.15926 Editor

Version 1.4

Volume 3

Extensions

February 23, 2013

Volume 3. Extensions

Contents

Other documentation volumes:

Volume 1. Getting Started
Volume 2. APIs: Scanner and Builder
Volume 4. Patterns and Mapping

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

10. Extensions

10.1. Introduction to extensions

Functionality of the .15926 Editor can be enhanced through the use of extensions. Extensions are external Python modules loaded by the Editor and available through Editor's menus. No installation of Python on a user computer is required; the Editor itself compiles and executes Python code.

With .15926 Editor extensions you can:

- Access ISO 15926-2 data model, reference data ontologies, and particular data elements specified by the standard (namespaces, roles, annotations, etc.).
- Use various Python libraries or APIs to access and process external data sources of any type (files, databases, web services, etc.).
- Add new or existing ISO 15926 data sources (local files or endpoints) to your project.
- Create new data in local data sources.
- Process ISO 15926 data combining Platform component's methods, external Python libraries and any other functionality available in Python code via API calls or similar methods.
- Add new types of data sources and define views and editing modes for them.

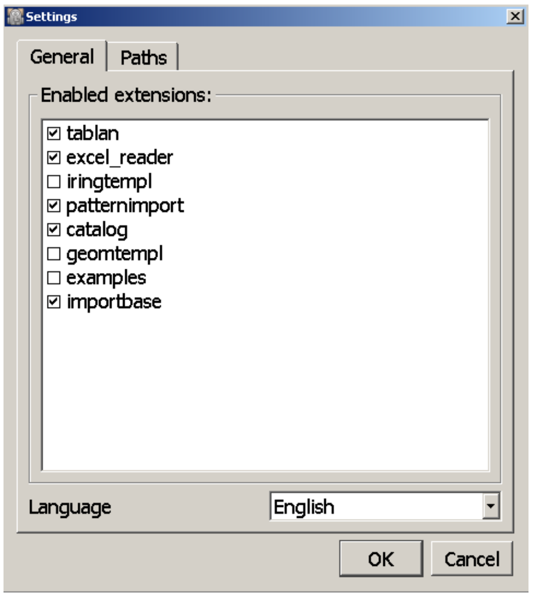
Extensions can access data structures and methods of .15926 Platform components included in the Editor. API functions of Scanner and Builder components are documented in **Volume 2. APIs: Scanner and Builder**. Some useful data structures and methods are listed beloq; others can be discovered through the study of open-source extensions and extension examples available in the *<installation_folder>/extensions* folder and in its subfolders.

Please use **help()** to access Python help utility in the console of the Editor. At **help>** prompt you can get lists of modules, their data structures, classes and methods (degree of component documentation varies).

Extension development requires basic (and probable, better) Python knowledge. Knowledge of PySide GUI toolkit will be a great benefit if really complex behavior of extension is required. Nevertheless simple extension code can be developed and debugged directly in the Python console of the Editor, and registered as an extension following guidelines below and using examples included in the distribution as templates.

Please contact TechInvestLab.ru for help in extension development.

10.2. Registering and enabling an extension



Enabled extensions:
- ☑ tablan
- ☑ excel_reader
- ☐ iringtempl
- ☑ patternimport
- ☑ catalog
- ☐ geomtempl
- ☐ examples
- ☑ importbase

Language: English

Extensions are placed in the *<installation_folder>/extensions* folder of the installation or in its subfolders. Extension can be a single Python file or a whole Python module. Remember that module must be initialized in ***__init__.py*** file.

The Editor automatically recognizes all extensions in *extensions* folder and lists them on the *Extensions* tab in the *Settings* form available through *File-Settings* menu command. To enable an extension check its checkbox, and restart the Editor. Extension will be compiled and menu commands defined in the extension will be added to Editor's menus.

Opening *Settings* form of the newly installed Editor you will find extensions developed by TechInvestLab.ru. For details about extensions *catalog, tablan* and *iringtempl* see Built-in extensions. Extensions *importbase* and *excel_reader* are required but may be disabled, as they are used as libraries by other extensions.

Extension *examples* is not enabled. You can enable it and study functionality of sample extensions placed in the *<installation_folder>/extensions/examples* folder.

If you have changed built-in extensions, please backup extension files before new version installation – an installer will overwrite them!

Some Python libraries are included in the Editor at compilation. If you need to import new external modules to your extension, the Editor may have troubles locating them. You can either copy all required libraries to the *<installation_folder>*, or add paths to them on the *Paths* tab in the *Settings* form as comma separated list.

You can change source code of any enabled extension and recompile it without closing the Editor. Save your changes to extension files and use *File-Reload modules ((Ctrl+Shift+R)* menu command.

10.3. Extension menus

To add a new menu item to main menu bar of the Editor declare new class with **public('workbench.menubar')** decorator.

Class attributes:

- **vis_label**: Text label of new menu item in menu bar.
- **menu_content**: Path to sub-items.
- **position**: Index of item position in main menu, from 10 to 100.

Sample:

```
@public('workbench.menubar')
class xTextMenu:
   vis_label = 'Example menu'
   menu_content = 'dot15926.menu.textmenu'
   position = 10
```

To add a sub-item to an item declare new class with **public('*<menu_content>*')** decorator, where ***<menu_content>*** is path to menu sub-item defined in menu item class.

Class attributes:

- **vis_label:** Text label of new menu sub-item.

Class methods:

- **Update(cls, action):** Update callback of QAction, passed as 'action' argument.
- **Do(cls):** Action required on menu click.

Sample:

```
@public('dot15926.menu.textmenu')
class xTextMenuItem:
  vis_label = 'Open text file...'
  @classmethod
    def Update(cls, action):
      action.setEnabled(True)
    @classmethod
    def Do(cls):
        path = framework.dialogs.SelectFiles('Open text file...')
        if not path:
                return
        doc = TextDoc()
        doc.OpenFiles([path])
        appdata.project.AddDocument(doc)
```

More examples can be found in *extensions/example/*_menu.py* files.

10.4. Accessing internal data structures and methods

10.4.1. ISO 15926 data

Important ISO 15926 data structures (Part 2 data model, standard namespaces, annotation sets for PCA RDL and Part 8, and more) and some external data model elements (XML Schema) are encoded as Python lists and dictionaries in Platform's module *iso15926.kb*.

You can bring this module in the console environment with

```
import iso15926.kb as kb
```

command and use **help(kb)** to list its content, then use **print(…)** to study individual data structures.

10.4.2. New data source

Standard ISO 15926-8 data graph is implemented in the Platform by **iso15926.GraphDocument** class.

To create new reference and project data document create GraphDocument instance:

```
document = GraphDocument()
```

Define parameters for new document, including *name*, *module_name*, *chosen_part2*, *namespaces*, *annotations* (all corresponding to data source properties described above in Data source section):

```
param = dict(module_name='new_data', chosen_part2=kb.ns_dm_part2,
namespaces=kb.namespaces_std_meta,
annotations=kb.annotations_rdfs+kb.annotations_meta)
```

To create a new file for the document call **NewFile()** method:

```
document.NewFile(path, **param)
```

To use document from existing file call **OpenFiles()** method:

```
document.OpenFiles([path], **param)
```

If you want to import data from an external data source via import mapping defined in your importer class – prepare your target document by **NewImport()** method (you'll be able to use *File-Reimport* menu command then) :

```
document.NewImport(path, ImporterType, **param)
```

where arguments are:

- **path**: Path to source file for import.
- **ImporterType**: Importer class.

To add a document to Project panel call **appdata.project.AddDocument()** method:

```
appdata.project.AddDocument(document)
```

If data source is correctly created the Editor will support all standard functionality available for it (save with new name, save as, change tracking, undo\redo. etc.).

10.4.3. Data processing

External libraries

To get data from external data sources you may need a specialized Python libraries (there are plenty to parse HTML, XML, JSON, make SQL or SPARQL queries, access web services of call other APIs, etc., etc.). You can import external Python libraries for use in your extension (do not forget to add paths to them on the *Paths* tab in the *Settings* form).

Platform components

.15926 Platform components are developed to process ISO 15926 data. The set of components available to the extension is defined by the work environment. There are two ways to get work environment – get environment directly or use **EnvironmentContext** instance.

The first way is to get work environment directly – by calling **GetWorkEnvironment()** method of **appdata.environment_manager**.

Example:

```
env = appdata.environment_manager.GetWorkEnvironment(document,
include_builder=True, include_scanner=True, include_patterns=False)
```

Method arguments:

- **document**: Target document, all components in the environment will work with it.
- **include_builder**: Builder is available for specified document if True.
- **include_scanner**: Scanner is available for specified document if True.
- **include_patterns**: Patterns are processed for specified document if True.

When work environment is obtained, you can access available tools using tool name or module name as a key (environment is a dictionary, you can check its keys with **dir(env)** command). Notice that all data sources present in the Project are included in the environment and are available with module names as keys.

For example, after the execution of the following code:

```
builder = env.get('builder', None)
part2 = env.get('part2', None)
p7tpl = env.get('p7tpl', None)
```

standard functionality of Builder and Scanner becomes available in your extension with API function's syntax documented in respective sections of this documentation.

For example, the following will work:

```
part2.ClassOfRelationship()

builder.annotate(id = class_id, annSource=source)

p7tpl.ClassOfArrangementOfIndividual(item, whole)
```

You can learn more about access to environment and its use by analyzing *importer.py* files found in *catalog* and *tablan* extension folders.

The second way to get work environment is to use **EnvironmentContext** instance for execution of Python code. It allows you to execute Python string or Python file in an environment replicating the one used in the Editor's console, different only by local variable set. In this way you can develop and debug your Python code in the console and later use this code as a base for registered extension available from menu.

To create **EnvironmentContext** instance call:

```
context = EnvironmentContext(document, locals)
```

- If specific **document** is required as environment target - pass it to the constructor as a first argument. Passing **None** as a first argument initializes environment with active view or active document as a target.
- If some local variables have to be passes to main extension code from console-developed code – pass the dictionary **locals** to the constructor as a second argument. After the execution of console-developed code values of local variables become available in this dictionary with variables as keys. For example:

```
locals = {}

context = EnvironmentContext(None, locals)

context.ExecutePythonString('a = find(type=part2.Class)')

print locals['a']
```

When console-replicating environment is created, you can run console-developed code using **ExecutePythonString()** or **ExecutePythonFile()** method.

For method **ExecutePythonString()** see example above.

ExecutePythonFile() method requires path to the file with console-developed code as an argument.

You can learn more about usage of **EnvironmentContext** for building extensions from console-developed code by analyzing *transformation_main.py* and *transformation_commands.py* files found in *examples* extension folder.

10.5. Defining new data views

Extensions allow you to define new types of data sources and define custom views for them. You can include texts or pictures in the Editor panels, and even make them editable.

In the extension *examples* simple view and edit functionality is implemented for a text file. If this extension is enabled, menu command *Example menu - Open text file...* appears in the Editor. It adds text file as a data source to the Project panel. This file can be opened in data panel and edited in it. Text will behave like other data sources – it can be saved, and it will be restored in the project after the Editor restart.

Menu item for text file opening is defined in *text_menu.py* file in *samples* extension.

Creation of text data is handled by *text_doc.py* file. To create a new text data source declare new class inherited from **framework.Document**. The Editor uses document-view architecture and appropriate view type should be set during document initialization.

Initialization example:

```
def __init__(self):
  framework.Document.__init__(self)
  self.viewtype = type('', (TextView,), dict(document=self))
  self.my_text = ''
```

Opening and saving files occurs through overloaded methods **OpenFiles()**, **Save()** and **SaveAs()**. Other information about document methods used can be found by **help()** command. When large files are loading or saved it is suggested to use worker thread instead of main thread. The example of worker thread use can be found in *text_doc.py*.

Documents in the Editor use simple event system to send events. Only two methods are needed: **Subscribe(self, item, subscriber)** and **Unsubscribe(self, item, subscriber)**. This event system is stored as global variable **wizard.**

Example of event sending:

```
wizard.W_TextDocChanged(self)
```

View class **TextView** is defined in *text_view.py* file in *samples* extension.

Note that one document can be rendered in multiple views. View class should inherit from **framework.View** class, which is inherited from QDockWidget. Any PySide element can be added to view.

To subscribe for document events view class uses **wizard** event system. Example of subscription for event:

```
wizard.Subscribe(self.document, self)
```

To handle events appropriate methods are defined in view class.

Example of handler for **W_TextDocChanged** event:

```
def W_TextDocChanged(self, doc):
        self.textfield.textChanged.disconnect(self.OnText)
        self.UpdateView()
        self.textfield.textChanged.connect(self.OnText)
```

Please use **help('framework.document')** and **help('framework.view')** console commands to find more about available methods and interfaces.

Text data source can be accessed from console. Please look at *text_con.py* file in *samples* extension. There a simple command is defined which overwrites the opened file with its string parameter.

For example:

```
SetText('New document text')
```

11. Built-in extensions

Current version of .15926 Editor is distributed with several open-source extensions developed by TechInvestLab.ru and with some examples. Source code for these is available in the *<installation_folder>/extensions* folder of the distribution and in its subfolders.

The code of built-in extensions and extension prototypes is 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.**

This section contains short descriptions of released extensions with further references to appendices or separate documents for detailed information. Notice that TabLan and Catalog adapters were developed some years ago before the concept of pattern mapping appeared and became available for data model mapping in the Editor.

Extensions can be turned off or completely removed from the software.

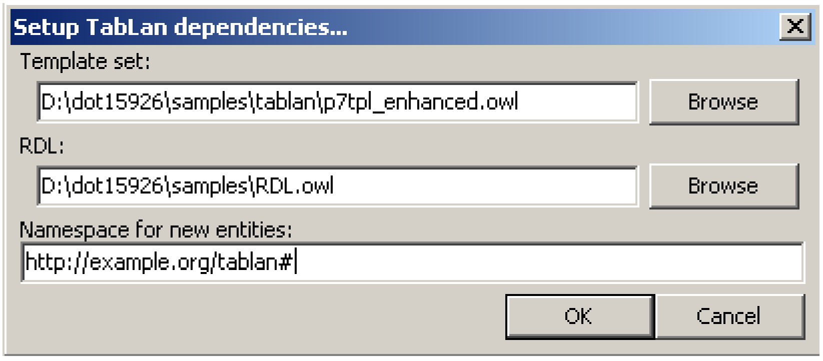
11.1. TabLan import

TabLan import extension facilitates import of data from data tables prepared according to the TabLan.15926 modelling methodology. TabLan.15926 tables are designed for formalized information extraction from engineering project technical specification documents.

Using TabLan import you can convert data from TabLan.15926 table (in Office Open XML .xlsx format) to ISO 15926. Conversion process creates reference data (classes, relationships and template instances) in Part 8 format, which can be further used to build project RDL to support data integration in CAD/CAE/PLM environment, organize data in PLM systems or facilitate requirement's traceability and verification.

The mapping of TabLan to ISO 15926 data structures is described by a simple Python code in a special .15926 Platform module using .15926 Builder API.

Full specifications for TabLan.15926 language and data mapping to ISO 15926 can be downloaded from http://techinvestlab.ru/TabLan or found in the documentation folder of this installation. Customized set of template definitions, sample TabLan data model in .xslx file and imported .rdf file can be found in *<samples>\tablan* folder.


Setup TabLan dependencies...
Template set: D:\dot15926\samples\tablan\p7tpl_enhanced.owl
RDL: D:\dot15926\samples\RDL.owl
Namespace for new entities: http://example.org/tablan#

TabLan.15926 is just an example of a simplified Gellish-like table language which can be mapped and translated to ISO 15926 with the help of .15926 Platform tools.

Menu command *Import - Setup TabLan dependencies…* - opens form for TabLan model import settings with the following fields ((settings are saved between Editor launches in the ***dot15926.cfg*** configuration file in <*installation_folder*>)):

– location of the file with template definitions for templates used in import mapping;
– location of the file with reference data used in TabLan model;
– namespace for new entities created during import process.

Select file ***p7tpl_enhanced.owl*** from folder *<samples>\tablan* as template definitions*.* Templates from the ***p7tpl_enhanced.owl*** file are instanced during the conversion. When loaded, template set is named "Template set (TabLan)" and registered with p7tpl module name (do not change it, or import will not work).

Select PCA RDL as file with reference data used in TabLan model. TabLan data is linked to PCA RDL reference data.

Fill in the namespace or leave default namespace in this field.

Then go to *Import – Import TabLan reference data from.xslx table…* menu and choose TabLan model ***Example_One.xlsx*** file from the same folder. Wait while template set and RDL are loaded. Do not close RDL or template set data sources until import process ends, or import will fail.

Imported data source is marked with **Imported file:…* name in the Project panel. You have to use *Save as…* (*Ctrl+Shift+S*) command to save imported data to a new .rdf file. Or you can change something in the source table, save it and repeat import with *File - Reimport* menu command.

You will find unrecognized entities (marked by "?") in some relationships. These are reference data items from TechInvestLab.ru sandbox. To see them properly add TechInvestLab.ru endpoint to the Project panel through *File – Add SPARQL endpoint… - TechInvestLab Sandbox* menu command. Then you can select any unrecognized entity and use *Edit - Search endpoints for URI (F4)* menu command. Or simply open TechInvestLab.ru endpoint in a separate panel and search for an empty string in the search box at the top of the panel. After all sandbox entities are added to the view you can reload unrecognized entities in import results.

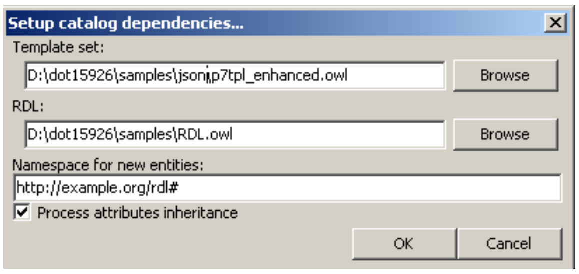
11.2. Catalog data model import

Catalog import extension facilitates import of reference data from data model of an engineering catalog application developed by 3V Services (third party software). For more details refer to Appendix A. Catalog Export Specification and Mapping.

Customized set of template definitions and two sample JSON files with fragments of catalog data model can be found in *<samples>\json* folder: ***ClassificationExample.json*** and ***ClassificationPart.json***.

Menu command *Import - Setup catalog dependencies…* opens form for catalog data import settings with the following fields ((settings are saved between Editor launches in the ***dot15926.cfg*** configuration file in <*installation_folder*>)):

- location of the file with template definitions for templates used in import mapping;
- location of the file with reference data used in import mapping;
- namespace for new entities created during import process;
- checkbox to specify whether inherited attributes have to be directly assigned to possessor classes.



Template set: D:\dot15926\samples\json\p7tpl_enhanced.owl

RDL: D:\dot15926\samples\RDL.owl

Namespace for new entities: http://example.org/rdl#

☑ Process attributes inheritance

Select file ***p7tpl_enhanced.owl*** from *<samples>\json* folder as template definitions*.* Templates from the ***p7tpl_enhanced.owl*** file are instanced during the conversion. When loaded, template set is named "Template set (Catalog)" and registered with *p7tpl* module name (do not change it, or import will not work).

Select PCA RDL as file with reference data used in import. Catalog data is linked to PCA RDL reference data.

Fill in the namespace or leave default namespace in this field.

Check the checkbox if you want to see catalog classes directly linked to attributes they've inherited from their superclasses.

Go to *Import catalog data from JSON file…* and select JSON file from in <*samples*>*json* folder. Wait while template set and RDL are loaded. Do not close RDL or template set data sources until import process ends, or import will fail.

Imported data source is marked with **Imported file:…* name in the Project panel. You have to use *Save as…* (*Ctrl+Shift+S*) command to save imported data to a new .rdf file.

11.3. IIP template table import

IIP template import is a built-in extension for import of template definitions from Excel spreadsheet used to populate iRING Tools software with templates and reference classes.

An *.xslx* file with the set of IIP reference data can be found in <*samples*>*iip* folder. The work on templates is work in progress, this file is the latest version available at the time of Editor release. It can not be considered more then an example!

Menu command *Import - Import IIP templates from xlsx table…* opens a dialog to select the spreadsheet (in .xlsx format) for import.

Imported template definition data source is marked as **Imported file:…* name in the Project panel. You have to use *Save as…* (*Ctrl+Shift+S*) command to save imported data to a new .owl file. Conversion results can be also found as file ***iip_fullset_20140131.owl*** in <*samples*>*iip* folder.

11.4. Template table import

Template import is a built-in extension for import of template definitions from the Excel spreadsheet similar to the previous one. The difference is in the URI handling. While IIP template table importer uses URIs directly specified in the spreadsheet, this template importer generates new URIs based on the import settings.

Please contact TechInvestLab.ru if you want to find more details about spreadsheet template format and conventions used for this importer.

11.5. Spreadsheet mapper

This powerful mapping extension is fully documented in **Volume 4. Patterns and Mapping**.

****

Appendix A. Catalog Export Specification and Mapping

A.1. Data model export specification

Adapter included as extension in the released version of .15926 Editor is designed for the engineering catalog application developed by 3V Services company (third party software). This catalog is developed on ENOVIA platform (Dassault Systemes). For more details about engineering catalog application refer to the developer's web site http://3v-services.com/ru/index.php?option=com_content&view=article&id=87 (in Russian).

Catalog data model was exported as JSON file from a web-service with method *getClassification(id)*

Class structure can be exported from the class *id*. If *id* is empty – full class structure from the root is exported.

Format:

Each class is represented by JSONObject with fields:

- id – internal ENOVIA identifier of a class (string)
- name – Russian class name (string)
- subClasses – subclasses of a class (array)
- attributes – attributes of a class (array)

    - id – internal ENOVIA identifier of an atttribute (string)
    - name – Russian attribute name (string)
    - type – attribute type (string).
        - Can have values: "string", "real", "integer", "boolean", "date", "ref"
    - unit – system name of UOM (string)

Catalog application supports attribute inheritance along the class structure, but inherited attributes were not exported.

A.2. Sample JSON reply

```
{
  "id": "RECPart",
  "name": "Голова",
  "attributes": [
        {
                "id": "Title",
                "name": "Наименование",
                "type": "string",
                "unit": ""
        },
        {
                "id": "relationship_REC_Protege_supplier_organization",
                "name": "Организация-поставщик",
                "type": "ref",
                "unit": ""
        }
  ],
  "subClasses": [
      {
                "id": "REC_Protege_Process_Equipment",
                "name": "Технологическое оборудование",
                "attributes": [
                {
                        "id": "REC_Protege_pressure_in",
                        "name": "Давление на входе клапана",
                        "type": "real",
                        "unit": "p"
                        },
                        {
                        "id": "REC_Protege_pressure_out",
                        "name": "Давление на выходе клапана",
                        "type": "real",
                        "unit": "p"
                        }
                ],
                "subClasses": [
              {
                    "id": "REC_Protege_Heat_transfer_equipment",
                    "name": "Теплообменное оборудование",
                    "attributes": [ ],
                    "subclasses": [ ]
              },
              {
                    "id": "REC_Protege_Material_transfer_equipment",
                    "name": "Транспортирующее оборудование",
                    "attributes": [ ],
                    "subclasses": [ ]
              }
            ]
        }
  ]
}
```

A.3. ISO 15926 mapping for catalog data model

This mapping from JSON Catalog data model export format is designed to automate initial stage of ISO 15926 mapping process for catalog application.

Reference data items created during an automated transformation of data model described below are not intended to become the final reference data items used for neutral representation of catalog data in data integration. Automated mapping creates ISO 15926 compliant data model which is just mirroring original application's data model with a small addition of ontology analysis. Drastically simplified mapping patterns based on the catalog attribute types are used in this mapping.

Such computer-generated data model of the original application have to be further reviewed by a specialist ISO 15926 data modeler, compared to reference data in reference data libraries used and to common data characterization patterns. To facilitate data exchange a fully functional mapping complying with recognized mapping methodology (preferably with PCA one, refer to https://www.posccaesar.org/wiki/FiatechJord for more details) should be created from catalog data model (whether in original format, or represented by JSON, or as automatically mirrored as described below).

Automated data model transformation can sufficiently simplify data model analysis and bring the work to the .15926 Editor environment specifically designed as a data modeler's tool.

a. Prerequisites

a.1. Namespace *proj:* for new reference data URI – it is specified as transformation parameter.

a.2. PCA reference data library as a local file is used for one reference data entity only:

```
DATE AND TIME
http://posccaesar.org/rdl/RDS16432820
```

a.3. Template set used is a standard proto and initial template set enhanced with one additional template:

```
PropertyRangeTypeRestrictionOfClass(
        1: hasClass: ClassOfIndividual,
        2: hasRestrictedProperty: ClassOfIndirectProperty,
        3: hasScale: Scale)
```

With the axiom:

```
PropertyRangeTypeRestrictionOfClass(x1, x2, x3) <->
        EXISTS u EXISTS v
        (PropertyRangeMagnitudeRestrictionOfClass(x1, x2, x3, u, v))
```

a.4. Three additional reference data entities not found in RDL are used:

```
CATALOG RUSSIAN IDENTIFICATION : ClassOfClassOfIdentification
CATALOG ATTRIBUTE TYPE : ClassOfClass
CATALOG EQUIPMENT CLASS : ClassOfClassOfIndividual
```

<u>b. Attribute processing</u>

Fields of an attribute in a JSONObject are identified below as:

id: *attribute_ENOVIA_id*
name: *attribute_Russian_name*
type: *attribute_type*
unit: *UOM*

When a new (judging by the field *attribute_ENOVIA_id*) attribute is met in the source file the following entities are created:

b.1. Part 2 type instance:

URI = attribute_URI generated in namespace *proj:*
label = "*attribute_ENOVIA_id"*
type = depending on value of *attribute_type* field:
"string" : ClassOfClassOfIndividual
"real" : ClassOfIndirectProperty
"integer" : EnumeratedPropertySet
"boolean" : ClassOfStatus
"date" : ClassOfRelationshipWithSignature
"ref" : ClassOfClassOfIndividual

b.2. Template instance:

Classification(*URI_of_attribute*, CATALOG ATTRIBUTE TYPE)

b.3. If attribute processed has *attribute_type* field with value "date", additional reference data is created:

b.3.1. RoleAndDomain instance

URI = URI generated in namespace *proj*:
label = "POSESSOR_OF_ *attribute_ENOVIA_id*"
type = RoleAndDomain

b.3.2. RoleAndDomain instance

URI = URI generated in namespace *proj:*
label = "RANGE_OF__ *attribute_ENOVIA_id*"
type = RoleAndDomain

b.3.3. Roles for the *attribute_ENOVIA_id* instance of ClassOfRelationshipWithSignature are assigned:

hasClassOfEnd1 = POSESSOR_OF_ *attribute_ENOVIA_id*
hasClassOfEnd2 = RANGE_OF_ *attribute_ENOVIA_id*

b.3.4. Template instance:

Specialization(RANGE_OF_ *attribute_ENOVIA_id,* DATE AND TIME) b.4. If an attribute has non-empty *UOM* field, the check is performed whether such *UOM* was met earlier. If nor, an instance of Scale is created:

> URI = URI generated in namespace *proj:*
> label = "*UOM"*
> type = Scale

<u>c. Class structure processing</u>

Fields of a class in a JSONObject are identified below as:

> id: *class_ENOVIA_id*
> name: *class_Russian_name*
> type: *attribute_type*

JSONOblect structure is processed and for each class the following entities are created:

c.1. Part 2 type instance:

> URI = attribute_URI generated in namespace *proj*:
> label = "*class_ENOVIA_id*"
> type = ClassOfIndividual

c.2. Template instances:

> Instances of Specialization according to the *subClasses* defined by JSONObject structure

c.4. Template instance:

> Classification(*class_ENOVIA_id*, CATALOG EQUIPMENT CLASS)

c.5. For all attributes of a class depending on the value of *attribute_type* field the following reference data is created:

> for "string"
>> ClassOfClassification(*class_ENOVIA_id*, *attribute_ENOVIA_id*)
>
> for "real"
>> PropertyRangeTypeRestrictionOfClass(*class_ENOVIA_id*, *attribute_ENOVIA_id*, *UOM*)
>
> for "integer"
>> ClassOfClassification(*class_ENOVIA_id*, *attribute_ENOVIA_id*)
>
> for "boolean"
>> ClassOfClassification(*class_ENOVIA_id*, *attribute_ENOVIA_id*)
>
> for "date"
>> Specialization(*class_ENOVIA_id*, POSESSOR_OF_ *attribute_ENOVIA_id*)
>
> for "ref"
>> ClassOfClassification(*class_ENOVIA_id*, *attribute_ENOVIA_id*)

c.6. Optionally at data transformation time attribute inheritance should be processed and for each class relationships from clause 3.5 above should be established for all its inherited attributes.

d. Russian name processing

If an attribute or a class has non-empty *attribute_Russian_name* or *class_Russian_name* field respectively, the following entities are created:

d.1. Part 2 type instance:

URI = attribute_URI generated in namespace *proj:*
label = "*attribute_Russian_name"* or "*class_Russian_name "*
type = ExpressString

d.2. Part 2 type instance:

URI = attribute_URI generated in namespace proj:
type = ClassOfIdentification

This instance of ClassOfIdentification (*URI_identification*) has roles defined for attribute or for a class respectively:

hasRepresented = *attribute_ENOVIA_id*
hasPattern = *attribute_Russian_name*

or

hasRepresented = *class_ENOVIA_id*
hasPattern = *class_Russian_name*

d.3. Template instance

Classification(*URI_identification*, CATALOG RUSSIAN IDENTIFICATION)

***