# pyMDG

Oct 10, 2022

# Features:

Take UML models created in Sparx EA or DrawIO and use pyMDG to generate code, schema and documentation. Provided templates generate Django, OpenAPI, Avro, POJOs and more. If you create reusable templates or improve upon the provided ones, please submit them in GitHub.

Currently used for generating data platforms, API & Kafka schema via UML packages, classes & enumerations.

CHAPTER 1

# Modelling

pyMDG has a few opinions on the way we data model. This nomenclature enables rule based declaration of things like API endpoints. This can be seen in the nomenclature reference documentation and also see the sample data models in the sample_recipies folder in the Git repo.

# CHAPTER 2

# Quickstart

The github project comes with sample recipes which can generate a many artifacts including a Django Rest data platform. The generation is definied in recipies, see the sample_recipies or the recipies docs to see how they work.

Generation:

```
git clone https://github.com/Semprini/pyMDG
cd pyMDG
virtualenv venv
. venv/bin/activate or .\venv\Scripts\activate
pip install -r requirements.txt
python -m mdg.tools.mdg_tool generate ./sample_recipe/drawio/config-drawio-django.yaml
```

Run the app:

```
cd build/sample_drawio_django/SampleIndustry
pip install -r requirements.txt
python manage.py makemigrations TestDomain TestDomain2
python manage.py migrate
python manage.py createsuperuser
python manage.py runserver
```

And browse to http://127.0.0.1:8000/admin/ or http://127.0.0.1:8000/api/

When installed via pip, and you've created your own project, pyMDG provides the mdg-tool command.

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

## 3.1 MDG Tool

pyMDG installs an execuitable called mdg-tool. We run the tool to generate artifacts but it can also validate and dump to JSON. A command to start a new project is uncompleted.

Usage:

```
mdg-tool [-h] [--verbose] {generate,validate,dumps,startproject}
```

### 3.1.1 Generate

Runs the model generation based on provided recipie

Usage:

```
mdg-tool generate [-h] recipe_path
```

### 3.1.2 Validate

Parses the source model specified in the recipie and checks:

- Does each concrete object have an Id field
- Both a parent and specialization should not have an Id
- Each 'auto' stereotyped attribute is either int or bigint

Usage:

```
mdg-tool validate [-h] recipe_path
```

### 3.1.3 JSON Dumps

This command will parse the source model from the provided recipie and dump the internal representation as JSON.

Usage:

```
mdg-tool dumps [-h] recipe_path
```

## 3.2 Recipes

The gitgub project has a sample_recipes which has some examples. Each recipe config is in yaml and has the following structure:

```
root_package: # <Root package name from modelling tool. Not needed for Sparx DB␣
↪sources>
model_package: # <Model package name within root_package where models are held. Sparx␣
↪DB parser uses package GUID>
test_package: # <Test package name within root_package where instances used for test␣
↪are held. Sparx DB parser uses package GUID>
source: # <path to exported model - xmi 2.0, drawio or for DB connection see https://
↪docs.sqlalchemy.org/en/14/core/engines.html>
parser: # <sparx, sparxdb or drawio>
dest_root: # <base path where to put each generated artifact>
templates_folder: # <base path where to find custom templates>
generation_type: # <language to translate attributes into optios are: default, spring␣
↪data rest, django, marshmallow, sqlalchemy, python, ddl>
model_templates: # <list of artifacts to generate, see below for details>
- dest: # <jinja template string which parses to the output file name>
  level: # <package, class, root, copy>
  source: # <path to jinja template file to render>
test_templates: # <list of test artifacts>
```

### 3.2.1 Model Templates Detail

In the config yaml for your project there is a list called:

```
model_templates:
```

This is a list of files to render based on the parsed classes as discussed in the Metamodel page

Each template item in the list can specify:

- dest: Mandatory. A jinja template string which is passed the package and renders to the output filename
- level: Mandatory. package, class or root - see below for details
- source: Mandatory. Path to the jinja template file
- filter: Optional. A jinja template string which causes the template to be rendered only if the filter renders to "True"

For example here is a django model template entry:

```
- dest: "{{package.path}}/models.py"
  level: package
  source: Django/app/models.py.jinja
  filter: "{% if package.classes %}True{% else %}False{% endif %}"
```

The output file will be based on the package path and the filter will exclude any packages without classes.

### 3.2.2 Model Template Levels

#### Package Mode

In package mode, each package is passed to the template and the template will probably want to iterate over the classes in the package. This is used when your output file has multiple data objects defined - like a django models.py

#### Class Mode

In class mode, each class is passed to the template. This is useful when There is a file output per class - like a POJO file.

#### Root Mode

The template will be called once and passed the root package

#### Copy Mode

No template will be called, the source file will simply be copied to the destination

## 3.3 Output Templates

The project has a set of templates provided in the mdg/templates folders. A template is rendered by specifying it in a recipe and calling mdg-tool generate <recipe file name>.

The recipe will also specify the level. If the level is “package” then the template will be called once for each package and subpackage in the UML model. If the level is “class” then the template will be called once for each class in each package in the UML model.

Passed into the template will be either a “package” or a “cls” argument. The provided argument will conform to the Metamodel.

We can then render almost any artifact.

Here’s an example of generating a list of classes in a package:

```
{% for cls in package.classes %}
    {{ cls.name }}
{% endfor %}
```

With the recipe specifying the above template at package level, we would get a file generated for each UML package containing just the names of the classes in the package

## 3.4 Test Cases

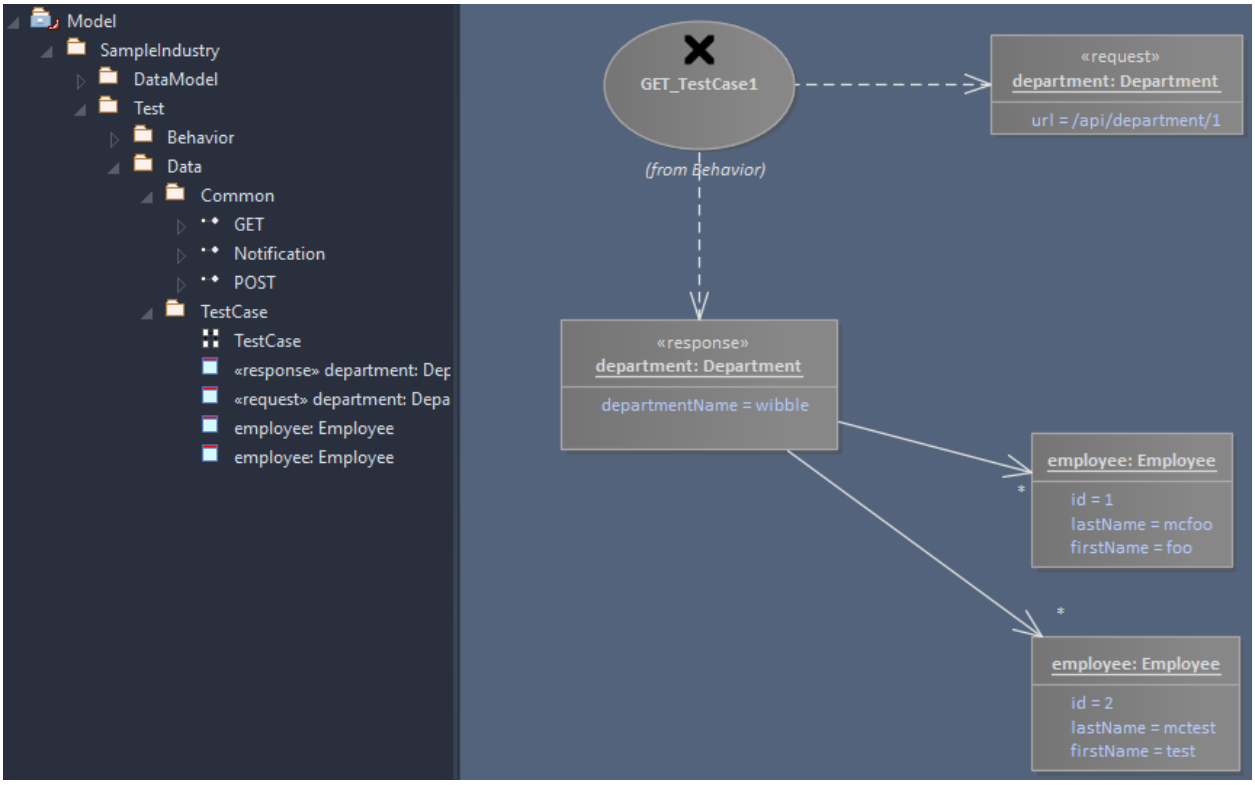
When generating, a recipie can optionally specify a test package containing test cases to output JSON request and response objects. Specify the package:

```
test_package: "{C2E30D48-F8AB-4c6f-9FA3-AFE44653D5EB}"
```

. . . and specify the output:

```
test_templates:
- dest: ./build/json{{ins.package.path}}/{{ins.stereotype}}.json
  format: json
```

In the sample Sparx EA project the SampleIndustry package has the Test child package where the teat modelling can be seen:




## 3.5 Diagrams.net Django Tutorial

To use pyMDG we need to build:

- A UML model
- A generation recipe

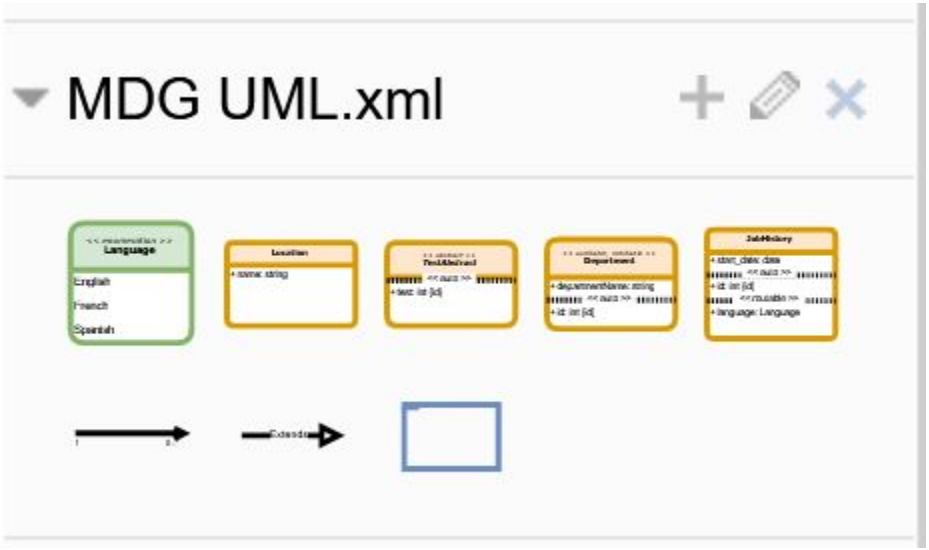
### 3.5.1 Data Modelling

pyMDG has a specific UML nomenclature. This needs to be added into diagrams.net by selecting:

File -> Open Library From -> URL and enter:

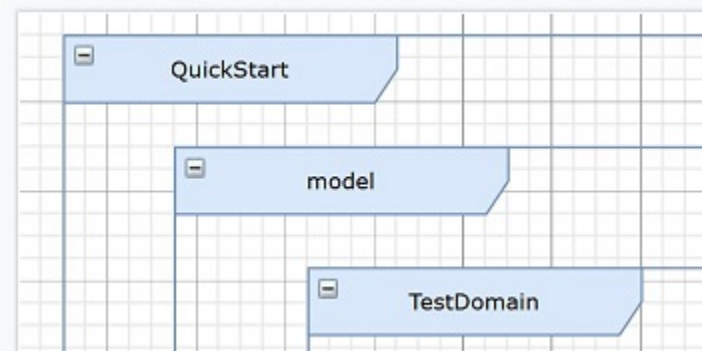
```
https://github.com/Semprini/pyMDG/raw/master/mdg/tools/DrawIO%20MDG%20UML%20Library.
↪xml
```

This will result in the pyMDG library being added to the sidebar:



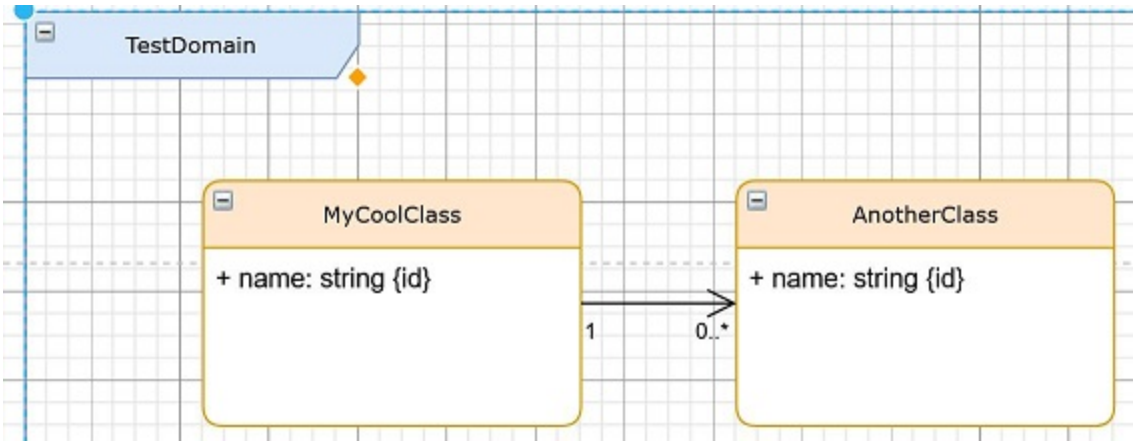

UML packages are set up as Frames in the library. This mimics the hierarchy found in full modelling tools like Sparx. Add 3 nested frames to the canvas and rename each:

- Top level is the name of our business domain/data platform
- 2nd level is the model container (test container can be added later)
- 3rd level is a data domain/app




We can then start modelling classes. Each class must have an attribute with {id} except where the {id} is in a parent class.

Drag 2 'Class Basic' objects into the TestDomain package. Rename classes and set the {id} attribute of each class. Drag on an Association and link the classes:




There are 5 templates for classes:

- Basic: Vanilla class with attributes
- Abstract: The <<auto>> attribute stereotype is optional.
- Stereotyped: Shows auditable (data platform will track changes) and notifiable (changes will cause events to be sent to the message broker)
- Routable: When events are sent to message broker then routing keys will include attributes with <<routable>> stereotype.

pyMDG supports 4 relation types:

- Association: Forms relations between classes:
  - One to One
  - One to Many
  - Many to One
  - Many to Many
- Generalization: Defines a parent/child inheritance
- Composition: Similar to association but can be used to control generated features. For example, when generating an OpenAPI schema, objects which are part of a composition may not get thier own endpoint.
- Aggregation: Like composition aggregations affect the features of our output. For example, an OpenAPI generation can use aggregations to specifiy when the endpoint is a sub-endpoint I.e. /customer/12/customer_address/1/

Export the diagram by File -> Export As -> XML and unselect Compressed

### 3.5.2 Generate

The recipe tells pyMDG about your model and what files to output. This tutorial uses the sample templates and config which you can find in the sample_recipe folder from the project on gitgub: https://github.com/Semprini/pyMDG

#### recipe - Django

A complete django app with django rest api can be created from the model.

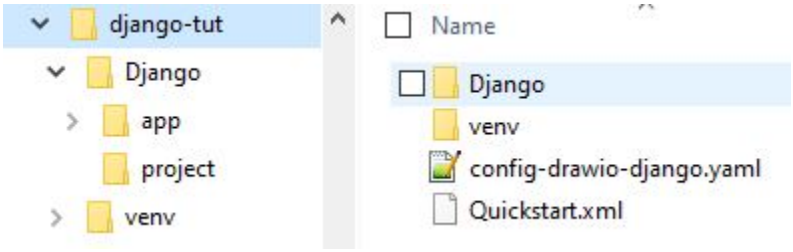
cd into a new project folder (I called mine django-tut):

```
django-tut> virtualenv venv
django-tut> pip install pymdg
```

Copy the following from the github project pyMDG/sample_recipe/drawio folder:

- config-drawio-django.yaml

The dir structure now looks like:




Edit config-drawio-django.yaml and update the following:

- root_package: QuickStart (matches the top level package)
- model_package: model (matches the middle package)
- source: the path to your saved model file
- templates_folder: ''
- dest_root: ./build

We can now build the project:

```
> django-tut> mdg-tool generate .\config-drawio-django.yaml
Config file loaded: .\config-drawio-django.yaml
Base Model Package: model
Generating model output for package /QuickStart/
Generating model output for package /QuickStart/TestDomain/
Generating test case output
```

And then run the generated django app:

```
> cd build/QuickStart/
> pip install -r .\requirements.txt
> python manage.py makemigrations TestDomain    <- matches the inner package
Migrations for 'TestDomain':
TestDomain\migrations\0001_initial.py
 - Create model AnotherClass
 - Create model MyCoolClass

> python manage.py migrate
> python manage.py createsuperuser
> python manage.py runserver
```

You can now browse to http://127.0.0.1:8000/admin/ and http://127.0.0.1:8000/api/

## 3.6 Sparx EA XMI Tutorial

This tutorial uses Sparx EA version 16 but any version which can output XMI 2.1 should work. Note, version 16 of Sparx EA changes it's internal file format to a SQLite DB with the same schema as the DB based repository so

pyMDG will add this as the prefered mode and XMI export will stop having features added.

To use pyMDG we need to build:

- A UML data model
- A generation recipe

We start by creating a folder somewhere called tutorial1.

### 3.6.1 Data Modelling

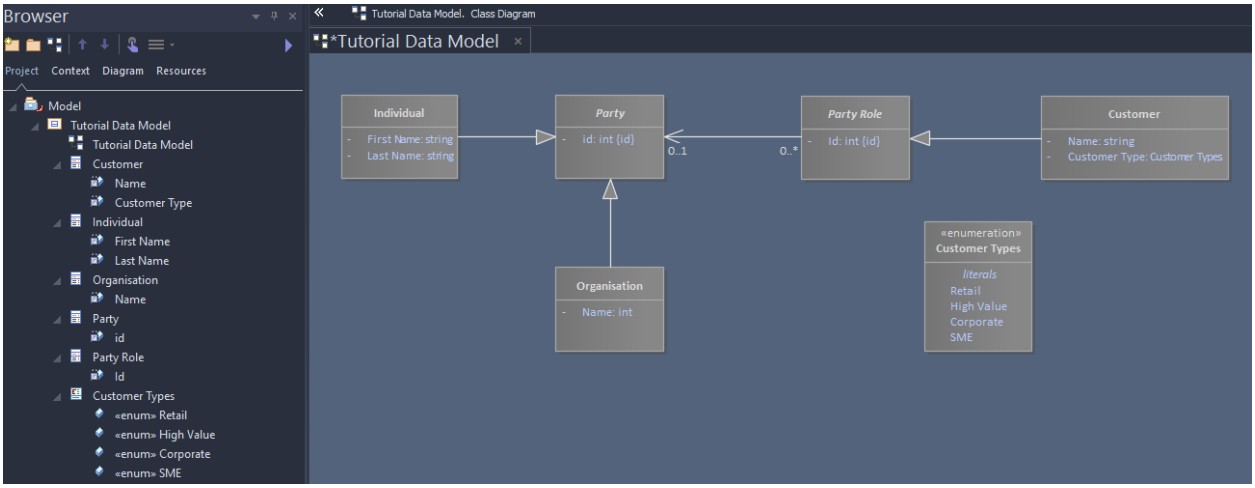
pyMDG has a specific UML nomenclature, expects Sparx EA project to have a package as a "Model Root" and an optional separate package root for test objects. pyMDG is a little opinionated in how we should create logical data models - We should always use plain language and not use any implementation specific details like snake_case. Let the generation handle realizing the physical models based on the rules in our generation templates.

1. Create the project

   In Sparx EA, create a new project called tutorial1 in the tutorial1 folder. This will add a default "Model" package as a start.
   Right click on the "Model" package and select Add View...
   This will show the "Add Package" window. Name the package "Tutorial Data Model" and select the "Create Diagram" option. Click Ok.
   This will show the "New Diagram" window. In the "Select From" panel selct "UML Structural" and in the "Diagram Types" panel select "Class" and hit Ok. The diagram name will pickup the package name.
   Open up the diagram and now we model.

2. Data Model

   We'll model a starndard party / party role for the customer role in this tutorial. Drag a Class object into the diagram, name it Party and set it as Abstract.
   Add an integer attribute to Party called id and set "is ID" to true.
   Add a new Class object onto the diagram, name it Individual and add a Generalization from Individual to Party.
   Add "First Name" and "Last Name" attributes to the Individual object. Make the attribute type of both "string"
   Add a new Class object onto the diagram, name it Organisation and add "Name" string attribute and a Generalization from Organisation to Party.
   Next we add the "Party Role" abstract class with an integer Id field. We also add a "Customer" object with a "Name" string attribute and a Generalization relationaship fom Customer to Party Role.
   To relate party role to party we add an Association from Party Role to Party. In the association properties, set the direction to "Source -> Destination" this is to make is obvious who owns the relationship which is important when generating code.
   A party can play many roles so under Properties we go to Source and set Multiplicity to 0..* and under Target we set 0..1. It's just a personal preference so you may want the Target to not be optional and so use a Multiplicity of 1.
   The last object to put on the diagram is an Enumeration. Name it "Customer Types" and add these attributes: Retail, High Value, Corporate, SME.
   To use our enumeration we add an attribute to the Customer object called "Customer Type" and under the Type dropdown choose "Select Type..." and browse to the Customer Types enumeration.
   Your project should now look like:




3. Exporting

   In the Sparx EA Browser window, single click on the "Tutorial Data Model" package.
   In the top menu bar select "Publish" and choose "Publish As…". I prefer this to "Export Package" as it gives the option to export diagram images if we're generating documentation.
   In the Publish Model Package window, make sure the Package is set to Tutorial Data Model, set an output file as tutorial1.xmi in the tutorial1 folder you created earlier, choose UML 2.1 (XMI 2.1) and hit Export.
   You can open the exported file in a text editor like Notepad++ and the 2nd line should say:
   `<xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.1">`

My generated XMI file can be found here: https://raw.githubusercontent.com/Semprini/pyMDG/master/docs/tutorials/sparx/tutorial1.xmi

### 3.6.2 Generation

We can generate many things from our UML declaration. Today I feel the need to generate an OpenAPI and an AVRO schema. We need to tell pyMDG where our source file is, what some of our preferences are and how our Sparx project is structured. We do this in a recipe file.

pyMDG parses the XMI into the internal classes shown in the metamodel section of the docs. It then uses a Jinja2 template for each of the artifacts it needs to generate. pyMDG comes with some templates to get you started - the OpenAPI template can be found in `mdg/templates/Schema/openapi.yaml.jinja` and defines GET endpoints for each non-abstract object which must have an Id field (in our case the Id fields for Individual, Organisation and Customer are inherited from an abstract through the generalization associations).

1. Recipie

   Our first step is to create a file in our tutorial1 folder called schemagen.yaml.
   We then add info on how our project is set up::

   root_package: Tutorial Data Model
   model_package: Tutorial Data Model

source: ./tutorial1.xmi
parser: sparx

Next we add the generation type and where to find our jinja2 templates to the yml::

dest_root: ./build
templates_folder: ./mdg/templates
generation_type: default

Lastly we add a list of the artifacts we want to generate::

model_templates:
# Avro Schema
- dest: “avro/{{cls.package.name}}.{{ cls.name }}.avsc”
level: class
source: “Schema/avro.avsc.jinja”
# OpenAPI Schema
- dest: “openapi/{{ package.name }}.yaml”
level: package
source: Schema/openapi.yaml.jinja

Each list item needs to specify:

- Which Jinja2 template we want to use which will add to the path specified in “templates_folder” but also look through the internal pyMDG templates.
- A level which specifies if we want the template run for each class or for each package. I want to generate an avsc file per UML class object and an open api yaml file for the package.
- Where we want to place the resulting artifact. We can use a jinja2 method to include our model structure in the filenames. If the level is class, the “cls” object is passed here and if the level is package then the “package” object is provided. Again see the meta model for what the fields are.

My complete recipe file can be found here: https://github.com/Semprini/pyMDG/raw/master/docs/tutorials/sparx/schemagen.yaml

2. Generation

Next we open a CMD prompt and cd into tutorial1
I always use a virtual environment so enter::

virtualenv venv
.\venv\Scripts\activate

Install pyMDG::

pip install pymdg

And finally run the generation::

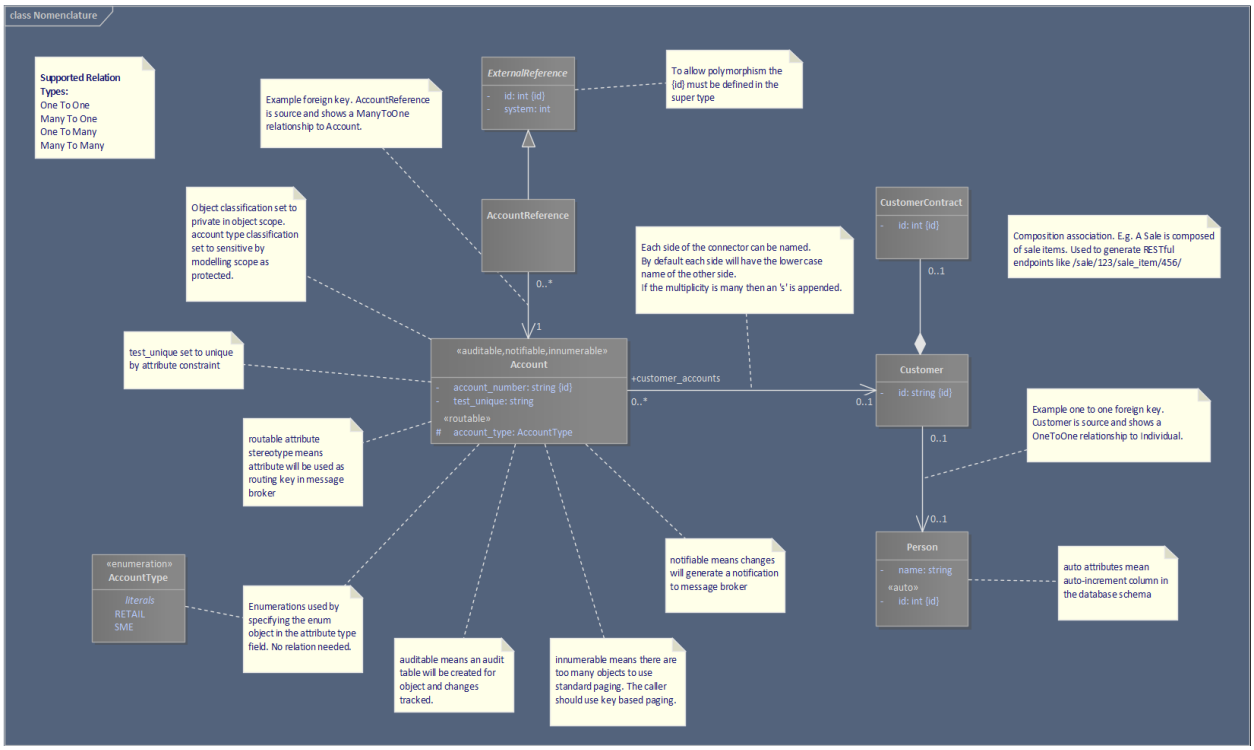
mdg-tool generate .\schemagen.yaml

2022-08-07 20:08:36,476 | mdg.config | INFO | Config file loaded: .schemagen.yaml
2022-08-07 20:08:36,722 | mdg.parse.sparx_xmi | INFO | Parsing models
2022-08-07 20:08:36,724 | mdg.parse | INFO | Base Model Package: Tutorial Data Model
2022-08-07 20:08:36,724 | mdg.generate.render | INFO | Generating model output for package /Tutorial Data Model/

3. Bask in our own pure awesomeness

We should find a build folder created and inside are 2 directories: openapi and avro with our generated artifacts.
Copy the openapi file contents and paste into https://editor.swagger.io/
You should be able to see the design decisions around endpoints only for non-abstract classes and to have definitions for lists, simple objects and full objects (the difference between simple and full is the inclusion of nested basic objects - see PartyRole for example).

## 3.7 Nomenclature

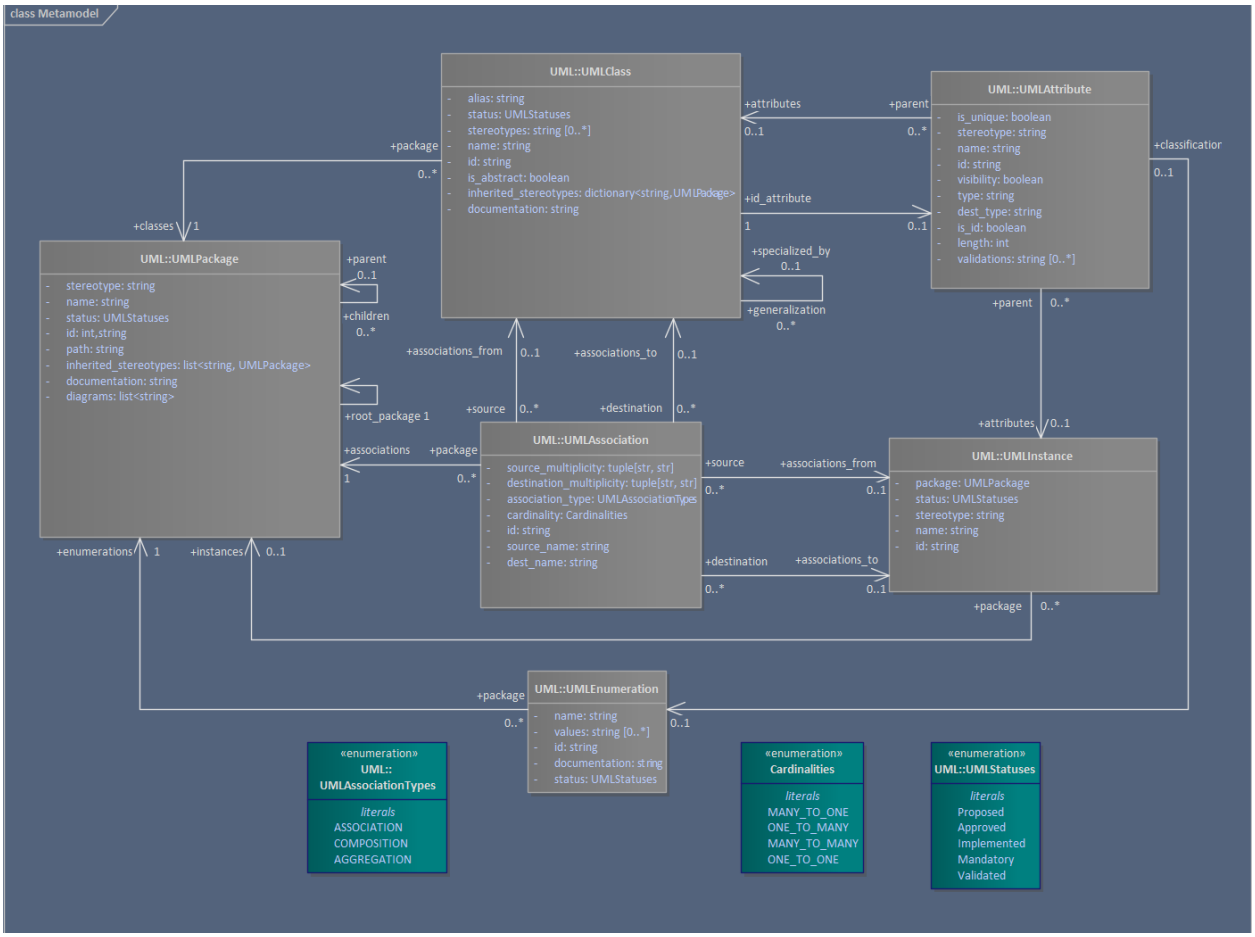



pyMDG supports 4 relation types:

- Association: Forms relations between classes:
  - One to One
  - One to Many
  - Many to One
  - Many to Many
- Generalization: Defines a parent/child inheritance. Multi-inheritance in not yet supported.
- Composition: Similar to association but can be used to control generated features. For example, when generating an OpenAPI schema, objects which are part of a composition may not get thier own endpoint.
- Aggregation: Like composition aggregations affect the features of our output. For example, an OpenAPI generation can use aggregations to specifiy when the endpoint is a sub-endpoint I.e. /customer/12/customer_address/1/

## 3.8 Metamodel

When pyMDG parses the source model file it parses into specific python classes.




This will form a hierarchy which looks like:

```
UMLPackage: my model package
    UMLPackage: sub package 1
```

(continues on next page)

(continued from previous page)

```
        classes:
            UMLClass: my class 1
                attributes:
                    UMLAttribute: my attribute 1
                    UMLAttribute: my attribute 2
            UMLClass: my class 2
            ...
    UMLPackage: sub package 2
    ...
```

The your projects config file you specify the level of the hierarchy to provide to the template rendering engine. There are 3 levels of rendering templates - root, package and class.

See Templates page for details on using the parsed classes

## 3.9 mdg

### 3.9.1 mdg package

#### Subpackages

#### mdg.generate package

#### Subpackages

#### mdg.generate.confluence package

#### Submodules

#### mdg.generate.confluence.content_update module

mdg.generate.confluence.content_update.**update**(*auth*, *content_id*, *path*)

#### mdg.generate.confluence.image_update module

mdg.generate.confluence.image_update.**update_images**(*auth*, *content_id*, *path*)

#### mdg.generate.confluence.util module

mdg.generate.confluence.util.**basic_auth**(*user*, *token*)

mdg.generate.confluence.util.**bearer_auth**(*api_client_id*, *api_client_secret*)

mdg.generate.confluence.util.**http_get_json**(*url*, *auth*, *session=None*)
A wrapper around the requests get function which will call SH and load the json response.

mdg.generate.confluence.util.**http_post**(*url*, *auth*, *data=None*, *file=None*, *session=None*)

mdg.generate.confluence.util.**http_put**(*url*, *auth*, *data=None*, *file=None*, *session=None*)

### Module contents

### Submodules

### mdg.generate.render module

mdg.generate.render.**output_level_assoc**(*source_template: jinja2.environment.Template, dest_file_template: jinja2.environment.Template, filter_template: Optional[jinja2.environment.Template], package: mdg.uml.UMLPackage*)
Render a jinja template for each association in the supplied package

mdg.generate.render.**output_level_class**(*source_template: jinja2.environment.Template, dest_file_template: jinja2.environment.Template, filter_template: Optional[jinja2.environment.Template], package: mdg.uml.UMLPackage*) → None
Render a jinja template for each class in the supplied package

mdg.generate.render.**output_level_copy**(*source_filename: str, dest_file_template: jinja2.environment.Template, package: mdg.uml.UMLPackage*) → None
Render a jinja template as pass a UML package as data

mdg.generate.render.**output_level_enum**(*source_template: jinja2.environment.Template, dest_file_template: jinja2.environment.Template, filter_template: Optional[jinja2.environment.Template], package: mdg.uml.UMLPackage*) → None
Render a jinja template for each enumeration in the supplied package

mdg.generate.render.**output_level_package**(*source_template: jinja2.environment.Template, dest_file_template: jinja2.environment.Template, package: mdg.uml.UMLPackage*) → None
Render a jinja template as pass a UML package as data

mdg.generate.render.**output_model**(*package: mdg.uml.UMLPackage*) → None
Loops through model templates in the settings and calls render functions

**Each template consists of:** dest: The filename of the output level: Do we generate a file for each package/class/enumeration/association or root source: Path to the jinja2 template filter: If supplied, If supplied The template must output "True" for a file to be generated. E.g. "{% if package.classes %}True{% else %}False{% endif %}"

mdg.generate.render.**output_test_cases**(*test_cases: List[mdg.uml.UMLInstance]*) → None
Test cases are parse into a list of UML instances. Loop through list and serialise

mdg.generate.render.**serialize_instance**(*instance: mdg.uml.UMLInstance*)
Generates a dictionary of attributes, values, dicts and lists from UML instance Recurses through associations originaing from supplied instance to serialise sub-objects

### Module contents

mdg.generate.**generate**()
Loads XMI file from settings as an etree Calls XMI parser to turn model and tests into python native (see UML metamodel) Calls output functions to render for model and tests

### mdg.parse package

### Submodules

### mdg.parse.bouml_xmi module

mdg.parse.bouml_xmi.**association_parse**(*package*, *source_element*, *dest_element*, *source*, *dest*)

mdg.parse.bouml_xmi.**attr_parse**(*parent: mdg.uml.UMLClass*, *element*, *root*)

mdg.parse.bouml_xmi.**class_parse**(*package*, *element*, *root*)

mdg.parse.bouml_xmi.**enumeration_parse**(*package*, *element*)

mdg.parse.bouml_xmi.**instance_parse**(*package*, *source_element*, *root*)

mdg.parse.bouml_xmi.**model_package_parse_inheritance**(*package*)
Looks for classes which are specializations of a supertype and finds the correct object

mdg.parse.bouml_xmi.**package_parse**(*element*, *root_element*, *parent_package*)
Extract package details, call class parser for classes and self parser for sub-packages. Associations are not done here, but in a 2nd pass using the parse_associations function. :param element: :param root_element: :return: :rtype: UMLPackage

mdg.parse.bouml_xmi.**package_parse_associations**(*package*, *element*, *root_element*)
Packages and classes should already have been parsed so now we link classes for each association. This gets messy as XMI output varies based on association type. This supports both un-specified and source to destination directional associations :param root_element: :param element: XML Element :type package: UMLPackage

mdg.parse.bouml_xmi.**package_parse_children**(*element*, *package*)

mdg.parse.bouml_xmi.**package_sort_classes**(*package*)

mdg.parse.bouml_xmi.**parse_test_cases**(*package*) → List[mdg.uml.UMLInstance]
Looks through package hierarchy for instances with request or response stereotype and returns list of instances. :rtype: list<UMLInstance>

mdg.parse.bouml_xmi.**parse_uml**() → Tuple[mdg.uml.UMLPackage, List[mdg.uml.UMLInstance]]
Root package parser entry point.

mdg.parse.bouml_xmi.**test_package_parse_inheritance**(*test_package*, *model_package*)
Links instances with the class they are instances of

### mdg.parse.drawio_xml module

mdg.parse.drawio_xml.**association_parse**(*package: mdg.uml.UMLPackage*, *element*, *root*)

mdg.parse.drawio_xml.**attr_parse**(*parent: mdg.uml.UMLClass*, *element*, *root*, *stereotypes*) → mdg.uml.UMLAttribute

mdg.parse.drawio_xml.**class_parse**(*package: mdg.uml.UMLPackage*, *element*, *root*) → mdg.uml.UMLClass

mdg.parse.drawio_xml.**enumeration_link**(*package*)

mdg.parse.drawio_xml.**enumeration_parse**(*package: mdg.uml.UMLPackage*, *element*, *root*) → mdg.uml.UMLEnumeration

mdg.parse.drawio_xml.**find_enumeration_by_name**(*package*, *name*)

mdg.parse.drawio_xml.**find_label_name**(*element*, *name*)

mdg.parse.drawio_xml.**generalization_parse**(*package: mdg.uml.UMLPackage*, *element*, *root*)

mdg.parse.drawio_xml.**get_label_name**(*element*)

mdg.parse.drawio_xml.**package_parse**(*element*, *root_element*, *parent_package: Optional[mdg.uml.UMLPackage]*) → mdg.uml.UMLPackage

Extract package details, call class parser for classes and self parser for sub-packages. Association linking is not done here, but in a 2nd pass using the parse_associations function.

mdg.parse.drawio_xml.**parse_uml**() → Tuple[mdg.uml.UMLPackage, List[mdg.uml.UMLInstance]]

### mdg.parse.sparx_db module

mdg.parse.sparx_db.**association_parse**(*session*, *tconnector: mdg.parse.sparx_db_models.TConnector*, *package: mdg.uml.UMLPackage*, *source: Union[mdg.uml.UMLClass, mdg.uml.UMLInstance]*, *dest: Union[mdg.uml.UMLClass, mdg.uml.UMLInstance]*)

mdg.parse.sparx_db.**attr_parse**(*session*, *parent: mdg.uml.UMLClass*, *tattribute: mdg.parse.sparx_db_models.TAttribute*)

mdg.parse.sparx_db.**class_parse**(*session*, *package: mdg.uml.UMLPackage*, *tobject: mdg.parse.sparx_db_models.TObject*)

mdg.parse.sparx_db.**enumeration_parse**(*session*, *package: mdg.uml.UMLPackage*, *tobject: mdg.parse.sparx_db_models.TObject*)

mdg.parse.sparx_db.**instance_parse**(*session*, *package: mdg.uml.UMLPackage*, *tobject: mdg.parse.sparx_db_models.TObject*)

mdg.parse.sparx_db.**package_parse**(*session*, *tpackage: mdg.parse.sparx_db_models.TPackage*, *parent_package: Optional[mdg.uml.UMLPackage]*)

Extract package details, call class parser for classes and self parser for sub-packages. Associations are not done here, but in a 2nd pass using the parse_associations function. :param element: :param root_element: :return: :rtype: UMLPackage

mdg.parse.sparx_db.**package_parse_associations**(*session*, *package: mdg.uml.UMLPackage*)

Packages and classes should already have been parsed so now we link classes for each association.

mdg.parse.sparx_db.**package_parse_children**(*session*, *package: mdg.uml.UMLPackage*)

mdg.parse.sparx_db.**package_sort_classes**(*package*)

mdg.parse.sparx_db.**parse_test_cases**(*package: mdg.uml.UMLPackage*) → List[mdg.uml.UMLInstance]

Looks through package hierarchy for instances with request or response stereotype and returns list of instances.

mdg.parse.sparx_db.**parse_uml**() → Tuple[mdg.uml.UMLPackage, List[mdg.uml.UMLInstance]]

Root package parser entry point.

mdg.parse.sparx_db.**test_package_parse_inheritance**(*test_package*, *model_package*)

Links instances with the class they are instances of and sets the attribute types which wern't in the run_state where instance attrs are created from

### mdg.parse.sparx_db_models module

**class** mdg.parse.sparx_db_models.**TAttribute**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

**Default**

**allowduplicates**

**classifier**

**const**

**container**

**containment**

**derived**

**ea_guid**

**genoption**

**id**

**iscollection**

**isordered**

**isstatic**

**length**

**lowerbound**

**name**

**notes**

**object_id**

**pos**

**precision**

**scale**

**scope**

**stereotype**

**style**

**styleex**

**type**

**upperbound**

**class** mdg.parse.sparx_db_models.**TAttributeconstraint**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

**Constraint**

**attname**

**id**

**notes**

**object_id**

**type**

**class** mdg.parse.sparx_db_models.**TConnector**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

**actionflags**

**btm_end_label**

**btm_mid_label**

**btm_start_label**

**connector_id**

**connector_type**

**destaccess**

**destcard**

**destchangeable**

**destconstraint**

**destcontainment**

**destelement**

**destisaggregate**

**destisnavigable**

**destisordered**

**destqualifier**

**destrole**

**destrolenote**

**destroletype**

**deststereotype**

**deststyle**

**destts**

**diagramid**

**direction**

**dispatchaction**

**ea_guid**

**end_edge**

**end_object_id**

**eventflags**

**headstyle**

**isbold**

**isleaf**

**isroot**

**issignal**

**isspec**

**isstimulus**

**linecolor**

**linestyle**

**linkaccess**

**name**

**notes**

**pdata1**

**pdata2**

**pdata3**

**pdata4**

**pdata5**

**ptendx**

**ptendy**

**ptstartx**

**ptstarty**

**routestyle**

**seqno**

**sourceaccess**

**sourcecard**

**sourcechangeable**

**sourceconstraint**

**sourcecontainment**

**sourceelement**

**sourceisaggregate**

**sourceisnavigable**

**sourceisordered**

**sourcequalifier**

**sourcerole**

**sourcerolenote**

**sourceroletype**

**sourcestereotype**

**sourcestyle**

**sourcets**

**start_edge**

**start_object_id**

**stateflags**

**stereotype**

**styleex**

**subtype**

**target2**

**top_end_label**

**top_mid_label**

**top_start_label**

**virtualinheritance**

**class** mdg.parse.sparx_db_models.**TObject**(***kwargs*)

Bases: sqlalchemy.orm.decl_api.Base

**abstract**

**actionflags**

**alias**

**author**

**backcolor**

**bordercolor**

**borderstyle**

**borderwidth**

**cardinality**

**classifier**

**classifier_guid**

**complexity**

**concurrency**

**createddate**

**diagram_id**

**ea_guid**

**effort**

**eventflags**

**fontcolor**

**genfile**

**genlinks**

**genoption**

**gentype**

**header1**

**header2**

**isactive**

**isleaf**

**isroot**

**isspec**

**modifieddate**

**multiplicity**

**name**

**note**

**ntype**

**object_id**

**object_type**

**package_id**

**packageflags**

**parentid**

**pdata1**

**pdata2**

**pdata3**

**pdata4**

**pdata5**

**persistence**

**phase**

**runstate**

**scope**

**stateflags**

**status**

**stereotype**

**style**

**styleex**

**tagged**

**tpos**

**version**

**visibility**

**class** mdg.parse.sparx_db_models.**TPackage**(***kwargs*)

Bases: sqlalchemy.orm.decl_api.Base

**batchload**

**batchsave**

**codepath**

**createddate**

**ea_guid**

**iscontrolled**

**lastloaddate**

**lastsavedate**

**logxml**

**modifieddate**

**name**

**namespace**

**notes**

**package_id**

**packageflags**

**parent_id**

**pkgowner**

**protected**

**tpos**

**umlversion**

**usedtd**

**version**

**xmlpath**

**class** mdg.parse.sparx_db_models.**TXref**(***kwargs*)

Bases: sqlalchemy.orm.decl_api.Base

**Constraint**

**behavior**

**client**

**description**

**link**

**name**

**namespace**

**partition**

**requirement**

**supplier**

**type**

**visibility**

**xrefid**

### mdg.parse.sparx_xmi module

mdg.parse.sparx_xmi.**association_parse**(*package*, *source_element*, *dest_element*, *source*, *dest*)

mdg.parse.sparx_xmi.**attr_parse**(*parent: Union[mdg.uml.UMLClass, mdg.uml.UMLEnumeration, mdg.uml.UMLComponent], element, root*)

mdg.parse.sparx_xmi.**class_parse**(*package*, *element*, *root*)

mdg.parse.sparx_xmi.**enumeration_parse**(*package*, *element*)

mdg.parse.sparx_xmi.**instance_parse**(*package*, *source_element*, *root*)

mdg.parse.sparx_xmi.**model_package_parse_inheritance**(*package*)
Looks for classes which are specializations of a supertype and finds the correct object

mdg.parse.sparx_xmi.**package_parse**(*element*, *root_element*, *parent_package*)
Extract package details, call class parser for classes and self parser for sub-packages. Associations are not done here, but in a 2nd pass using the parse_associations function. :param element: :param root_element: :return: :rtype: UMLPackage

mdg.parse.sparx_xmi.**package_parse_associations**(*package*, *element*, *root_element*)
Packages and classes should already have been parsed so now we link classes for each association. This gets messy as XMI output varies based on association type. This supports both un-specified and source to destination directional associations :param root_element: :param element: XML Element :type package: UMLPackage

mdg.parse.sparx_xmi.**package_parse_children**(*element*, *package*)

mdg.parse.sparx_xmi.**package_sort_classes**(*package*)

mdg.parse.sparx_xmi.**parse_test_cases**(*package*) → List[mdg.uml.UMLInstance]
Looks through package hierarchy for instances with request or response stereotype and returns list of instances. :rtype: list<UMLInstance>

mdg.parse.sparx_xmi.**parse_uml**() → Tuple[mdg.uml.UMLPackage, List[mdg.uml.UMLInstance]]
Root package parser entry point.

mdg.parse.sparx_xmi.**test_package_parse_inheritance**(*test_package*, *model_package*)
Links instances with the class they are instances of

### Module contents

**exception** mdg.parse.**ParseError**
Bases: Exception

mdg.parse.**parse**()
Calls parser to turn model and tests into python native (see UML metamodel)

### mdg.tests package

### Submodules

### mdg.tests.test_dumps module

**class** mdg.tests.test_dumps.**TestDumps**(*methodName='runTest'*)
Bases: unittest.case.TestCase

**setUp**()
Hook method for setting up the test fixture before exercising it.

**test_object_serialise**()

### mdg.tests.test_instance module

**class** mdg.tests.test_instance.**TestInstanceOutput**(*methodName='runTest'*)
Bases: unittest.case.TestCase

**setUp**()
Hook method for setting up the test fixture before exercising it.

**test_object_serialise**()

### mdg.tests.test_sample module

**class** mdg.tests.test_sample.**TestSample_DrawIO_Django**(*methodName='runTest'*)
Bases: unittest.case.TestCase

**setUp**()
Hook method for setting up the test fixture before exercising it.

**test_parse**()

**test_render_model**()

**test_validate**()

### mdg.tests.test_tools module

**class** mdg.tests.test_tools.**TestIO**(*methodName='runTest'*)
Bases: unittest.case.TestCase

**setUp**()
Hook method for setting up the test fixture before exercising it.

**test_to_dict**()

**test_to_obj**()

**class** mdg.tests.test_tools.**TestUMLModel**(*methodName='runTest'*)
Bases: unittest.case.TestCase

**setUp**()
Hook method for setting up the test fixture before exercising it.

**test_camel**()

**test_snake**()

**class** mdg.tests.test_tools.**blort**
Bases: object

**class Meta**
Bases: object

**owned_subobjects = 'my_list'**

**class** mdg.tests.test_tools.**foo**
Bases: object

**class Meta**
Bases: object

**id_field = 'id'**

### mdg.tests.test_uml_model module

**class** mdg.tests.test_uml_model.**TestUMLModel**(*methodName='runTest'*)
Bases: unittest.case.TestCase

**setUp**()
Hook method for setting up the test fixture before exercising it.

**test_association_cardinality**()

**test_association_type**()

**test_attribute_get_type**()

**test_attribute_type**()

**test_find_class**()

**test_find_package**()

**test_string_to_multiplicity**()

### mdg.tests.test_xmi_model_parse module

**class** mdg.tests.test_xmi_model_parse.**TestXMIAssociationParse**(*methodName='runTest'*)
Bases: unittest.case.TestCase

**setUp**()
Hook method for setting up the test fixture before exercising it.

**test_association_parse**()

**class** mdg.tests.test_xmi_model_parse.**TestXMIAttributeParse**(*methodName='runTest'*)
Bases: unittest.case.TestCase

**setUp**()
Hook method for setting up the test fixture before exercising it.

**test_attr_parse**()

**class** mdg.tests.test_xmi_model_parse.**TestXMIClassParse**(*methodName='runTest'*)
Bases: unittest.case.TestCase

**setUp**()
Hook method for setting up the test fixture before exercising it.

**test_class_parse**()

**class** mdg.tests.test_xmi_model_parse.**TestXMIGeneralizationParse**(*methodName='runTest'*)
Bases: unittest.case.TestCase

**setUp**()
Hook method for setting up the test fixture before exercising it.

**test_generalization_parse**()

### Module contents

### mdg.tools package

### Submodules

### mdg.tools.case module

String convert functions

mdg.tools.case.**alphanumcase**(*string: str*) → str
Cuts all non-alphanumeric symbols, i.e. cuts all expect except 0-9, a-z and A-Z.

**Args:** string: String to convert.

**Returns:** string: String with cutted non-alphanumeric symbols.

mdg.tools.case.**backslashcase**(*string: str*) → str
Convert string into spinal case. Join punctuation with backslash.

**Args:** string: String to convert.

**Returns:** string: Spinal cased string.

mdg.tools.case.**camelcase**(*string*)

mdg.tools.case.**capitalcase**(*string: str*) → str
Convert string into capital case. First letters will be uppercase.

**Args:** string: String to convert.

**Returns:** string: Capital case string.

mdg.tools.case.**constcase**(*string: str*) → str
Convert string into upper snake case. Join punctuation with underscore and convert letters into uppercase.

**Args:** string: String to convert.

**Returns:** string: Const cased string.

mdg.tools.case.**dotcase**(*string: str*) → str
Convert string into dot case. Join punctuation with dot.

**Args:** string: String to convert.

**Returns:** string: Dot cased string.

mdg.tools.case.**lowercase**(*string: str*) → str
Convert string into lower case.

**Args:** string: String to convert.

**Returns:** string: Lowercase case string.

mdg.tools.case.**pascalcase**(*string: str*) → str
Convert string into pascal case.

**Args:** string: String to convert.

**Returns:** string: Pascal case string.

mdg.tools.case.**pathcase**(*string: str*) → str
Convert string into path case. Join punctuation with slash.

**Args:** string: String to convert.

**Returns:** string: Path cased string.

mdg.tools.case.**sentencecase**(*string: str*) → str
Convert string into sentence case. First letter capped and each punctuations are joined with space.

**Args:** string: String to convert.

**Returns:** string: Sentence cased string.

mdg.tools.case.**snakecase**(*string*)

mdg.tools.case.**spinalcase**(*string: str*) → str
Convert string into spinal case. Join punctuation with hyphen.

**Args:** string: String to convert.

**Returns:** string: Spinal cased string.

mdg.tools.case.**titlecase**(*string: str*) → str
Convert string into sentence case. First letter capped while each punctuations is capitalsed and joined with space.

**Args:** string: String to convert.

**Returns:** string: Title cased string.

mdg.tools.case.**trimcase**(*string: str*) → str
Convert string into trimmed string.

**Args:** string: String to convert.

**Returns:** string: Trimmed case string

mdg.tools.case.**uppercase**(*string: str*) → str
Convert string into upper case.

**Args:** string: String to convert.

**Returns:** string: Uppercase case string.

### mdg.tools.filters module

mdg.tools.filters.**get_filters**()

### mdg.tools.io module

mdg.tools.io.**dict_to_obj**(*input: dict*, *base_object_class*, *references: dict = {}*) → object

mdg.tools.io.**dict_to_obj_pass1**(*input: dict*, *base_object_class*, *references: dict = {}*) → Tuple[Any, dict]
Creates objects from input dictionary. Pass 1 keeps references and compiles a list of objects with ids used in pass 2.

mdg.tools.io.**dict_to_obj_pass2**(*obj: Any*, *references: dict*) → None
Pass 2 walks through objects and replaces attributes which are references with the actual class.

mdg.tools.io.**obj_to_dict**(*obj: Any*, *base_dict={}*) → dict
Creates nested dictionaries from input object

### mdg.tools.mdg_tool module

mdg.tools.mdg_tool.**dumps**(*args*)

mdg.tools.mdg_tool.**generate**(*args*)

mdg.tools.mdg_tool.**main**()

mdg.tools.mdg_tool.**startproject**(*args*)

mdg.tools.mdg_tool.**validate**(*args*)

### mdg.tools.schema module

### mdg.tools.sparx_db_models_raw module

**class** mdg.tools.sparx_db_models_raw.**TObject**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

**abstract**

**actionflags**

**alias**

**author**

**backcolor**

**bordercolor**

**borderstyle**

**borderwidth**

**cardinality**

**classifier**

**classifier_guid**

**complexity**

**concurrency**

**createddate**

**diagram_id**

**ea_guid**

**effort**

**eventflags**

**fontcolor**

**genfile**

**genlinks**

**genoption**

**gentype**

**header1**

**header2**

**isactive**

**isleaf**

**isroot**

**isspec**

**modifieddate**

**multiplicity**

**name**

**note**

**ntype**

**object_id**

**object_type**

**package_id**

**packageflags**

**parentid**

**pdata1**

**pdata2**

**pdata3**

**pdata4**

**pdata5**

**persistence**

**phase**

**runstate**

**scope**

**stateflags**

**status**

**stereotype**

**style**

**styleex**

**tagged**

**tpos**

**version**

**visibility**

**class** mdg.tools.sparx_db_models_raw.**TPackage**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

**batchload**

**batchsave**

**codepath**

**createddate**

**ea_guid**

**iscontrolled**

**lastloaddate**

**lastsavedate**

**logxml**

**modifieddate**

**name**

**namespace**

**notes**

**package_id**

**packageflags**

**parent_id**

**pkgowner**

**protected**

**tpos**

**umlversion**

**usedtd**

**version**

**xmlpath**

### mdg.tools.validate module

**class** mdg.tools.validate.**AttributeValidationError**(*package*, *cls*, *attr*, *error*)
Bases: object

**class** mdg.tools.validate.**ClassValidationError**(*package*, *cls*, *error*)
Bases: object

mdg.tools.validate.**validate**()
Loads XMI file from settings as an etree Calls XMI parser to turn model and tests into python native (see UML metamodel) Calls output functions to render for model and tests

mdg.tools.validate.**validate_package**(*package*)

### Module contents

### mdg.uml package

### Module contents

**class** mdg.uml.**Cardinality**
Bases: enum.Enum

An enumeration.

**MANY_TO_MANY = 3**

**MANY_TO_ONE = 1**

**ONE_TO_MANY = 2**

**ONE_TO_ONE = 4**

**class** mdg.uml.**UMLAssociation**(*package: mdg.uml.UMLPackage, source: Union[mdg.uml.UMLClass, mdg.uml.UMLInstance, mdg.uml.UMLComponent], destination: Union[mdg.uml.UMLClass, mdg.uml.UMLInstance, mdg.uml.UMLComponent], id: Union[int, str], assoc_type=<UMLAssociationType.ASSOCIATION: 1>*)
Bases: object

**class Meta**
Bases: object

**id_field = 'id'**

**owned_subobjects = []**

**cardinality**

**string_to_multiplicity**(*value*)

**class** mdg.uml.**UMLAssociationType**
Bases: enum.Enum

An enumeration.

**AGGREGATION = 3**

**ASSOCIATION = 1**

**COMPOSITION = 2**

**class** mdg.uml.**UMLAttribute**(*parent: Union[mdg.uml.UMLClass, mdg.uml.UMLEnumeration, mdg.uml.UMLComponent, mdg.uml.UMLInstance], name: str, id: Union[int, str]*)
Bases: object

**class Meta**
Bases: object

**id_field = 'id'**

**owned_subobjects = []**

**get_name**() → str

**get_type**(*translator: Optional[str] = None*) → str

**set_type**(*source_type: str*)

**class** mdg.uml.**UMLClass**(*package: mdg.uml.UMLPackage, name: str, id: Union[int, str]*)
Bases: object

**class Meta**
Bases: object

**id_field = 'id'**

**owned_subobjects = ['attributes']**

**get_name**() → str

**class** mdg.uml.**UMLComponent**(*package: mdg.uml.UMLPackage, name: str, id: Union[int, str]*)
Bases: object

**class Meta**
Bases: object

**id_field = 'id'**

**owned_subobjects = ['attributes']**

**class** mdg.uml.**UMLEnumeration**(*package: mdg.uml.UMLPackage, name: str, id: Union[int, str]*)
Bases: object

**class Meta**
Bases: object

**id_field = 'id'**

**owned_subobjects = []**

**class** mdg.uml.**UMLInstance**(*package: mdg.uml.UMLPackage, name: str, id: Union[int, str]*)
Bases: object

**class Meta**
Bases: object

**id_field = 'id'**

**owned_subobjects = ['attributes']**

**class** mdg.uml.**UMLPackage**(*id: Union[int, str], name: str, parent: Optional[mdg.uml.UMLPackage] = None, stereotype: Optional[str] = None*)
Bases: object

**class Meta**
Bases: object

**id_field = 'id'**

**owned_subobjects = ['classes', 'associations', 'children', 'enumerations', 'compone**

**add_child**(*id: Union[int, str], name: str, stereotype=None*) → mdg.uml.UMLPackage

**find_by_id**(*id: Union[int, str], find_type: Optional[str] = None*)
Finds UMLPackage, UMLClass, UMLEnumeration or UMLInstance object with specified Id Looks for classes part of this package and all sub-packages

**class** mdg.uml.**UMLStatuses**
Bases: enum.Enum

An enumeration.

**Approved = 2**

**Implemented = 3**

**Mandatory = 4**

**Proposed = 1**

**Validated = 5**

mdg.uml.**dumps**(*package: mdg.uml.UMLPackage*) → str

### Submodules

### mdg.config module

mdg.config.**load**()

### Module contents

# Python Module Index

## m

# Index

## C



## D

## E



## F



## G



## H

## I



## L



## M

## N



## O

## P

## R



## S

## T

## U

## V



## X