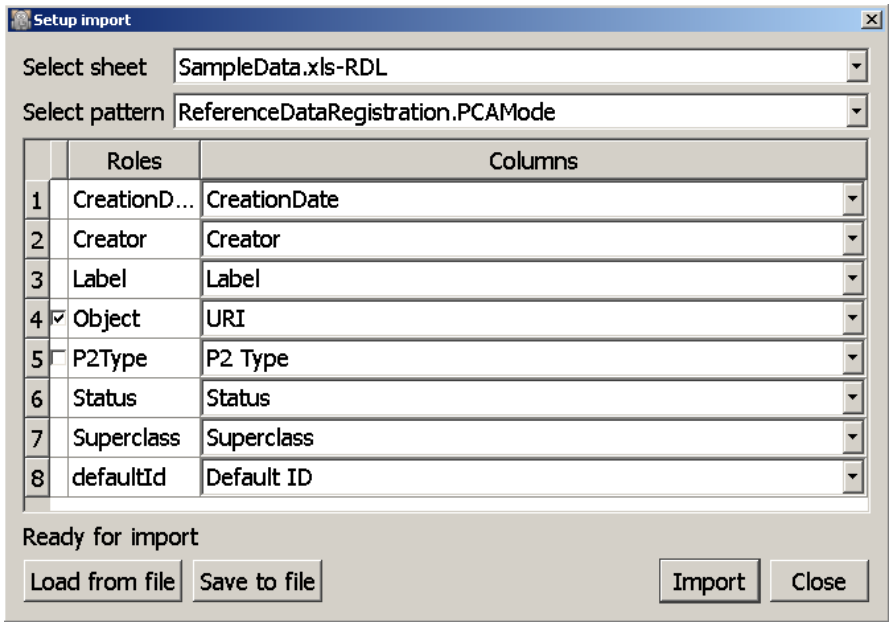
Previously imported data may be edited or changed when the import is repeated, but never removed. Changes to the data source incurred by the spreadsheet import can not be undone via Editor's *Undo*.

The mapping and some overhead information (classifiers, generated URIs for pattern Scolem variables, etc.) are stored in the open and in the hidden data fields and attributes of the spreadsheet. Do not change any fields except the ones you've designed for information entry, and the adapter will open your worksheets again preserving all previously defined mappings.

13.2. Mapping and import

Select the worksheet and the pattern in the mapping panel to start the mapping. You can select any worksheet of any file open on your computer at the time adapter is called. You can select any pattern option of any pattern in pattern libraries present in the Editor. Patterns are listed alphabetically with option names (unnamed options are numbered sequentially). Always name pattern options designed for mapping!

Once worksheet and pattern are selected, you will be presented with a list of pattern roles (left column) and will be able to choose corresponding worksheet column names (right column) to map them.



Data transformation during the import depends in the mapping, on the content of the imported spreadsheet and on the data contained in the project data sources. Please pay attention to the rules describing data changes for incremental entry of data and sequential imports of the same spreadsheet!

Mapping and import rules:

1. You can leave a pattern role unmapped by leaving column selection field empty. Unmapped roles will be ignored. If unmapped role corresponds to some pattern part or to the role of some pattern part – corresponding entity will be omitted, and entities linked to it may be omitted also.

2. Sequential imports can not delete any previously imported entity, property or relationship in your data source, whatever changes are made to the spreadsheet. You can edit any imported property or relationship; edited information will replace the old one.

3. All pattern roles defined with **'self'** value in corresponding pattern parts represent entities which potentially may be created by the adapter. Such roles have a checkbox before their names.

3.1. Mark this checkbox if you want to create new data entities for this part of a pattern. The cells in the column mapped to the marked pattern role may be left empty, may contain special value ***new*** or contain text which will be treated as an URI.

- If the cell is empty – corresponding entity will be not be created for this row. If according to the pattern structure it should occupy a mandatory role in some relational entity of a known type (Part 2 type instance or a template instance) – this one will be skipped in turn, and entities linked to it may be skipped also.

- If the cell contains special value ***new*** – new entity will be created in the data source. Its URI will generated in the *Namespace for new entities* of the data source according to the rules described in **Volume 1. Getting started**.

- If the cell contains any other text – it will be interpreted as an URI.

  If an entity with this URI already exists in the data source – data for it will be added. All changed properties and relationships will replace the old ones, but there is no way to delete properties or relationships by changing the spreadsheet.

  If there is no entity with this URI in the data source – it will be created with this URI. The adapter will not check whether URI is well-formed or not!

3.2. If checkbox at the role name is not marked, the adapter will not create new entity for the corresponding part of the pattern. If this part of the pattern is restricted by a **'type'** key – the adapter will attempt to build a full classifier for this restriction by searching for all occurrences of **Classification** pattern in **all** project data sources. The set of classes identified in this process will be exported to the spreadsheet and built into the drop-down list for all cells in the column.

If such classifier is present as a part in the pattern, try to import an empty worksheet immediately after you've defined the mapping. It will create drop-down menus for all cells of the corresponding columns which will simplify further manual data entry.

4. The cells in all other mapped spreadsheet columns may be left empty, or may contain strings, including labels of some reference data entity or URIs.

4.1. If the column is mapped to the pattern role which represents annotation property:

- If the cell is empty – the property will be omitted. If such property for the entity already exists – it will be left intact.

- If the cell contains any string – the property will be created with the specified value.

4.2. If the column is mapped to the pattern role which represents relational entity role or other object property, values of its cells will be interpreted as references to entities in the project. The adapter will create relational entity of a known type (Part 2 type instance or a template instance) only if all its mandatory roles are successfully filled by the entities imported from spreadsheet according to the mapping. The adapter will always create any other object property with specified value.

To find a referenced entity for the role (property) the adapter will:

- Attempt to search for an entity with exactly the same ***rdfs:label*** property. If search brings such an entity – it will be placed in the referred role.

- If search brings nothing – adapter will use a value in the cell as an URI of an entity without checking whether it is well-formed or not.

5. Entities corresponding to the parts of the pattern which represent Scolem variables (parts which are not present in the signature) will be always created at import, except for the following cases:

- When they are unambiguously identified by **'uri'** key with a single value. Such entities will be used to populate relations to which they are assigned, but entities themselves will not be created in the data source.

- When one or more of mandatory roles of a relational entity (Part 2 type instance or a template instance) can not be filled after iterative attempts to do data transformation for the worksheet.

6. For partially filled worksheet the adapter will iterate data transformation process attempting to create as many entities as possible.

7. The adapter is creating parts of the pattern in the order they are listed in the pattern definition dictionary. It is advisable to arrange pattern parts in the definition dictionary in such a way that an entity definition part (the part with **'self'** value for pattern role or Scolem variable key) appears before this particular pattern role or Scolem variable is used to reference a property key in another part representing relational entity.

8. The adapter is writing generated URI to the worksheet (overwriting **new** strings) at the moment this URI is generated. It is possible to reference newly generated URI in some other cell and use it in the same import (see an example below).

13.3. Saving and restoring the mapping

Although the adapter will store your mapping in the spreadsheet itself, the way is provided to save the mapping to a file and restore it from a file. It allows to do several imports with different patterns from the same worksheet as described above. This feature is also useful if you are designing spreadsheet form to be filled by other parties and returned to you for processing, and you can not be sure it will not be reformatted or otherwise corrupted in the process.

To save the mapping to a file press *Save to file* button on the adapter panel. The mapping is saved to a JSON file. The mapper will ask you for the file name and location.

To restore the mapping from a file select the worksheet for which the mapping was saved and press *Load from file* button on the adapter panel. The mapper will ask you to locate the JSON file and will restore the mapping.

Remember that you have to save a separate mapping for each worksheet kind!

13.4. Mapping example

Files for mapping demo can be found in the <*samples*>*mapping* folder. Take the file ***demo_mapping_ptrns.py*** with the pattern library required for demonstration from this folder and copy it to the <*installation_folder*>*patterns* folder. Reload patterns by pressing ***Ctrl+Shift+W***.

Open project file ***demomapping.15926*** from the same folder.

Two files should appear in the project: ***PCA RDL*** from local file (the Editor may ask you to locate it) and ***Demo Local RDL*** data source (***DemoLocalRDL.rdf*** file).

New entities can will be imported to ***Demo Local RDL***. It also contains data for the pattern which describes data structures of the sample spreadsheet. You can see there some additional information recorded for 201 entity types of ISO 15926-2. There are labels defined for all entity types (absent in Part 2 built-in data model) and all entity types are declared to be members of a special ClassOfClass entity *ISO 15926-2 TYPE CLASS.* These data are necessary to form convenient drop-down lists for the spreadsheet form.

Another ClassOfClass entity *Demo Local RDL Class* is also predefined in the local RDL to collect all reference data entities created in the local RDL during imports from the spreadsheet.

Now open the ***SampleData.xls*** entity registration form in Excel (worksheet ***RDL***). We will use this form to define some classes for local reference data library and import them into the ***Demo Local RDL***.

Refer to the pattern **ReferenceDataRegistration** with a single option **PCAMode** in the file ***demo_mapping_ptrns.py***. It describes the data model of this worksheet. It is really a simplified realization of the current rules of PCA RDL, or part of the planned JORD Registration TSP.

An entity (named **Object** in the pattern) is described with all required annotations and link to its Part 2 type. Annotation properties are referenced using the short names defined for the data source ***Demo Local RDL***.

In addition part2:Specialization instance is created linking the new entity to its ***Subclass*** entity, and part2:Classification instance is created to classify the new entity with a special classifier *Demo Local RDL Class* used to collect all reference data entities created in the local RDL.

Look at sample classes already described in the spreadsheet.

There are four classes in total.

A PCA RDL superclass is defined for two of them - once with the name and once with the URI.

There is a drop-down list of Part 2 types and each cell in the ***P2 Type*** column can be filled using it.

There are three columns for annotations used in PCA RDL and a column for a special unique ID property.

Look at the row defining *VERTICAL SALT WATER PUMP* and see that its ***Superclass*** is defined as a link to the cell which will contain an ***URI*** for the class *SALT WATER PUMP*. In this manner an import of the class hierarchy can be structured and executed at once.

In the cells of ***Default ID*** column you can see an Excel function which puts there substrings of the cells from ***URI*** column in the same row. The moment the adapter generates and writes an URI for a newly created entity to the ***URI*** column – this function will cut off the namespace (as it has a known length) and the adapter will use UUID fragment identifier to populate unique ID property.

Make sure that ***Demo Local RDL*** data source is open in the active panel and call mapping adapter (*Import – Build patterns from Excel*). Choose ***SampleData.xls-RDL*** in the *Select sheet* menu. The adapter will load the mapping stored in the worksheet, if it fails to do so - Press *Load from files* button and select ***rdl_mapping_simple.json*** file. Look for the message *Ready for import* under the mapping fields.

Press *Import* button.

You can see URIs and unique IDs being written in the spreadsheet. Some additional information will appear to the right of the main work area of the sheet.

The message *Ready for import* under the mapping fields appears again when the import is completed.

Don't close the adapter panel; you can move it on your screen freely. Look for new entities in the ***Demo Local RDL*** (you can use console query **show(hasCreator='vvagr')** to see all entities created as a result of the first import. Check that their properties and relations were created as described in the pattern.

Now you can change or add data to the worksheet and repeat the import. Please contact TechInvestLab.ru for more pattern mapping examples and consultations on spreadsheet mapping.