# Monthly Energy Balance (MEB)

# Technical Document

*Hadise Rasoulian, Guillermo Gutierrez Morote*

January 2024

# Contents

**1 MEB Overview** **2**
1.1 Requirements . . . 2
1.2 Summary of the process . . . 2

**2 Contribution to the Hub development; using MEB tool as an example** **5**
2.1 Geometry factory . . . 5
2.2 Dictionaries . . . 6
2.3 Enriching the city simulation and importing the results . . . 6
2.4 Construction factory class . . . 7
2.5 Result factory . . . 7
2.6 SRA engine . . . 7
2.7 Usage Factory . . . 7
2.8 Energy system factory . . . 8
2.9 MEB engine . . . 8

**Appendices** **9**

**A Importers** **9**
A.1 Adding a new Import factory . . . 9

**B Levels of Detail** **11**

**C Exporters** **12**
C.1 Adding a new exporter factory . . . 12

**D Geometry Factory** **13**

**E Catalogs** **17**
E.1 Construction catalog factory . . . 18
E.2 Cost catalog factory . . . 21
E.3 Energy systems catalog factory . . . 23
E.4 Usage catalog factory . . . 25
E.5 Creating a new catalog . . . 28
E.5.1 Creating a new catalog factory . . . 30

# 1 MEB Overview

This document aims to present the Hub, its modules and its interconnections, so that will (a) learn how to create a workflow using the Hub, (b) understand what the available options are for each module, and (c) know how to modify or add new features to it.

The document uses the Monthly Energy Balance (MEB) workflow as a guide to lead you through the nooks and crannies of the software.

Please note that this document is written based on the `MEB tutorial`, and some parts may be repetitive. The primary purpose of the MEB workflow is to calculate building energy demands and consumption at city scale, enabling city planners and decision-makers to explore viable retrofitting options to optimize energy efficiency and reduce carbon emissions.

The MEB workflow calculates the monthly energy balance for each building in a given region, and as a result, it provides the energy and electrical demand, the energy consumption and the peak loads for heating, cooling, lighting and appliances.

All of these results are provided on both a monthly and a yearly basis.

In the design of this workflow, the following assumptions are made:

- No building in the region has attics
- All buildings have basements but those are not heated
- The construction details come from NRCAN
- The usage details come from NRCAN
- The system details come from the Montreal Custom catalog
- The weather file used is the epw of the area provided by EnergyPlus

## 1.1 Requirements

You will need to install the following dependencies on your computer to run the software; please look at the install process for your system-specific details.

1. INSEL 8.
2. Simplified Radiosity Algorithm (SRA).

To get access to these tools, please contact guillermo.gutierrezmorote@concordia.ca

## 1.2 Summary of the process

We will go through each part of the MEB workflow, look in depth at the classes involved, and investigate how to create new classes, objects, dictionaries, etc.
We'll use seven factories in the **Monthly Energy Balance** code. Six of them enrich the city, and the last one exports information from the city to third party tools. These factories are:

- Geometry factory
- Usage factory
- Construction factory
- System factory

- Weather factory
- Result factory
- Energy building exports factory

The process of calculating the energy consumption is depicted in Figure 1.
/ghPlease change the figure 1 so the sra go right after the geometry factory :)

Please note that the diagram, shows the enrichment steps at the point where they are required to allow next steps (for instance SRA can be done just after create the city), but in the code all are located at the beginning for a cleaner and clearer code organization.

The Monthly Energy Balance workflow follows a series of steps to assess the energy demand and consumption of the city:

- **Geometry Factory** creates the city object containing the buildings.
- **Construction Factory** enriches them with construction data.
- **SRA Engine** calculates the short-wave radiation on the external surfaces of the buildings.
- **Result Factory** enriches the city with those results.
- **Usage and Systems Factories** incorporates usage and energy systems data respectively.
- **INSEL Engine** calculates the monthly building demands separated into heating, cooling, domestic hot water, lighting and other electrical appliances, together with the peak loads of those demands for each month of the year. It also calculates the energy consumption of the systems that cover heating, cooling and domestic hot water demands, and the onsite electrical production in the case that the system includes PV.

The following code is extracted from the MEB tool and shows the steps followed in the workflow:

```
city = GeometryFactory('geojson',
                       path=file_path,
                       height_field='heightmax',
                       year_of_construction_field='ANNEE_CONS',
                       function_field='CODE_UTILI',
                       function_to_hub=Dictionaries().
  montreal_function_to_hub_function).city
ConstructionFactory(construction_format, city).enrich()
UsageFactory(usage_format, city).enrich()
EnergySystemsFactory(energy_systems_format, city).enrich()

SraEngine(city, tmp_folder)
MonthlyEnergyBalanceEngine(city, tmp_folder)
```

As mentioned, in this example, all the factories are placed at the beginning of the workflow and then the third-party tools are called. Two types of factories are presented: the Geometry Factory, which creates the city, and the rest that populate the city with additional information. All of them will be explained in the following sections.

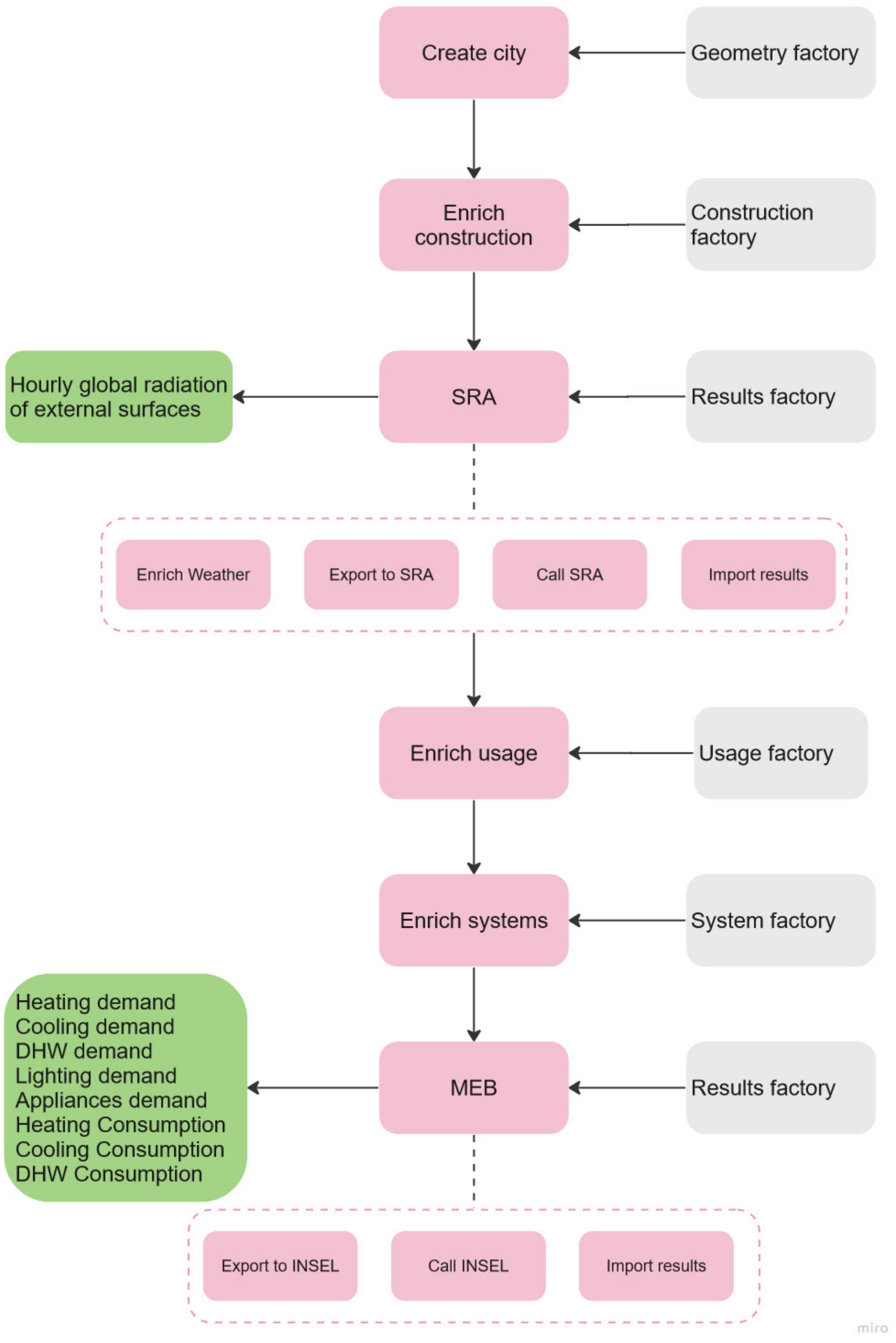

Figure 1: Caption

# 2 Contribution to the Hub development; using MEB tool as an example

The MEB tutorial is available at the this address. The MEB tutorial is a prerequisite of this manual. The current document is an extension and in-depth explanation of the MEB tool.

## 2.1 Geometry factory

The comprehensive detail of this factory is available at D.
`GeometryFactory` instantiates a specific geometric object to load a geometric file in a specific format (Eg: **'geojson'** at 1.2 format was selected). Then, the geometric module reads the input file and creates a city with the geometries contained in it.
The `GeometryFactory` has two mandatory parameters, the `file type`, and `path`. the rest are optional and provides additional configuration for specific callback functions.

An example of the method to call the factory is shown below.

```
    GeometryFactory(handler, path).city
```

Some geometry file formats have the ability to include metadata for the geometry information, like the year of construction and function, and the Geometry factory offers some optional parameters to configure the access to the fields containing the metadata.

When present, the optional parameters allow the developer to configure the data source and automatically save the metadata in the proper central data model properties.

For example, while `CityGML` has standard names defined for function and year of construction, `GeoJson` doesn't provide those, but it has a space for attributes, and in order to be able to locate the fields, the importer needs to know which names are used in the specific `GeoJson` file. Thus, this information must be provided through the arguments in the factory call.

In some cases, the metadata information may need to be transformed into Hub known values, and this is done via transformation **Dictionaries** containing the expected values in the input file and the corresponding constant value used by the Hub.

Please notice that even though the Hub provided several **standard** transformation dictionaries, it is also possible to use a user-defined one if needed (more about dictionaries can be found in section 2.2).

This factory has the following mandatory and optional arguments:

- `File type`: This is one of the two mandatory arguments of this class. It is a handler to select the proper importer and it works in the same way in all factories, regardless of its function. In this factory, the handler refers to the file format for an easier understanding. The available handlers are `CityGML`, `OBJ` and `GeoJSON`.
- `Path`: This is the second mandatory argument and it provides the full path of where to find the input file of the Geometry Factory.
- `Aliases field`: (Optional) Provides a way to add other information that could be interesting such as building name, address or postal code.
- `Function field`: (Optional) Indicates the field name for the building function. This function was defined by the one who created the geometry file and can be in any format(strings, numbers...), any language (English, French...), etc.
- `Height field`: (Optional) Indicates the field name containing the building height.
- `year of the construction field`: (Optional) Indicates the field name containing the building year of construction.

- `function field`: (Optional) Indicates the field name containing the building function.
- `Function to hub`: The Hub has a standard list of building functions. If applicable, this parameter changes the function to the Hub's standard. This is done through `dictionaries` that have been explained in 2.2. Dictionaries are defined to convert external information to Hub standards and vice versa.

Please notice that, for example, `CityGML` has standard names defined for function and year of construction, so the importer does not need extra information. It also provides a standard name for the height for LOD[1] 0.5 models, so this field is not needed either. However, `GeoJson` provides space for a list of attributes but without any standard naming, therefore the importer needs to know which names are used in each specific geojson file. Thus, this information must be provided through the arguments.

## 2.2 Dictionaries

The Hub includes Dictionaries (at **hub.helpers.dictionaries**) for translating external information into values understandable by the Hub ("Hub language") and from "Hub language" into external information.

For example, there exist some Dictionaries to translate from some geometry data sources building function into Hub building function:

1. `pluto_function_to_hub_function`: This dictionary converts the building classification of New York City to the hub standard.
2. `hft_function_to_hub_function`: This dictionary converts the building classification defined by Stuttgart Technology University to the hub standard.
3. `montreal_function_to_hub_function`: This dictionary converts the building classification of Montreal to the hub standard.
4. `alkis_function_to_hub_function`: This dictionary converts the building classification ALKIS building code to the hub standard.

Also, in order to access the information coming from catalogs (for example with the construction factory), the Hub provides Dictionaries that translate from Hub building function into the catalog building function:

1. `hub_function_to_nrcan_construction_function`: This dictionary converts the hub standard into the NRCAN standard.
2. `hub_function_to_nrel_construction_function`: This dictionary converts the hub standard into the NREL standard.

More information about the Dictionaries and creating them can be found in section D.

## 2.3 Enriching the city simulation and importing the results

Hub factories are divided into imports and exports. Import is used in the sense of importing values to the city object and storing those in the central data model, while export means extracting the data from the city and writing it in any other supported format.

The monthly energy balance workflow minimal requirements include construction and usage information to be **imported** (more about importers can be found in annex A) into the city before it can be **exported** (more about exporters can be found in annex C) to insel monthly energy balance format, then the python script can trigger the building simulations and finally the results can be **imported** into the city.

[1] For more information about CityGML LOD visit https://docs.ogc.org/DRAFTS/20-010.html#geometry-lod-section

## 2.4 Construction factory class

This factory, as well as the usage and result factories, are designed to enrich the city with new information associated with different topics. They reach specific categories-related catalogs and fill the city's buildings with data based on each building's archetypes.

In the same way as all factories, regardless of the type, **construction factory** has a mandatory attribute, the handler. Furthermore, this one, as well as all factories in charge of the enrichment, require the object city to be enriched as a second attribute. Finally, again as in all enrichment factories, it has a method called enrich. The construction factory is called as shown below.

```
ConstructionFactory(handler, city).enrich()
```

### Construction factory handlers

The construction factory uses the handler to know which catalog it must access. It can reach at the time being only two different catalogs, **NRCAN** and **NREL**. This factory first extracts the information from the source and feeds Hub catalogs with it. The Hub catalogs are designed to help with data understanding by providing a standardized structure. They homogenize the information from both catalogs into a fixed structure using international units. Secondly, the construction factory extracts the information from the catalogs to feed each building in the city with its specific archetype, defined by the function and year of construction of the building, and the climate zone to which it belongs. More information about the catalogs can be found in annex section E.

## 2.5 Result factory

The Result factory retrieves the specific tool results and stores the data in the given city. Three methods for this factory are available:

1. `SRA` that enrich the city with Simplified Radiosity Algorithm results.
2. `heat_pump`, which enriches the city (energy system specifically) with heat pump INSEL simulation.
3. `insel_monthly_energy_balance`, which enriches the city with insel monthly energy balance results.

## 2.6 SRA engine

The Simplified Radiosity Algorithm (SRA) engine performs calculations to determine the amount of radiation that falls on every surface of each building. The `SraEngine` class encapsulates the process of running the external SRA tool. It prepares data for the calculation, runs the software, and enriches the results. The input data should be in `xml` format. The engine needs three arguments, city, path of the xml file, and path for the output of the calculation.

```
sra_file = (tmp_folder / f'{city.name}_sra.xml').resolve()
SraEngine(city, sra_file, tmp_folder)
```

The `SraEngine` calls the SRA software and calculates the radiation on each surface. Then, the `result factory` class is used to enrich the SRA calculation results. This step involves processing and analyzing the calculated data to provide monthly and yearly peak loads for the buildings in the city.

## 2.7 Usage Factory

The Usage factory assigns the occupant behaviour, i.e. the detailed characterization of building usage, including people and facilities, to the buildings.
To enrich the city with **usage factory**, the usage catalog needs to be specified. The Hub supports schedules for Canada `(NRCAN)` and US `(COMNET)` described below.

- NRCAN catalog was created based on the usage archetypes defined by the Canadian Natural Resources agency, available at this address. The division of the occupancy density into convective, radiative and sensible fractions were not available; these values were extracted from the COMNET catalog to complete the needed information. For the time being it provides archetypes for 16 different usages that are those defined by the Building Technology Assessment Platform (BTAP).
- The Pacific Northwest National Laboratory on behalf of the United States Department of Energy provides the COMNET catalog that covers 33 building types and their usage parameters, including lighting power, plug loads, occupancy, etc. It can be downloaded from this link. Together with this file, a second one containing schedule information is also provided here. These schedules distribute the values of the previous parameters over 24 hours divided into 3 different day types.

The method to call the usage factory is shown below; It needs two parameters, the handler ('nrcan' or 'comnet'), and the object 'city'.

```
UsageFactory('nrcan' OR 'comnet', city).enrich()
```

## 2.8 Energy system factory

The energy system catalog factory enriches the city based on the city name and the handler. The information of this factory includes heating, cooling, and electrical systems in the buildings. Currently, the only available handler is `montreal_custom`. This handler loads data from an XML file, organizes it into appropriate structures, and provides methods to retrieve information about archetypes, systems, generation equipment, and distribution equipment from the catalog.

## 2.9 MEB engine

The last step of the MEB workflow is calling the `MonthlyEnergyBalanceEngine` to perform a monthly energy balance calculation based on the enriched data. The outputs are monthly heating, cooling, domestic hot water, lighting, and appliance demand.
The following steps are happening in the MEB engine:

- The `EnergyBuildingsExportsFactory` class exports the city into INSEL format using `insel_monthly_energy_balance` handler.
- The INSEL is being called and runs the model
- The MEB engine enriches the city using `result factory` with regards to the results of INSEL

# Appendices

## A Importers

The `Import` section of the hub includes the six factories; `Construction factory`, `Energy Systems factory`, `Geometry factory`, `Results factory`, `Usage factory`, and `Weather factory`. Each factory is an abstract class that starts with `__init__ method`. Importers are divided into two groups, the `Geometry factory` in one group and the rest of the importers in the other group. The main difference is that the second group aims to read data from several data sources and use it to enrich cities with relevant information, whilst the `geometry factory` creates an instance of the city based on pure geometry and metadata.

- The construction factory retrieves the specific construction module for the given region and enriches the city using `NREL` or `NRCAN` information.
- Energy systems factory retrieves the energy system module for the given region. Currently, the Hub supports three handlers including `air_source_hp, water_to_water_hp, and montreal_custom`. Two first handlers enrich the city by providing information of the heat pumps. The last one enriches the city by using Montreal's custom energy systems catalog information.
- Geometry factory retrieves the specific geometric module to load the given format. This class supports many formats; Citygml, obj and GeoJSON. New formats could and will be added to the `Geometry factory` in the future. A detailed description to this factory is provided in section D.
- Results factory retrieves the specific tool results and stores the data in the given city. Three methods for this factory are available: (a) `SRA` that enrich the city with Simplified Radiosity Algorithm results, (b) `heat_pump`, which enriches the city (energy system specifically) with heat pump INSEL simulation, and (c) `insel_monthly_energy_balance`, which enriches the city with insel monthly energy balance results.
- Usage factory enriches the city given to the class using one of the Usage factory's current handlers. Two available handlers are (a) `COMNET`, which enriches the city with COMNET usage library, and (b) `NRCAN` which enriches the city with NRCAN usage library.
- Weather factory enriches the city using a weather handler. The available weather format is energy plus weather file `(epw)`.

### A.1 Adding a new Import factory

A new import class should have a purpose; retrieve the specific module for the given source format. The new import class needs to have the `__init__` method to instantiate the object, at least one handler (for example, geojson and citygml in geometry factory, and comnet and nrcan in usage factory) to enrich the city with, and an enrich method to enrich the city given to the class using the given handler.

The structure of `Geometry factory` is slightly different to that of the other importers. While other importers enrich the city with provided information in their handler catalogs, the geometry factory creates a city object based on the information of the given handler.

In the `Geometry factory` class, `@property` decorators have been used to define methods that can be accessed like attributes. This approach provides a more intuitive way to access different types of geometry data without having to call explicit methods.

A new factory could be created similar to the geometry factory (using `@property` decorators) or similar to other types of factories.

An example of adding a new class named `NewFactory`, which has a similar structure to `Geometry factory`, is provided below:

```python
# Import required libraries

class NewFactory:
    """
    NewFactory class
    """
    def __init__(self, handler, city):
        self._handler = '_' + handler.lower()
        validate_import_export_type(NewFactory, handler)
        self._city = city

    @property
    def _handler1(self) -> City:
        """
        Description of handler1
        """
        # Your implementation here

    @property
    def _handler2(self) -> City:
        """
        Description of handler2
        """
        # Your implementation here

    # Add more handler functions as needed

    @property
    def city(self):
        """
        Enrich the city using the new factory handler
        :return: City
        """
        return getattr(self, self._handler, lambda: None)
```

In this code, the `City`, following each handler, returns an instance of the City.

An example of adding a new class named `NewFactory`, which has a similar structure to other factories than `Geometry factory`, is provided below:

```python
# Import required libraries

class NewFactory:
    """
    NewFactory class
    """
    def __init__(self, handler, city):
        self._handler = '_' + handler.lower()
        validate_import_export_type(NewFactory, handler)
        self._city = city

    def _handler1(self):
        """
        Description of handler1
        """
        # Your implementation here

    def _handler2(self):
        """
        Description of handler2
        """
        # Your implementation here

    # Add more handler functions as needed

    def enrich(self):
        """
        Enrich the city given to the class using the new factory given handler
        :return: None
        """
        getattr(self, self._handler, lambda: None)()
```

## B Levels of Detail

Data may have several details that could be useful or not depending on the purpose. Therefore, the Hub includes the concept of Level of Detail (LoD). We use a very generic classification system to divide up data into levels of detail, following the idea used by CityGML of details in geometry. In that sense, geometry is classified in 5 levels (figure 2), and it is also possible to have intermediate levels. For example, an LoD 0.5 in geometry would be providing footprint (as in LoD 0) together with the building height, giving an idea of its 3D exterior aspect.

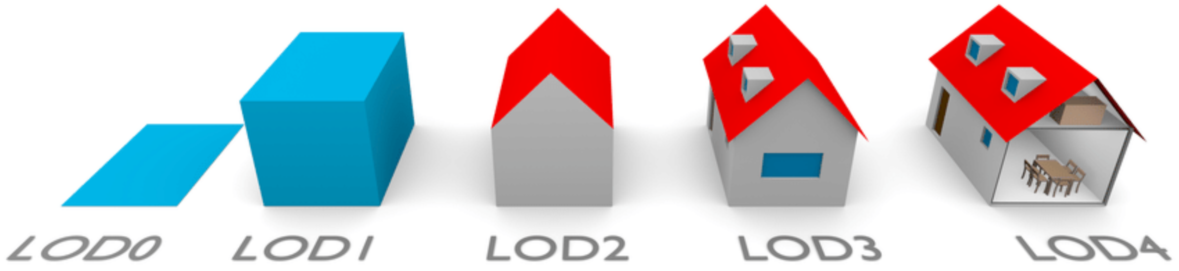

Figure 2: The five LODs of CityGML

Other classifications are, depending on the topic:

- Construction and Usage: LoD 1 should give enough information to allow static energy demand calculations, while LoD 2 must allow dynamic simulations.
- Energy Systems: In the same line as before, LoD 1 should provide at least static mean values of the equipment's performance in order to allow calculations of energy flows. Meanwhile, LoD 2 should enable dynamic simulations, including temperatures and mass flows. Finally, LoD 0 could be defined as a data source with information about the existence of equipment.
- Costs: In the case of costs, we will split it into capital costs and operational costs.
  - In the case of capital costs, LoD 1 will provide enough information to evaluate the envelope retrofit costs, per surface area (walls, windows, roof, basement). LoD 2 will help refine the calculation with the detailed layers of construction, and a connection with external costing databases will be established.
  - In the case of operational costs, LoD 1 will provide costs based on yearly and monthly values of fuels and energy vectors, whilst when we move to LoD 2, hourly values and detailed calculations will be required.

# C Exporters

`Exporter factories` export the created city object into different formats. Available factories are `exports factory`, `energy systems factory`, and `energy building exports factory`.

- `exports factory`: Mandatory inputs are handler type, city, and the output path for the exported data. You can select an export format by setting the 'handler' when initializing the class, and then use the 'export' method to perform the export operation based on the chosen format. This factory exports city to the following formats `CesiumJS Tileset`, `GeoJSON`, `GLB`, `OBJ`, `STL`, and `Simplified Radiosity Algorithm (SRA)`.
- `energy systems export factory`: This factory is designed to export energy systems simulation results (currently, the focus is on heat pumps). Mandatory parameters are the city object, handler type, heat pump model, and output path to store simulation results. This factory has the following classes `heat pump export`, `air source hp export`, and `water to water hp export`. Two last classes are inherited from the `heat pump export` class, which runs INSEL, and extracts wanted information.
- `energy building exports factory`: This factory exports a city into several formats related to energy in buildings. Mandatory parameters are the city object, handler type, and output path to store results. `energy building exports factory` has the following classes `idf`, `energy_ade`, and `insel_monthly_energy_balance`.

## C.1 Adding a new exporter factory

A new export class should have a purpose; retrieve the specific module for the given source format. The new export class needs to have the `__init__` method to instantiate the object, at least one handler to export the city to, and an output path to save the exported file.
An example of adding a new class named `NewFactory` is provided below:

```python
# Import required libraries

class NewFactory:
    """
    NewFactory class
    """
    def __init__(self, handler, city, path):
        self._handler = '_' + handler.lower()
        validate_import_export_type(NewFactory, handler)
        self._city = city
        self._path = path

    @property
    def _handler1(self):
        """
        Description of handler1
        Export to _handler1
        """
        # Your implementation here

    @property
    def _handler2(self):
        """
        Description of handler2
        Export to _handler2
        """
        # Your implementation here

    # Add more handler functions as needed


    def export(self):
        """
        Export the city given to the class using the given export type handler
        :return: None
        """
        return getattr(self, self._handler, lambda: None)
```

## D Geometry Factory

The `Geomerty factory` class has eight arguments, two of which are mandatory for all the handlers: path and file type. Other arguments could be used depending on the file type.

- `File type`: Is a mandatory argument and could be any of the available handlers, including `CityGML`, `OBJ` and `GeoJSON`. The handler refers to the file format in this factory for easier understanding. `GeometryFactory` class accepts the three mentioned formats. You can add new file formats to the Hub. A new `@property` should be added to this class, and the corresponding importer class should be created.
- `Path`: Is the second mandatory argument, and it provides the full path for the input file of the Geometry Factory.
- `Aliases field`: Provides a way to add other information that could be interesting such as building name, address or postal code.
- `Height field`: Indicates the field name containing the building height.
- `Year of the construction field`: Indicates the field name containing the building year of construction.
- `Function field`: Indicates the field name for the building function. This function was defined by the one who created the geometry file and can be in any format(strings, numbers...), any language (English, French...), etc.
- `Function to hub`: Hub has a standard list of building functions; if applicable, this parameter changes the function to the Hub's standard. This is done through `dictionaries` that have been explained in 2.2. Dictionaries are defined to convert external information to Hub standards and vice versa.
- `Hub crs`: It indicates the crs (coordinate reference system) to be used by the hub.

The year of the construction field, height field, and function field arguments give the name of the fields inside the input file that save the desired attributes when needed. For example, `CityGML` has standard names defined for function and year of construction, so the importer does not need extra information. It also provides a standard name for the height for LOD[2] 0.5 models, so this field is not needed either. However, `GeoJson` provides space for a list of attributes but without any standard naming, therefore the importer needs to know which names are used in each specific geojson file. Thus, this information must be provided through the arguments.

The `Geomerty factory` has three classes (file type or handler): CityGML, GeoJSON, and OBJ. These handlers read the data from a given path, extract and assign the needed information, and create the city object. Any user-defined handler should follow the same logic.

### CityGML

The CityGML class parses CityGML data, extracts information about city geometry, and creates a structured representation of the city, including its buildings and their properties. This class has the following parameters:

- `path`: The path to the CityGML file location.
- `year_of_construction_field`: This is an optional parameter that indicates the construction year of the building. If not provided, the factory will use the standard field `yearOfConstruction` from CityGML definition.
- `function_field`: This is an optional parameter that identifies the function of the building. If not provided the factory will use the standard field `function` from CityGML definition.

[2]For more information about CityGML LOD visit https://docs.ogc.org/DRAFTS/20-010.html#geometry-lod-section

- `function_to_hub`: The Hub has a basic list of building functions. If applicable, this parameter changes the value in the function to the Hub's known value Eg: 1000 to residential.
- `hub_crs`: It indicates the `CRS` (coordinate reference system) to be used by the Hub. In CityGml it will act as the srs name for a city if the information is not present in the `srsName` field.

The CityGML class reads the input file and creates the building envelope based on the float numbers representing the upper and lower corners of the buildings. The Hub's standard (`hub_crs` = EPSG:26911) will be used if the coordinate system is not provided in the input file and no value was indicated in the `hub_crs` field.

CityGML provides different levels of detail, and the current CityGML factory can only handle up to LOD2.

To add an additional LOD reader for CityGML the new handling class needs to inherit from the gml_base class.

## GeoJSON

The `GeoJSON` provides methods to parse and manipulate GeoJSON data and convert it into a City object. This class has the following parameters:

- `path`: The path of the folder containing the GeoJSON file.
- `aliases_field`: Field list to be used as building aliases from the `Feature properties`.
- `extrusion_height_field`: The field that contains the height of a building inside the `Feature properties`.
- `year_of_construction_field`: The field that contains the construction year of a building inside the `Feature properties`.
- `function_field`: The field that contains the function of a building inside the `Feature properties`.
- `function_to_hub`: The transformation dictionary to covert the function field value to a value known by the Hub. More details on 2.2.
- `hub_crs`: It indicates the crs (coordinate reference system) to be used by the Hub. This parameter will transform from the GPS coordinates to the given format and configure the proper srs name. If the `hub_crs` is not specified, 'epsg:26911' will be used to transform the coordinate system.

## OBJ

The `OBJ` class loads 3D geometry data from OBJ files and converts it into a City object. It initializes by loading the OBJ file specified by the 'path' parameter, extracting information about the 3D scene, including its boundaries, and creating a representation of a city with buildings and associated attributes. The only parameter of this class is the path to the OBJ file.

## Integrate user-defined dictionaries to Hub

A new dictionary could be integrated in the Hub in two ways:

1. Add the dictionary to the user script
2. Integrate the dictionary as a class to the Hub

**1. Add the user-defined dictionary to the script**

A dictionary in the user script could be passed as a parameter to the factory to perform the data transformation. In this example, a new dictionary is created and is being used after importing the geometry factory.

Let's consider the example of Tutorial level 1. The building function in that example is printed as '6000' which we know, according to the 'montreal_function_to_hub_function' dictionary, is a 'medium office'. Therefore, we can create a dictionary for that transformation. To avoid errors when writing strings in the code, the Hub uses constants that save the right spelling, and they are mandatory.

```
import hub.helpers.constants as cte

MY_CUSTOM_DICTIONARY = {
  '6000': cte.MEDIUM_OFFICE
}
```

In the constants library, see all the available constants.

We select MEDIUM_OFFICE as it is the outcome we expect. In the code below, we created a loop on the buildings of the city to overwrite the function using our own dictionary to **MANUALLY** update the building functions.

```
from hub.imports.geometry_factory import GeometryFactory
import hub.helpers.constants as cte

geojson_file = './test_one_building.geojson'
city = GeometryFactory('geojson',
                       geojson_file,
                       height_field='citygml_me',
                       year_of_construction_field='ANNEE_CONS',
                       function_field='CODE_UTILI').city

MY_CUSTOM_DICTIONARY = {'6000': cte.MEDIUM_OFFICE}

for building in city.buildings:
  building.function = MY_CUSTOM_DICTIONARY[building.function]

for building in city.buildings:
  print(f"Hub-format function of building {building.name} is: {building.function}")
```

The same behaviour can be obtained by using the optional parameter function to the Hub and the Hub will perform the update automatically for us, in a more convenient way.

```
from hub.imports.geometry_factory import GeometryFactory
import hub.helpers.constants as cte

geojson_file = './test_one_building.geojson'

MY_CUSTOM_DICTIONARY = {'6000': cte.MEDIUM_OFFICE}

city = GeometryFactory('geojson',
                       geojson_file,
                       height_field='citygml_me',
                       year_of_construction_field='ANNEE_CONS',
                       function_field='CODE_UTILI',
                       function_to_hub=MY_CUSTOM_DICTIONARY
                       ).city

for building in city.buildings:
  print(f"Hub-format function of building {building.name} is: {building.function}")
```

### 2. Integrate the dictionary as a class

If your dictionary is useful for multiple projects or covers a known standard, you could integrate it directly into the Hub and request that it becomes part of future releases. The Dictionary class is a container for various dictionaries. It contains read-only attributes (using `@property` decorator), to access each dictionary contained within the Dictionaries class. To define a new dictionary, please follow the Hub standard and naming convention. An example of a Hub Dictionary is :

```
  @property
  def montreal_function_to_hub_function(self) -> dict:
    """
    Get Montreal function to hub function, transformation dictionary
    """
    return MontrealFunctionToHubFunction().dictionary
```

`MontrealFunctionToHubFunction` class represents a transformation dictionary that maps Montreal-specific function to corresponding hub function. This class has `_dictionary` attribute that holds the actual transformation dictionary, where the keys are Montreal function (as strings, e.g. `‘1000’, ‘2089’`) and the values are corresponding hub function codes from the helper constants (e.g. `cte.RESIDENTIAL`, `cte.INDUSTRY`).
The `MontrealFunctionToHubFunction` class has an empty constructor and a property called dictionary In the example, the constructor initializes the transformation dictionary with the Montreal function codes as keys and the corresponding hub function codes as values.
For the dictionary property, you must include the `@property` decorator to declare it as a property and returns `self._dictionary` which is the previously initialized dictionary.
The same format is valid for other available dictionaries. Therefore, to define a new dictionary you should follow this convention. A new dictionary could be defined as:

```
  @property
  def test_function_to_hub_function(self) -> dict:
    """
    Get test function to hub function, transformation dictionary
    """
    return TestFunctionToHubFunction().dictionary
```

In `TestFunctionToHubFunction` class we will have:

```
import hub.helpers.constants as cte

class TestFunctionToHubFunction:
  """
  Test function to hub function class
  """
    def __init__(self):
    self._dictionary = {'A100': cte.RESIDENTIAL,
                        'B100': cte.INDUSTRY,
                        'C100': cte.MEDIUM_OFFICE,
                        'D100': cte.WAREHOUSE}

    @property
    def dictionary(self) -> dict:
    """
    Get the dictionary
    :return: {}
    """
    return self._dictionary
```

## E Catalogs

The primary purpose of catalogs is to facilitate the coherence and consistency of data among users. By providing a centralized source of detailed technical and commercial information, catalogs help to ensure that all users are working from the same foundation of knowledge, which can improve collaboration, efficiency, and accuracy.

Each catalog factory lists the possible options in a specific topic, using handlers to access them.

All catalogs are inherited from the abstract class Catalog and follows the same logic.

Catalogs are the Hub ETL components for static or semi-static data, and as such, perform a set of steps to Extract, Transform and Load the data making it available for the rest of the Hub components

- **Extract** data from any source, with any format (Cloud, Local files or Databases)
- **Transform** and organize the data so it can be saved in the catalog data structure. This transformations includes, among other things, the conversion of units to the International System of Units and, when possible, homogenizes the information using the same parameters regardless of the original source (e.g. if a source provides reflectance $\rho$ and absorptance $\alpha$, and a second one reflectance $\rho$ and transmittance $\tau$, for the second case it calculates absorptance $\alpha$ from the other two and saves only reflectance $\rho$ and absorptance $\alpha$).
- **Load** the transformed information in the catalog structure. This structure aims to be generic and practical, so it doesn't need to follow 1:1 the `central data model` structure.
- The full catalog is stored using the class `Content` of its data model, and can be retrieved using three different methods:
  - `names(category)`: This method returns a dictionary with the catalog entries' names for each category (if specified).
  - `entries(category)`: This method returns the entire catalog data model, with all values and units, or just those entries of a category if specified.
  - `get_entry(name)`: This method retrieves a specific catalog entry by name.

At the time of writing this document, the available catalog factories are:

- Construction.
- Cost.
- Energy systems
- Usage

A brief description of each catalog is provided in the following sections.

## E.1 Construction catalog factory

The `Construction_catalog_factory` factory provides access to two data sources, i.e. NREL and NRCAN. This factory homogenizes the information from both catalogs into a fixed structure using international units.

This catalog factory is composed of five classes (apart from content class): archetype, construction, material, layers and windows.

- `Archetype`: Represent a construction archetype
  - `archetype_id`: Identification value defined as a string.
  - `function`: The building function that is the subject of this archetype.
  - `climate zone`: The climate zone that is the subject of this archetype.
  - `construction period`: The construction period that is the subject of this archetype.
  - `archetype constructions`: List of constructions that compose this archetype.
  - `indirect heated ratio`: Area of the complete building that is indirectly heated, divided by the total building area.
  - average storey height in meters.
  - `thermal capacity`: The thermal capacity of the complete building represented by this archetype in J/m$^3$K.
  - extra losses due to thermal bridges for the complete building represented by this archetype in W/m$^2$K.
  - `infiltration rate` for ventilation system on and off, both in 1/s.
- `Construction`: Represents a specific combination of materials.
  - `construction_id`: identification of each item by a string.
  - Unique: `name` of an item in string format
  - `type`: types of surfaces including wall, ground, floor, etc.
  - `window ratio`: a dictionary of construction window ratios
  - `window`: information of windows type, i.e. frame ratio, thickness, conductivity, etc.
  - `layers`: a list of construction layers. type of material and thickness in meters (for US construction).
- `Material`: Contains the construction material information.
  - `material_id`: identification of each item by a string.
  - Unique `name` of an item in string format
  - `conductivity` of each material in W/mk
  - `specific heat` of materials in J/kgK
  - Material `density` in kg/m$m^3$
  - `solar absorptance` is the fraction of the sun's radiation that the surface absorbs (between 0 and 1)
  - `thermal absorptance` is the fraction of incident long wavelength infrared radiation that is absorbed by the material (between 0 and 1)
  - `visible absorptance` is the fraction of incident visible wavelength radiation that is absorbed by the material
  - `no mass` could be none or Boolean, showing whether the material's thermal resistance is known.
  - Material `thermal resistance` in m$^2$K/W

- `Layers`: Models each material layer in the construction.
  - `layer_id`: identification of each item by a string.
  - Unique `name` of an item in string format
  - `material` of each layer
  - `thickness` of each layer in meters
- `Windows`: Model windows properties.
  - `windows ID`: identification of each item by a string.
  - Unique `name` of an item in string format
  - `frame ratio` is a float number showing the windows frame ratio.
  - `g-value` is a float number between 0 and 1, addressing each window type's solar heat gain value.
  - `overall_u_value` is the overall heat transfer coefficient represented by a float number in $W/m^2K$ unit.
  - `type` indicates the transparent surface type, either window or skylight.

### NRCAN catalog class

This catalog was created based on the construction archetypes defined by the Canadian Natural Resources agency, available at this address and this address. Some values needed for building demand calculations were unavailable and were therefore assumed to complete the catalog. For the time being, it provides archetypes for all Canadian climate zones, 16 different functions and all years of construction divided into 15 periods, before 1900, every decade until 2010, from 2011 to 2016, from 2017 to 2019 and after 2020.
This catalog reads data regarding the archetype and construction of buildings from two JSON files (`nrcan_archetypes.json` and `nrcan_constructions.json`). Then, loads the information about an archetype, construction, material, and windows and stores it in the catalog data model.

- `nrcan_archetypes.json`: This file stores information in the following categories: function, period of construction, climate zone, average storey height, thermal capacity, extra loss due to thermal bridges, infiltration rate for ventilation system on, infiltration rate for ventilation system off, and details about construction material.
- `nrcan_constructions.json`: Includes the construction information in three categories, i.e. opaque surfaces, transparent surfaces, and materials

### NREL catalog class

This catalog is based on the BCL standards defined by the US National Renewable Energy Laboratory. Click here to find out more about BCL standards. The catalog is not completed. It contains archetypes for 18 different functions but, only for the periods from 2004 to 2008 and after 2009, and only for climate zone 4A.

This class loads the information about windows, material, construction, and archetype and stores it in the catalog data model. Input data for this catalog are stored in two files:

- `us_archetypes.xml`: In this file, each item is recognized by id, building type, reference standard, and climate zone.
- `us_constructions.xml`: This file has information in three groups: windows, materials, and constructions. Each item in these categories is named by id, type, and name. Then, followed by a variety of information, for example, frame ratio, thickness, outside solar absorptance, etc.

To access the catalog information, use the following code:

```python
from hub.catalog_factories.construction_catalog_factory import
    ConstructionCatalogFactory

catalog_handler = 'nrcan' # valid values are 'nrcan' OR 'nrel'

catalog = ConstructionCatalogFactory(catalog_handler).catalog
print(catalog.entries())
```

## E.2 Cost catalog factory

`Cost_catalog_factory` publishes the life cycle cost catalogs.
This tool calculates the Life Cycle Costs for each building in a given region. In the case of the Life Cycle Costing process, we have dealt with them in a separate Classes structure. The different nature of economic hypotheses, much more volatile than energy calculations, has pushed the team to develop a parallel structure that can be coupled with the hub structure. This parallel structure will allow for further analysis, from the perspectives of economic scientists, like being able to implement dynamic cash flow modelling with sensitivity and risk analysis, a topic that has clearly been detected as essential, or monetizing the externalises.

- `Archetype`: Represent a life cycle cost archetype. The properties of this class are:
  - `lod`: a string that identifies the level of detail of the catalog.
  - `function`: The building function that is the subject of this archetype.
  - `municipalicty`: The municipality that is the subject of this archetype.
  - `country`: The country that is the subject of this archetype.
  - `currency`: The currency that is the subject of this archetype.
  - `capital_cost`: The capital cost is returned as a dictionary that has three values; design allowance (%), overhead and profit (%), and chapters.
  - `operational_cost`: The operational cost is returned as a dictionary that includes fuel costs, maintenance costs (heating, cooling and PV), and $CO_2$.
  - `end_of_life_cost`: The end-of-life cost in the given currency per $m^2$
  - `income`: Income is in the form of different subsidies (construction subsidy, HVAC subsidy, etc) and tax reductions that apply to an archetype.
- `Capital cost`: This class processes data, organizes it into chapters, calculates design allowances and overhead/profit percentages and returns the capital cost. The properties of this class are:
  - `general_chapters`: Returns a list of general chapters in capital cost.
  - `design_allowance`: A float number indicating design allowance in percentage for chapters.
  - `overhead_and_profit`: A float number indicating overhead profit in percentage for chapters.
- `Chapter`: This class returns the chapter type and a list of items within a chapter. The properties of this class are:
  - `chapter_type`: In the case of CAPEX, UNIFORMATII has been chosen as the format to classify building elements and related site work. The chapter type depends on this UNIFORMAT II classification.
  - `items`: A list of items contained in the chapter.
  - `item`: Description of a specific item by name.
- `Fuel`: This class returns fixed and variable operational costs of fuel consumption. The properties of this class are:
  - `fuel_type`: An string that identifies fuel type.
  - `fixed_monthly`: Fixed monthly operational cost (dollar or other currencies) per month (currency/month).
  - `fixed_power`: Fixed operational costs depending on the peak power consumed in currency per month per W (currency/month W).
  - `variable`: Variable costs in given units.
  - `variable_units`: Unit of the variable costs.

- `Income`: Income refers to the subsidies in terms of construction, HVAC, photovoltaic, electricity, and tax reduction that a building could benefit from. This class has the following properties:
  - `construction_subsidy`: Subsidy for construction in percentage (%)
  - `hvac_subsidy`: Subsidy for HVAC system in percentage (%)
  - `photovoltaic_subsidy`: Subsidy for PV system in percentage (%)
  - `electricity_export`: Electricity export incomes in currency/J
  - `reductions_tax`: Reduction in taxes in percentage (%)
- `Item description`: Returns initial investment, lifetime and other description of a particular item. This class has the following properties:
  - `item_type`: Returns the type of the item.
  - `initial_investment` and `initial_investment_unit`: The initial investment of the specific item in the given units. The returned value could be a string or float.
  - `refurbishment` and `refurbishment_unit`: The refurbishment costs of the specific item in given units. The returned value could be a string or float.
  - `reposition` and `reposition_unit`: The reposition costs of the specific item in given units. The returned value could be string or float.
  - `lifetime`: Lifetime of an item in years.
- `Operational cost`: Returns the operational costs, including fuel, maintenance, and $CO_2$ emissions. The properties of this class are:
  - `fuels`: A list of fuels in capital costs. Fuel could be electricity, gas, etc.
  - `maintenance_heating`: The cost of maintaining the heating system in currency/W.
  - `maintenance_cooling`: The cost of maintaining the cooling system in currency/W.
  - `maintenance_pv`: The cost of maintaining the PV system in currency/$m^2$.
  - `CO2`: The cost of $CO_2$ emissions in currency/kg $CO_2$

**Montreal custom catalog class**

This factory's only available data source is "Montreal Custom". The data source (XML format) includes cost-related (CA$) information for Montreal's residential and non-residential archetypes. The LOD in this file is 1.
This class extracts and organizes data related to different cost components such as superstructure, envelope, HVAC systems, electrical systems, fuels, and subsidies. Then, stores the information with regard to different archetypes for Montreal. It loads data from an XML file (`montreal_costs.xml`) and organizes it into a structured content model. The catalog includes information on capital costs, operational costs, end-of-life costs, and income-related details for each archetype.

To access the catalog information, use the following code:

```python
from hub.catalog_factories.costs_catalog_factory import CostsCatalogFactory

catalog = CostsCatalogFactory('montreal_custom').catalog
print(catalog.entries())
```

## E.3 Energy systems catalog factory

The `energy systems catalog factory` creates a new `Catalog` object enriched with information about energy systems.
Apart from the class `Content`, that exists in all catalogs, this catalog is composed of five main classes:

- `Archetype`: An archetype is a system cluster. Any combination of systems is allowed to perform a cluster. The only limitation is that more than one system does not cover the same demand. A cluster of systems may represent an archetypal set of systems connected to a building or group of buildings. This class has the following properties:
  - `lod`: The level of details of the catalog.
  - `name`: The name of the energy system archetype.
  - `systems`: A list of equipment that compose the total energy system. See the `system` class below.
- `Distribution system`: Equipment in this class defines water or air distribution systems. Each system's distribution heat loss, distribution consumption fix, and variable flows are specified (%). This class has the following properties:
  - `system_id`: The distribution equipment ID.
  - `name`: The distribution equipment name.
  - `system_type`: The type of the system which could be air, water, and refrigerant.
  - `supply_temperature`: The distribution system's supply temperature in degree Celsius.
  - `distribution_consumption_fix_flow`: The distribution consumption of the pump or fan if they operate at fix mass or volume flow in ratio over peak power (W/W).
  - `distribution_consumption_variable_flow`: The distribution consumption of the pump or fan if they operate at variable mass or volume flow in ratio over energy produced (J/J).
  - `heat_losses`: The heat losses of distribution systems in ratio over energy produced in J/J.
- `Emission system`: Each equipment type has a parasitic consumption in percentage. This class has the following properties:
  - `system_id`: The emission equipment ID.
  - `name`: The emission equipment name.
  - `system_type`: The type of the emission equipment system.
  - `parasitic_energy_consumption`: The parasitic energy consumption per energy produced ratio (J/J).
- `Generation system`: Is a type of generation system, e.g. boiler and furnace, that has specific fuel type, i.e., gas, electricity, renewable. Depending on the generation type, the heating or cooling efficiency and the presence of the storage are defined. This class has the following properties:
  - `system_id`: The generation equipment ID.
  - `name`: The generation equipment name.
  - `system_type`: The type of the generation equipment system.
  - `fuel_type`: The generation system fuel type that could be renewable, gas, diesel, electricity, wood, or coal.
  - `source_types`: The generation system source type which could be air, water, geothermal, district heating, grid, or on-site electricity.
  - `heat_efficiency`: The heat efficiency of a generation equipment.
  - `cooling_efficiency`: The cooling efficiency of generation equipment.
  - `electricity_efficiency`: The electricity efficiency of generation equipment.
  - `source_temperature`: The source temperature of generation systems in degrees Celsius.

  - `source_mass_flow`: The source mass flow of a generation system in kg/s.
  - `storage`: A boolean value showing the existence of a storage system in a generation system.
  - `auxiliary_equipment`: If applicable, a list of auxiliary equipment in a system.

- `System`: A system is a set of equipment that has at least one generation equipment and covers one or more types of demand from Heating, Cooling, Domestic Hot Water and Electricity. This class has the following properties:
  - `lod`: The level of details of the catalog.
  - `system_id`: The ID of the system.
  - `name`: The name of the system.
  - `demand_types`: The covered demand by the system which should be at least one of the following; heating, cooling, domestic hot water, or electricity.
  - `generation_system`: The `generation_system` class that is defined above.
  - `distribution_system`: The `distribution_system` class that is defined above.
  - `emission_system`: The `emission_system` class that is defined above.

**Montreal custom catalog class**

For the time being, this factory only provides access to one data source, named "Montreal custom". This data source is the result of a study made within the group of the energy systems installed in buildings in Montreal. The original data to perform the study comes from these sources:

- Commercial Buildings Energy Consumption Survey CBECS
- Survey of Household Energy Use SHEU
- Dataportal for cities (Montreal)
- NECB: HVAC systems typologies use the equivalence proposed in the National Energy Code for Buildings for Canada between typologies of buildings and typologies of HVAC systems. The table used is table 8.4.4.7-A from the NECB 2020.

The information provided by this catalog has a maximum level of detail of 1. It is LoD = 0 in the case of the storage systems. This catalog is designed based on several system archetypes. Each archetype is made up of a combination of the systems; for example, a heat pump, boiler, and PV could be an archetype.

To access the catalog information, use the following code:

```
from hub.catalog_factories.energy_systems_catalog_factory import
    EnergySystemsCatalogFactory

catalog = EnergySystemsCatalogFactory('montreal_custom').catalog
print(catalog.entries())
```

## E.4 Usage catalog factory

`Usage catalog factory` publishes usage information, i.e. those parameters related to the interaction of the people with the building. This factory provides access to three data sources, i.e. COMNET, NRCAN and EILAT. It homogenizes the information from mentioned catalogs into a fixed structure using international units. The usage data includes information on how the building is utilized, such as occupancy patterns and energy demand profiles. The seven main classes of this factory are appliances, domestic hot water, lighting, occupancy, schedule, thermal control, and usage (the archetype definition).

- `Appliences`: Returns the schedule and other information regarding the power density of appliances. This class has the following properties:
  - `density`: The power density of appliances in W/$m^2$.
  - `convective_fraction`: Convective fraction of the power density.
  - `radiative_fraction`: Radiant fraction of the power density.
  - `latent_fraction`: Latent fraction of the power density.
  - `schedules`: A list of objects' `Schedule` (see the `schedule` class below). In this case, they save a fraction of the power density distributed in time (depending on the data source, it could vary based on the day type).
- `Domestic hot water`: Returns usage information, including schedule, density, etc., of domestic hot water. The properties of this class are:
  - `density`: The domestic hot water load density in W/$m^2$.
  - `peak_flow`: The domestic hot water peak flow density in m3 per second and $m^2$ ($m^3$ / $sm^2$).
  - `service_temperature`: The service temperature in degrees Celsius.
  - `schedules`: A list of objects' `Schedule` (see the `schedule` class below). In this case, they save a fraction of the load density distributed in time (depending on the data source, it could vary based on the day type).
- `Lighting`: Return the schedule and density related to lighting usage. This class has the following properties:
  - `density`: The lighting density in Watts per m2 (W/m$m^2$).
  - `convective_fraction`: Convective fraction of the lighting density.
  - `radiative_fraction`: Radiant fraction of the lighting density.
  - `latent_fraction`: Latent fraction of the lighting density.
  - `schedules`: A list of objects' `Schedule` (see the `schedule` class below). In this case, they save a fraction of the lighting density distributed in time (depending on the data source, it could vary based on the day type).
- `Occupancy`: Returns the occupant's schedule, density and other details. This class has the following properties:
  - `occupancy_density`: The density in persons per $m^2$
  - `sensible_convective_internal_gain`: The sensible convective internal gain in W/$m^2$.
  - `sensible_radiative_internal_gain`: The sensible radiant internal gain in W/$m^2$.
  - `latent_internal_gain`: The latent internal gain in W/$m^2$.
  - `schedules`: A list of objects' `Schedule` (see the `schedule` class below). In this case, they save a fraction of the occupancy density distributed in time (depending on the data source, it could vary based on the day type).

- `Schedule`: Returns a schedule with a specific time step, day type, etc., for available schedule types (electricity, lighting, domestic hot water, occupancy, cooling, and heating). This class has the the following properties:
  - `schedule_type`: The type of the schedule. For example, lighting, occupancy, etc.
  - `values`: A list of schedule values.
  - `data_type`: The schedule data type could be one of the following: any number, fraction, on / off, temperature, humidity, and control type.
  - `time_step`: The scheduled time step that could be second, minute, hour, day, week and month.
  - `time_range`: The scheduled time range that could be minute, hour, day, week, month, and year.
  - `day_types`: A schedule could have a schedule for different day types, including Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday, holiday, winter design day, and summer design day.
- `Thermal control`: Returns the usage information of the thermal control, including heating, cooling and HVAC system. This class has the following properties:
  - `mean_heating_set_point`: The heating set point is defined for a thermal zone in Celsius. It is the mean value of the heating set points of the full period defined (normally one year, but not exclusively.)
  - `heating_set_back`: It represents a lower heating set point when the system is on, but the need for comfort conditions is not so high. The heating set back is defined for a thermal zone in Celsius.
  - `mean_cooling_set_point`: The cooling set point defined for a thermal zone in Celsius. It is the mean value of the cooling set points of the full period defined (normally one year, but not exclusively.
  - `hvac_availability_schedules`: A list of objects' `Schedule` (see the `schedule` class above). In this case, they save the availability of the conditioning system defined for a thermal zone (on / off). Depending on the data source, it could vary based on the day type.
  - `heating_set_point_schedules`: A list of heating set point schedules defined for a thermal zone in Celsius.
  - `cooling_set_point_schedules`: A list of cooling set point schedules defined for a thermal zone in Celsius.
- `Usage`: Return the details of usage, including hours per day, occupancy, thermal control, etc., for a given usage zone. This class has ten properties:
  - `name`: Name of each usage zone.
  - `hours_day`: A float number indicating usage hour per day for each usage zone.
  - `days_year`: A float number indicating usage days per year for each usage zone.
  - `mechanical_air_change`: The mechanical air change in air change per second (1/s) when system on for each usage zone.
  - `ventilation_rate`: The ventilation rate in $m^3/m^3$s for each usage zone.
  - `occupancy`: The object `Occupancy` (density, sensible convective internal gain, etc.) associated to this usage.
  - `lighting`: The object `Lighting` (density, convective fraction, etc.) associated to this usage.
  - `appliances`: The object `Appliances` (density, convective fraction, etc.) associated to this usage.
  - `thermal_control`: The object `ThermalControl` (heating and cooling set points' temperatures, schedules) associated to this usage.
  - `domestic_hot_water`: The object `DomesticHotWater` (peak flow, service temperature, etc.) associated to this usage.

### COMNET catalog class

This catalog contains comprehensive information about various building usage parameters, including occupancy, lighting, appliances, thermal control, and domestic hot water in the US. The class parses and processes data from two Excel files, one containing the general information and the second one the schedules. It differentiates between 33 usages that include residential and non-residential types. The two Excel files are:

- `comnet_archetypes`: for each *usage*, or archetype category, it provides specifications and standards (ASHRAE standard) for lighting, plug loads, occupancy, hot water, etc. And the associated schedules.
- `comnet_schedules_archetypes`: Provides the schedules information associated to the usages. For each usage, there can be found several schedule types (occupancy, lighting, HVAC, hot water, etc.) and 3 schedules for each of them (weekday, Saturday, Sunday) on an hourly basis.

### NRCAN catalog class

NRCAN provides different building usages and energy characteristics in Canada. It is data from the National Energy Code of Canada for Buildings (NECB) published by the National Research Council of Canada (NRC). The codes can be found here. This catalog parses and processes that data using an XML file with the reference links. The `base url` in the XML file provides address to three files:

- `‘‘space types’’` specifies the relative path to a JSON file named `space_types.json` which contains information about space types and building archetypes based on NECB (National Energy Code of Canada for Buildings) year 2015 data. Archetypes are categorized based on whole building type.
- `‘‘space types compliance’’` specifies the relative path to another JSON file named `space_compliance_2015.json`. This file contains compliance data, for example, usage type, occupancy density, lighting density, and domestic hot water peak flow, for different building types based on the NECB standard.
- `‘‘schedules’’` specifies the relative path to a JSON file named `schedules.json` which contains information about various schedules, i.e. occupancy, domestic hot water, lighting, heating, and cooling schedule, used in different building types.

To access the catalog information, use the following code:

```
from hub.catalog_factories.usage_catalog_factory import UsageCatalogFactory

catalog = UsageCatalogFactory('comnet' OR 'nrcan' OR 'eilat').catalog
print(catalog.entries())
```

## E.5 Creating a new catalog

If a new catalog of an existing topic is required, the following steps must be taken:

1. The test where that factory is tested should be run to ensure that the starting point works well.
2. The existing data model must be reviewed to confirm that it supports the new catalog. If not, this must be extended or modified to accommodate the new data.

   **NOTES:**

   The data model should be modified as little as possible.

   If a new characteristic is already represented with a different parameter, for example circle area and circle diameter, only one of those parameters should be selected to save the information. The catalogs should do the needed calculations to translate from the original source to the data model parameter.

   All other catalogs in the same topic must be reviewed to ensure that the modifications didn't affect them or to make the needed updates.
3. The test where that factory is tested should be run again to check that nothing was broken. If so, FIX IT and re-run it.
4. The `Catalog` class (or classes) named `NewNameCatalog` has to be written and saved in the folder of that topic. One `Catalog` per data source is required, creating as many as needed per topic. The files with the data or the link to access the data should be added to the folder ./data/topic/
5. The new `Catalog` should be added to the corresponding `CatalogFactory`, creating a new property that is accessed with a new handler.
6. Finally, a new test should be added to test the new code.

An example of a new catalog is presented below.

```
"""
My New Data Source energy systems catalog module
SPDX - License - Identifier: LGPL - 3.0 - or -later
Copyright 2023 Concordia CERC group
Project Coder Name email@concordia.ca
"""

from hub.catalog_factories.catalog import Catalog
from hub.catalog_factories.data_models.energy_systems.system import System
from hub.catalog_factories.data_models.energy_systems.content import Content
from hub.catalog_factories.data_models.energy_systems.generation_system import
    GenerationSystem
from hub.catalog_factories.data_models.energy_systems.distribution_system import
    DistributionSystem
from hub.catalog_factories.data_models.energy_systems.emission_system import
    EmissionSystem
from hub.catalog_factories.data_models.energy_systems.archetype import Archetype
from hub.catalog_factories.catalog import Catalog
# Add new imports if needed


class MyNewDataSourceCatalog(Catalog):
    """
    My New Data Source Energy Systems Catalog class
    Implement methods to retrieve energy system data specific to the custom catalog.
    """
    def __init__(self, base_path):
        super().__init__(base_path)  # Initialize the base class

    # Implement methods to retrieve energy system data for the custom catalog
    # Three following methods must be included:

    def names(self, category=None):
        """
        Method to return the catalog entries' names.
        :return: Catalog names filter by category if provided
        """
        return _names

    def entries(self, category=None):
        """
        Method to return the full catalog entries or those of a specified category
        :return: Catalog content filter by category if provided
        """
        return entries

    def get_entry(self, name):
        """
        Method to return a catalog entry matching the given name
        :return: Catalog entry with the matching name
        """
        return entry
```

An example of a modified catalog factory adding the new catalog is presented here.

```
"""
Energy Systems catalog factory, publish the energy systems information
SPDX - License - Identifier: LGPL - 3.0 - or -later
Copyright 2023 Concordia CERC group
Project Coder Pilar Monsalvete Alvarez de Uribarri pilar.monsalvete@concordia.ca
"""

from pathlib import Path
from typing import TypeVar

from hub.catalog_factories.energy_systems.montreal_custom_catalog import
    MontrealCustomCatalog
# This is new:
from hub.catalog_factories.energy_systems.my_new_data_source_catalog import
    MyNEWDataSourceCatalog


from hub.helpers.utils import validate_import_export_type

Catalog = TypeVar('Catalog')


class EnergySystemsCatalogFactory:
  """
  Energy system catalog factory class
  """
  def __init__(self, handler, base_path=None):
    if base_path is None:
      base_path = Path(Path(__file__).parent.parent / 'data/energy_systems')
    self._handler = '_' + handler.lower()
    validate_import_export_type(EnergySystemsCatalogFactory, handler)
    self._path = base_path

  @property
  def _montreal_custom(self):
    """
    Retrieve Montreal Custom catalog
    """
    return MontrealCustomCatalog(self._path)

# This is new
  @property
  def _my_new_data_source(self):
    """
    Retrieve My New Data Source catalog
    """
    return MyNewDataSourceCatalog(self._path)

  @property
  def catalog(self) -> Catalog:
    """
    Returns a new catalog using given handler
    :return: Catalog
    """
    return getattr(self, self._handler, lambda: None)
```

### E.5.1 Creating a new catalog factory

If a complete new topic should be introduced (e.g. an Embodied Carbon Catalog), several steps must be taken:

1. The corresponding data model must be created and saved in a new folder with the name of the catalog factory hanging from the `data_models` folder.
2. The `Catalog` class (or classes) named `NewNameCatalog` has to be written in the same way it was explained in section E.5 and saved in a new folder specifically created for this new topic. One `Catalog` per data source is required, creating as many as needed per topic.
3. A new `CatalogFactory` class called `NewNameCatalogFactory` must be added to the `catalog_factories` folder.
4. Finally, a new test should be added to test the new factory.

An example of a new catalog factory is shown below. In this case, the python file should be called `embodied_carbon_catalog_factory.py`.

```
"""
Embodied Carbon catalog factory, what it does...
SPDX - License - Identifier: LGPL - 3.0 - or -later
Copyright 2023 Concordia CERC group
Project Coder Name email@concordia.ca
"""

from pathlib import Path
from typing import TypeVar

from hub.catalog_factories.embodied_carbon.new_embodied_carbon_catalog import
    NewEmbodiedCarbonCatalog
# (as many of these as catalogs to add to the factory)

from hub.helpers.utils import validate_import_export_type

Catalog = TypeVar('Catalog')


class EmbodiedCarbonCatalogFactory:
  """
  Embodied Carbon Catalog Factory class
  """
  def __init__(self, handler, base_path=None):
    if base_path is None:
      base_path = Path(Path(__file__).parent.parent / 'data/embodied_carbon')
    self._handler = '_' + handler.lower()
    validate_import_export_type(EmbodiedCarbonCatalogFactory, handler)
    self._path = base_path

  @property
  def _new_embodied_carbon(self):
    """
    Retrieve new embodied carbon catalog
    """
    return NewEmbodiedCarbonCatalog(self._path)

  @property
  def catalog(self) -> Catalog:
    """
    Returns a new catalog using the class given handler
    :return: Catalog
    """
    return getattr(self, self._handler, lambda: None)
```

TODO: Some additional examples need to be provided in the future