# BUILDING QUERY LANGUAGE

## DOCUMENTATION

Author: Ing. Martin Cerny, 25/09/13

# Content

# 1 About this document

This document is intended to be a documentation of Building Query Language (BQL) and its reference implementation for xBIM Toolkit. BQL is implemented in Xbim.Script.dll shared library and it is part of the xBIM Toolkit which use this as a scripting engine. Definition of the language is open and may change during the time. So this documentation is valid only for the actual state of the language. However, main parts and syntax are not likely to change.

Language is fully defined by the syntax defined in appendix A and lexicon defined in appendix B. BQL syntax is defined in YACC-like language defined in Gardens Point Parser Generator (GPPG) documentation [1]. Lexicon is defined in LEX-like language defined by Gardens Point LEX (GPLEX) documentation [2]. These two tools were used to generate reference implementation of the language and describe full range of language expressions. Meaning and functionality of the BQL expressions will be described in the following chapters. Language examples will be given which will explain the syntax and expected behaviour.

# 2 General concepts of the language

Language is declarative language like SQL (Structured Query Language). While SQL is designed to query relational data in relational databases BQL is designed on top of IFC object data model. IFC 2x3 version is used for the actual implementation. IFC 4 is more actual release but it is not supported industry-wide at the moment so IFC 2x3 is better choice for now. It is hence useful to know something about the IFC data structure [3] but it is not required. Most of the language constructs are intuitive as names of IFC objects follow industry naming conventions. List of the building objects which may be of the greater interest for most of the users are listed in appendix C. However BQL is focused mainly on this type of objects it can be used to query any object in IFC model. Definition of all products in IFC meaning is following [3]:

> *Subtypes of IfcProduct usually hold a shape representation and a local placement within the project structure. This includes manufactured, supplied or created objects (referred to as elements) for incorporation into an AEC/FM project. This also includes objects that are created indirectly by other products, as spaces are defined by bounding elements. Products are defined by their properties and representations.*

If you try to dig into IFC data structure you may realize that it is very complex with all the objectified relations, abstract types, interfaces and other features from object oriented modelling and STEP specification. To make it easier we have made some of these features flat and transparent to the user even if they still exist in the background. It applies to the data querying as well as to data creation when complex data structures are created in the background.

Saying so it is useful to know something about the hierarchy of the products because you can great take advantage from that. Hierarchy is shown in appendix D which you can use as a reference. If you want, for example, to select all sites, buildings, spaces and storeys it is enough to select all spatial structure elements (see the structure in appendix D on page 23). This feature of the object model is called polymorphism.

It was a design intend to create language which is close to natural language so that it is “human readable”. It means that people with no programming knowledge can use the language and they can learn it easily from examples which are self-explaining and readable. For the sake of simplicity and to avoid typos in the language BQL is designed to be case insensitive as much as possible except for the names of variables. There is a certain upper case and lower case convention used in examples in the following chapters but it is only to make examples more clear.

All of the typical expressions for querying can be expressed in a verbose form “is greater than” or as a mathematical operator “>”. First makes it easier to read for people who don't like maths. The

later one is shorter and may make it easier to read longer scripts and commands.

Close world evaluation is another important assumption built in the BQL interpret. In normal world it is normal to answer “I don't know”. But it would be very complicated to work with this in computer environment. So we took approach similar to most of the query languages where if you ask a question which can lead to “YES” or “NO” answer you will always get “NO” even if the fact is unknown. You should keep this on mind when you structure any query to get the data.

Base language for BQL expressions is English language as this is widely used language worldwide and most of the technically educated people are able to cope with this language at least on the level of technical expressions. However, it is easy to port BQL to another language by translating the lexicon of the language which defines keywords and culture specifics for decimal numbers (typically using comma instead of decimal point). Translation of the IFC specific keywords can be done manually (the hard way for more than 653 objects) or it can take advantage from IDF project of buildingSMART [4] (the smart way of doing so).

To summarize base design concepts of the Building Query Language:

- Data querying, creation and manipulation
- IFC data structure compatible
- Simplified access to the IFC data model
- Close to natural language (intuitive)
- Case insensitive
- Close world evaluation
- Easy translation to other languages (Czech, French, German etc.)

# 3 Language description

BQL language constructs will be described in the following chapters. Text is structured according to different type of language expressions. These are queries which can be user to interrogate and inspect the model, data creation expressions which can be used to create new objects and data manipulation expressions which can be used to modify existing or newly created objects.

## *3.1 Using variables*

There is a concept of variables defined in BQL. They are important for data querying and manipulation. Notation of queries is any sequence of alphabetical characters preceded with '$' sign. Examples of variable identifier could be: *$walls, $newSlab, $foo*. There is also special variable '*$$*'. This variable is used by BQL to assign results of queries or data creation expressions which are not assigned to a specific variable. You can use this as a short-cut or just to explore the latest result. Just be careful that this variable changes its content every time you use expressions like “Select ...” or “Create new ...”. Variables are called *IDENTIFIER* in BQL syntax definition so you can explore it more into detail in appendix A. Their specific use will be explained in the following chapters. There id no need to declare variables. They appear when used for the first time.

You can add elements to the variables with selection or creation statements. Selection will always add or remove objects from the selection set but creation will always empty the set and add only the object just created. The basic operations are shown in the examples bellow and more details will be given in the following chapters.

```
$WallsAndSlabs IS EVERY wall;
$WallsAndSlabs IS EVERY slab;
$WallsAndSlabs IS NOT EVERY slab WHERE PREDEFINED_TYPE IS ROOF;
```

Result of this expression is a set of objects called *$WallsAndSlabs* which contains all walls and all slabs except for the slabs with predefined type *ROOF*.

```
$NewWall IS NEW wall WITH NAME 'Brand new shiny wall' AND DESCRIPTION
'This is the best wall ever and it is the only member of the set.';
```

This expression will empty content of the *$NewWall* if there is any and it will add only the newly created wall object to it. If you use the language from the interactive interpret you can use the *COUNT* command to see the number of objects in the set.

```
COUNT $WallsAndSlabs;
```

If you want to see the list of attributes or properties of the objects in the set use the *DUMP* command. You can also use this command to export the values to the CSV file. This might be useful for simple data extraction. This functionality might be extended in the near future so that it is possible to export data to the spreadsheet as well.

```
DUMP $WallsAndSlabs;
DUMP name, description, 'fire rating', 'thermal performance' from
$WallsAndSlabs;
DUMP name, description, 'fire rating', 'thermal performance' from
$WallsAndSlabs TO FILE 'results.txt' ;
```

First expression will print the results to the output and it will show its label (number introduced by '#' character), which can be used to identify the object in the IFC file, type and name if it is defined for the object. Second expression will print to output values of attributes or properties with the names defined separated by comma (hence the result is 'Comma Separated Values'). The last expression will save the CSV to the file specified. If the attribute and property with the same name exist attribute will always take precedence. Difference between attributes and properties will be explained in the chapters 3.3.1 and 3.3.2.

Finally if you want to clear content of the variable use the *CLEAR* command which will empty the set of objects.

```
CLEAR $WallsAndSlabs;
```

## 3.2 Operators

It was described above that BQL is designed to be human readable and easy to understand. To make statements more similar to the natural language all operators have their short and verbose alternative. BQL operators are listed in the following table where short form is the first one and verbose variants follow.

| Short form | Verbose form | | | |
|---|---|---|---|---|
| = | equals | is | is equal to | |
| != | is not equal to | is not | does not equal | doesn't equal |
| > | is greater than | | | |
| < | is less than | | | |
| >= | is greater than or equal to | | | |
| <= | is less than or equal to | | | |
| && | and | | | |
| \|\| | or | | | |
| ~ | contains | is like | | |
| !~ | does not contain | doesn't contain | isn't like | is not like |

## 3.3 Data selection

Queries are first class citizens of BQL language and it was the reason to create new language. It was already mentioned above that IFC data structure is rather complex but BQL makes it easier to access and extract data from the model of the building. Some of the concepts of IFC which are useful for modelling purpose are flatten down to make it easier to understand the model. One of such a concepts is existence of objectified relations in IFC schema. These are used very often to assign material to product, associate product with certain group of classification and many other purposes but make the model more complex at the same time.

Result of the query is always set of IFC objects. This set can be empty and some of the actions require only one object in a set but it is always set. The most general syntax is described as:

```
selection
      : SELECT EVERY selection_statement
      | IDENTIFIER op_bool selection_statement
      ;
```

First expression is introduced by "*SELECT*" keyword which is followed by the selection statement. These statements are described in the chapters bellow. This will perform select operation and will return set of selected objects to the special variable '*$$*'. It you use command line interpret of the language results are listed by type of the object, its name and its label (number introduced by hash). You can use label to find the object easily in the IFC file. Example of this king of query would be:

```
SELECT EVERY wall WHERE name IS LIKE 'Cladding wall';
```

Second expression is introduced by *IDENTIFIER* which represents the name of the variable as defined in chapter 3.1. Result of the query is assigned to or excluded from the variable according to the operator (see chapter 3.2 for reference). Example of this king of query would be:

```
$floors IS EVERY floor WHERE PREDEFINED TYPE IS floor;
$externalWalls IS NOT EVERY wall WHERE TYPE name IS NOT LIKE 'internal';
```

Selection statements can be of many kinds and will be described in the following text. The type of the result must always be specified at the beginning of the selection statement. There is wide range of objects specified in IFC schema probably but the most common subjects of querying are products which are listed in appendix C. You can use three simple ways to specify the object type – IFC notation, IFC notation without "Ifc" prefix and IFC notation where words are separated by underscore. All the names are case insensitive but it is convenient to use so called "Camel Case" to keep the names readable. Example of this notation is listed in the table bellow:

| | | |
|---|---|---|
| IfcWall | Wall | wall |
| IfcWallType | WallType | wall_type |
| IfcStairCase | StairCase | stair_case |
| IfcDoorStyle | DoorStyle | door_style |
| IfcSlab | Slab | slab |
| IfcBuildingStorey | BuildingStorey | building_storey |
| IfcDistributionPort | DistributionPort | distribution_port |
| ... | ... | ... |

Every selection statement is focused on one type of the object. If you want to create more general query you can use the hierarchical structure shown in appendix D. Or you can create multiple queries using one variable which will accumulate all the results. You can also use the "*is not*" statement to exclude objects from the selection. This is important because the resulting set of results has different types. See the example bellow:

```
$WallsAndSlabs IS EVERY wall;
$WallsAndSlabs IS EVERY slab;
$WallsAndSlabs IS NOT EVERY slab WHERE PREDEFINED_TYPE IS ROOF;
```

Result of this example will be set of all walls and slabs excluding slabs with predefined type set to *ROOF*.

If there is a conflict between the name of property and any of the keywords just enclose it between simple of double quotes. These have the same meaning marking the start and the end of the text. If it is a single word text which is not a keyword (keep in mind that BQL is case insensitive) it doesn't have to be enclosed in quotes but we recommend to do so anyway as set of keywords may be extended in the future and so you will avoid the future conflicts. Moreover, text without quotes is always evaluated to be the name of the object type shown above (wall, slab, etc. ).

### 3.3.1 Attribute queries

We will distinguish between attributes and properties for the purpose of this text. By attributes we mean data fields of the objects which are defined for the specific type of the object in the IFC schema. Example of the attribute would be the name or description of the wall. If you have a look into the IFC documentation, attribute would be every field of the object with the simple value (not another referenced object). For the actual version of the BQL specification only three attributes are supported which are the most specific for all the products defined in IFC. These are *NAME*, *DESCRIPTION* and *PREDEFINED TYPE*. Name is more than just a name. In many systems there is a requirement for the name to be unique identifier of the object. However this is not the case of IFC specification it is true for BIM authoring tools like Autodesk Revit. Predefined type is used to distinguish between specific types of more general object. Typical use is to distinguish between slab used to model floor, slab used to model basement and slab used to model parts of the roof.

Attribute names are case insensitive like most of the BQL expressions. Predefined type follow the approach used for the names of object types made by more than one word. It can be defined as "*predefined type*", "*predefinedtype*" or "*predefined_type*" with any variation in case of the letters. Examples of the attribute queries:

```
SELECT EVERY slab WHERE PREDEFINED_TYPE IS basement_slab;
SELECT EVERY wall_type WHERE NAME IS 'The best wall type ever';
SELECT EVERY column WHERE DESCRIPTION CONTAINS 'eternal support';
```

In the xBIM implementation of BQL if attribute with the specified name is not found for the object it tries to find property with the same name. It might be a case that some types of the objects don't have an attribute *NAME* defined but they can have a property representing the same concept. Properties are described in the chapter below.

### 3.3.2 Property queries

For the purpose of this text we understand properties as a simple values assigned to the objects which are not defined as the attributes of the objects. Some of the property sets are predefined in IFC schema (like *Pset_WallCommon* which should be used to store common properties of the wall like acoustic rating, fire rating, thermal transmittance and others), but any other property set and property can be defined. You can imagine properties like a free fields you can add to specify and describe the object.

Properties can be organized into property sets in IFC model and all of this is assigned to the objects by objectified relations. BQL is hiding all of this complexity and it uses flat structure which assigns properties straight to the objects for the querying purpose. This approach is much simpler and it makes it easier to interrogate the model to find the objects you want. Type of the value to compare is checked against the type of the property (it can be either text, integer, decimal number, boolean

value or NULL special value). If the type of the property is different BQL interpret tries to convert it. If the value is still not compatible interpret will give an error (it wouldn't crash but would warn you that the type is incompatible).

It is mentioned in the chapter 3.3.1 that properties are fall back option for the attribute queries which don't find any relevant attribute defined for the object. If you want to search just properties which could be attributes as well define the name in quotes. Following statements will search only properties (compare to the example in chapter 3.3.1):

```
SELECT EVERY slab WHERE "predefined_type" IS "basement slab";
SELECT EVERY wall_type WHERE "name" IS 'The best wall type ever';
SELECT EVERY column WHERE "description" CONTAINS 'eternal support';
```

### 3.3.3 Material queries

Material is important part of the information about the product in the building model. There is complex infrastructure of distinct data objects in IFC schema used to define and assign the material or material layers to the products of product types. All of this complexity is hidden in BQL as it tries to access every available material definition related to the product. Syntax for the material conditions is defined as:

```
materialCondition
      : MATERIAL op_bool STRING
      | MATERIAL op_cont STRING

      | THICKNESS op_bool num_value
      | THICKNESS op_num_rel num_value
      ;
```

First syntax is usable only for the objects which have just one material defined and the name specified must match the name of the material (case insensitive). Second syntax is more relaxed and it will search all the materials including materials in the material layers and will return all the objects where any of the **material names contain the text specified as STRING**.

The last two conditions provide the ability to as for the thickness of the material. This is not geometrical condition but material layer condition. So it makes a sense for layered products like walls, slabs etc. If the product doesn't have a material layer set defined this condition returns false.

Lets have a wall with material layers defined as follows:

| Wall name | Material | Thickness |
|---|---|---|
| External Block Cavity Insulation Partial Fill Block | Aggregate Concrete Blocks | 100,00 |
| | Cavity | 50,00 |
| | Polyisocyanurate Foam Boards | 50,00 |
| | Aggregate Concrete Blocks | 100,00 |

First of the following examples will return empty set because the wall doesn't have just the 'Cavity' material. But the next two conditions will evaluate to true because one of the materials defined in the wall's layer structure contains the name specified.

```
SELECT EVERY wall WHERE MATERIAL IS 'Cavity';
SELECT EVERY wall WHERE MATERIAL CONTAINS 'Cavity';
SELECT EVERY wall WHERE MATERIAL CONTAINS 'cav';
```

### 3.3.4 Type queries

Type of the element can be little confusing in the IFC schema. There is a concept of objectified

element type which can hold attributes and properties which are the same for all the instances of the element itself. This is different from the programmatic point of view where element type has its programmatic type and element has its programmatic type as well. But the concept is truly simple ans sensible. For example, there might be several instances of wall which are all related to one instance of wall type which defines properties and material layer set for all the walls. This makes it easier to change any property for all the occurrences at once and it avoids redundancies in the data. Definition form International Alliance for Interoperability defines:

> *The product type defines a list of property set definitions of a product and an optional set of product representations. It is used to define a product specification (i.e. the specific product information, that is common to all occurrences of that product type).*

Using BQL you can interrogate the objects in the model based on their type's attributes and properties if the objects have the type defined or you can just check if the type is defined. Syntax is defined as follows:

```
typeCondition
      : TYPE op_bool PRODUCT_TYPE
      | TYPE op_bool STRING
      | TYPE op_cont STRING
      | TYPE op_bool NONDEF
      | TYPE OP_NEQ DEFINED
      | TYPE OP_EQ DEFINED
      | TYPE propertyCondition
      | TYPE attributeCondition
      ;
```

First expression evaluates against the product type which can be any of the types like *IfcWallType, IfcSlabType, IfcDoorStyle, IfcWindowStyle* and others. You can use the same notation as for the products. Examples are listed in the following table:

| | | |
|---|---|---|
| *IfcWallType* | *WallType* | *wall_type* |
| *IfcDoorStyle* | *DoorStyle* | *door_style* |
| *IfcSlabType* | *SlabType* | *slab_type* |
| *IfcDoorStyle* | *DoorStyle* | *door_style* |
| ... | ... | ... |

Next two conditions are short-cuts to evaluate name of the type.

```
SELECT EVERY wall WHERE TYPE IS 'Wall type No.1';
SELECT EVERY wall WHERE TYPE CONTAINS 'No.1';
```

Next three conditions check if the element type is associated with the element.

```
SELECT EVERY wall WHERE TYPE IS NOT null;
SELECT EVERY wall WHERE TYPE IS DEFINED;
SELECT EVERY wall WHERE TYPE IS NOT DEFINED;
```

Last two conditions act just like the attribute and property conditions for the selection statement itself but it is nested for the type.

```
SELECT EVERY wall WHERE TYPE name IS 'Wall type No.1';
SELECT EVERY wall WHERE TYPE description IS DEFINED;
SELECT EVERY wall WHERE TYPE 'Manufacturer' IS 'Bob the builder';
```

As a matter of convenience for Revit users there is possible syntax variant where you can use 'FAMILY' instead of 'TYPE' as the concept of families in Revit is pretty much the same as the

concept of element types in IFC schema.

```
SELECT EVERY wall WHERE FAMILY name IS 'Wall type No.1';
SELECT EVERY wall WHERE FAMILY description IS DEFINED;
SELECT EVERY wall WHERE FAMILY 'Manufacturer' IS 'Bob the builder';
```

### 3.3.5 Group queries

Groups are not used very often in the actual IFC models but it is very useful concept for grouping of objects into the systems or classifications. Groups can form hierarchical structures to represent classification systems for example. Note that there is infrastructure in IFC schema dedicated to definition of classification itself but from our point of view it is often more convenient to use groups and systems to represent this structure.

BQL acts with groups in similar way as with types. So the syntax is defined as:

```
groupCondition
      : GROUP propertyCondition
      | GROUP attributeCondition
      ;
```

Examples of the statements should be self-descriptive:

```
SELECT EVERY wall WHERE GROUP name IS '02.01.01';
SELECT EVERY wall WHERE GROUP description IS 'Steel Frames';
SELECT EVERY wall WHERE GROUP 'Fancy Property' IS 'The fancier property
ever';
```

### 3.3.6 Model queries

There is a concept of referenced models and federation model in BQL and xBIM. This is explained more into detail in chapter 3.6. If you open or create federation model with one or more referenced models you can also use the model name, owner and organization as a part of the queries. This condition is applicable only on the elements within the referenced models, not the main model.

```
SELECT EVERY wall WHERE MODEL IS 'C:\MyModel.ifc';
SELECT EVERY wall WHERE MODEL IS 'MyModel';
SELECT EVERY wall WHERE MODEL OWNER IS 'Bob';
SELECT EVERY wall WHERE MODEL ORGANIZATION IS 'Bob the Builder ltd.';
```

As you may have noticed there is a variation in the model name condition. You can either use the path as it is defined first time when creating the federation model or you can use just the name of the file as long as it is unique within the federation model.

### 3.3.7 Chaining the selection statements

In most of the examples listed above there was just one condition used to select the objects from the model. But it could be often the case that more than just one condition need to be applied. In the actual definition of BQL you can chain conditions with *AND* or *OR* operators. If you use *OR* operator it is enough if any of the conditions is evaluated to be true. If you use *AND* operator all of the conditions must be true. You can use normal braces to create logical groups of conditions as shown bellow.

```
CREATE NEW wall WITH NAME 'My wall No. 1' AND DESCRIPTION 'First
description contains dog.';
CREATE NEW wall WITH NAME 'My wall No. 2' AND DESCRIPTION 'Second
description contains cat.';
```

```
CREATE NEW wall WITH NAME 'My wall No. 3' AND DESCRIPTION 'Third
description contains dog and cat.';
CREATE NEW wall WITH NAME 'My wall No. 4' AND DESCRIPTION 'Fourth
description contains dog and cow.';
$Selection IS wall WHERE (description CONTAINS cat OR description
CONTAINS cow) AND description CONTAINS dog;
```

Result of this will be the last two walls (No. 3 and No. 4).

## 3.4 Data creation

Brief example of data creation was already given in the previous example. It is possible to create new objects using BQL. However, this ability is constrained for the simple operations only.

> *Data created with actual BQL implementation are not granted to be valid IFC 2x3 models or valid IFC 2x3 Coordination View models. It is possible to check validity of the data against IFC 2x3 schema but responsibility for the validity is left on the user.*

Syntax for data creation is as follows:

```
creation
      : CREATE creation_statement
      | CREATE CLASSIFICATION STRING
      | IDENTIFIER OP_EQ creation_statement
      ;
creation_statement
      : NEW object STRING
      | NEW object WITH_NAME STRING
      | NEW object WITH_NAME STRING OP_AND DESCRIPTION STRING
      | NEW MATERIAL LAYER_SET STRING ':' layers
      ;
```

You can use this creation statements to create any IFC object defined in the schema. If the object has a name and description attributes defined (like all products and all product types) you can specify that as well. Newly created objects can be assigned to the variable which will than contain set with just the newly created object. Any objects contained in the variable will be removed from the set. If the object is just created but not assigned to the variable it still lives in the model and can be queried. Examples of data creation statements are for example:

```
CREATE NEW wall;
CREATE NEW slab WITH NAME 'My brand new slab' AND DESCRIPTION 'Slab
description';
$NewGroup IS NEW group 'Brand new group';
```

If you want to set more attributes and properties it is possible to do so with data manipulation expressions which are described below in the chapter 3.5. Material layer set creation has slightly different syntax.

```
CREATE NEW MATERIAL LAYER SET 'Set of the dreams': Plaster 10, Bricks
300, Insulation 300, Plaster 10;
```

This will result in a material layer set which you can use in the SET statement to assign it to the element type (like wall type, slab type etc. ) or to the instances of the types (like wall, slab etc. ). If you assign it to the instance rather than type it will be assigned correctly with *IfcMaterialLayerSetUsage* object.

```
$NewLayerSet IS NEW MATERIAL LAYER SET 'Set of the dreams': Plaster 10,
Bricks 300, Insulation 300, Plaster 10;
$NewWall IS NEW wall 'My wall No.1';
```

```
SET MATERIAL to $NewLayerSet FOR $NewWall;
```

## 3.5 Data manipulation

BQL specification contains important part which enables users to manipulate existing data and objects in the model. It is possible to change attributes and properties of the objects and to add or remove objects to or from the group, spatial element (like space or building) and element type (like wall type, slab type and others).

### 3.5.1 Setting attributes and properties

Attributes and properties are important part of the information about the objects in the building model. You can use following expressions to change or set the properties of the objects. If the attribute or property with the name defined already exist in the model it is changed and the type is checked accordingly. If the type doesn't match error is reported. You have to apply the operation on some variable which contains one or more elements. Modification will be applied on all elements according to the rules stated bellow.

```
SET Name TO 'New name', Description TO 'New description', 'Warranty
comment' TO 'No warranty', 'IsExternal' TO FALSE for $walls;
```

If you use '*Name*', '*Description*' or '*Predefined type*' as a name and the type of the object can have an attribute with such a name it will be set. Attribute always takes precedence in other words. In our opinion, there shouldn't be a property with the same name as attribute anyway as it is kind of redundancy. If the property doesn't exist it is created with simple basic type like *IfcLabel* for text, *IfcBool* for boolean values, *IfcInteger* for integer values and *IfcNumericMeasure* for decimal numbers. This is important because some of the predefined property sets might impose specific type. But this is beyond the scope of this scripting interface and it would defeat the purpose of the scripting interface which is to make it easier to access the building data. However, it is possible that standard property sets defined in the IFC schema might be supported including correct types in the future. If such a property set is already present in the model you don't have to care about it.

You can also unset the values of attributes and properties which means their value will be set to NULL which means that the value is not defined. Be just careful what you wish for because the value holds the information about the type. So if you unset the value and set it to some value later on using the script default IFC value type will be used as stated above. This doesn't apply on attributes (Name, description and predefined type) which have a type hard-wired in the IFC schema.

```
SET Name TO NULL, 'Fire ratind' TO NULL FOR $slabs.
```

Newly created properties are created in the property set called *xbim_extended_properties* by default. But you can use the following syntax to save properties and their values in the specified property set. That would be created if it doesn't exist. 'PROPERTY SET' has usual variants '*PropertySet*', '*property set*' and '*property_set*' which are all equal.

```
SET 'My property' to 123, 'IsAllRight' TO TRUE FOR $MyElements IN
PROPERTY SET 'Brand new property set';
```

### 3.5.2 Setting materials of the elements and element types

It was already mentioned earlier that you can use BQL to assign material to the element or element type (see the end of the chapter 3.4). There are several ways how to represent material layer and how to assign it to the element. Especially material layer sets should be treated in a specific way. Actual implementation takes care about this but you should know about it at least for the case you inspect the resulting IFC file.

```
$LayerSet IS NEW MATERIAL LAYER SET 'Composite wall structure': Plaster
10, Bricks 300, Insulation 300, Plaster 10;
$wallType IS NEW wall_type 'First wall type';
$wall IS NEW wall 'My Wall';
$slab IS NEW slab 'My slab';
SET MATERIAL to $LayerSet FOR $wallType;
SET MATERIAL to $LayerSet FOR $slab;
ADD $wall TO $wallType;
```

This script would create new material layer set, wall type and wall. Material layer set would be assigned to the wall type. When you add the wall to the wall type it gets automatically material of the wall assigned as well so you don't need to do it twice. And this is how it should work in IFC world. When the material layer set is assigned to the slab it does something similar and it will fill in the IFC data structure accordingly. So this is just an information to let you know.

### 3.5.3 Adding objects to the group, element type or spatial element

The syntax is the same for all the operations stated in the heading of the chapter. Semantic definition is following:

```
addition
      : ADD IDENTIFIER TO IDENTIFIER
      | REMOVE IDENTIFIER FROM IDENTIFIER
      ;
```

It consist of two identifiers representing variable and *ADD* or *REMOVE* command. For all the operations the first variable is supposed to contain any number of objects which can actually be added or removed according to the IFC schema but it is rather intuitive. Second variable determines type of the operation and it is supposed to contain exactly one object of one of the types (or their subtypes according to the IFC schema): *group* (system is subtype of the group), *type object* (like wall type, slab type, etc.) and *spatial element* (like site, building, building storey or space).

```
$walls IS EVERY wall;
$building IS NEW building 'Default building';
ADD $walls TO $building;
```

Example above would get all walls in the model, create new building object and it would add all the walls to the building using *IfcRelContainedInSpatialStructure* relation.

```
$group IS NEW group WITH NAME '02.02.01' AND DESCRIPTION 'External
walls';
ADD $walls TO $group;
```

This example would create new group and add the walls to the group using *IfcRelAssignsToGroup* relation. You can use the same to add groups to groups which can be used to create hierarchical structures.

```
$wallType IS NEW wall_type 'My wall type';
ADD $walls TO $wallType;
```

This last example would create new wall type and add the walls to the type using *IfcRelDefinedByType* relation. This can be used to define general shared properties, attributes and representations as explained in 3.3.4.

## 3.6 Referenced models

It is sometimes useful to be able to explore more models at once. It is common workflow in many CAD/BIM SW that some of the information is just attached as a reference model/drawing. We have

the same concept built in BQL and xBIM. If there is a model with one or more reference models we call the master model **federation model**. You can use following syntax to reference external models:

```
ADD REFERENCE MODEL 'C:\MyModel.ifc' WHERE ORGANIZATION IS 'Our best QS'
AND OWNER IS 'Bob the Builder';
```

These references are saved within a model using IFC infrastructure for referenced documents. So you can save the file and if you open it again referenced models will be loaded automatically. Documents are referenced using their file path in the file system. So it will break as soon as you move the referenced models.

External models are opened as a read-only files so you can't make any changes to them. Their main purpose is to explore federated models from different owners, companies, suppliers etc. However, you can still make changes to the main model opened. But be careful with changes to the selection which can contain elements from more than one model.

## 3.7 Model handling commands

In order to enable full scripting support and independent execution without user interaction it is possible to open, close and save models from the script using BQL commands like to the following:

```
OPEN MODEL FROM FILE 'testfile.ifc';
CLOSE MODEL;
SAVE MODEL TO FILE 'outputmodel.ifc'
VALIDATE MODEL;
```

BQL interpret recognize the input and output format from the file extension you specify. You can use several input and output IFC formats listed in the table bellow:

| | |
|---|---|
| *.*ifc* | Model formatted as STEP21 file. This is the original IFC text representation. |
| *.*ifcZIP* | Compressed IFC STEP21 file. |
| *.*ifcXML* | XML representation of the model. This format is derived from the original EXPRESS schema. |
| *.*xbim* | XBIM binary format optimized for speed. This is fully compatible with IFC schema and can be converted to any of the standard forms at any time. |

If you close model all variables are deleted and new empty model is created so there is some active model all the time. If you don't save the model to the file all the changes will be lost with no warning.

*VALIDATE* command evaluates just IFC schema compliance but not Coordination View or development agreements. This is left to the user. Command will print the validation message to the output (Console or other output specified via API).

## 3.8 Comments and white spaces

It is always useful to have comments in the script. It makes the script more easy to understand to your colleagues, customers or yourself after some time. Current BQL script can contain only one line comments introduced by the double slash '//'.

```
//This is a single line comment
//which can contain on the next line but must be introduced by '//' again
SELECT EVERY wall WHERE 'warranty' IS DEFINED; //Comment as well
```

By white spaces we mean all spaces, new lines and tabulators in the source text. All white spaces

are insignificant and are skipped except for the white spaces contained in the text enclosed in the single or double quotes. This means you can structure the script easily to make it better readable (see appendix E for example).

```
SET
   Name to 'Sample name',
   Description to 'Sample description',
   'Action required' to TRUE,
FOR $SelectedElements;
```

# 4 Language interpret

BQL language interpret is one of the sample applications within xBIM Toolkit. It is a command line application (don't be afraid of command line – it can be fun!) which runs on top of *Xbim.Script* other and xBIM libraries. It is available as a separate install package called *XbimInterpret*. When you install the application it associates the **.bql* extension to it so that all BQL scripts with **.bql* extension can be executed in their location just by double clicking on it. Or you can launch the console application and enter your commands interactively.

```
Enter your commands (use 'exit' command to quit):
>>> Create new wall 'My first wall';
IfcWall #1: My first wall
>>> Set name to 'My first special wall', description to 'Special description',
'Fire rating' to 123.456 for $$;
>>>Select every wall where 'fire rating' is 123.456;
IfcWall #1: My first special wall
>>>Save model to file 'MyFirstFile.ifc';
>>>
```

# 5 Application Interface

It is also possible to use *Xbim.Script.dll* library with other xBIM Toolkit libraries [5]. Using scripting engine API is extremely easy. See the following example:

```
XbimQueryParser parser = new XbimQueryParser();

//create data using queries
parser.Parse(@"
Create new wall with name 'My wall No. 1' and description 'dog';
Create new wall with name 'My wall No. 2' and description 'cat';
Create new wall with name 'My wall No. 3' and description 'dog and cat';
Create new wall with name 'My wall No. 4' and description 'dog and cow';
Create new wall_type with name 'Wall type No. 1';
Create new system with name 'System No. 1';
Create new system with name 'System No. 2';");

//check data are created
parser.Parse("Select wall;");
Assert.AreEqual(parser.Results["$$"].Count(), 4);
Assert.AreEqual(parser.Errors.Count(), 0);
```

There is just one main class *XbimQueryParser* which can be used to interact with the model using script expressions and commands.

## 5.1 XbimQueryParser

### 5.1.1 Constructors

```
Constructor which takes a existing model as an argument.
You can also close and open any model from the script.

public XbimQueryParser(XbimModel model)


Parameterless constructor of the class.
Default empty model is created which can be used
or you can open other model from the script.

public XbimQueryParser()
```

### 5.1.2 Properties

```
Errors encountered during parsing.
public List<string> Errors { get ; }


Locations of the errors. ErrorLocation contains error message and
location in a structure usable for text selection and other reporting.

public List<ErrorLocation> ErrorLocations { get;}


Messages go to the Console command line normally but
you can use this property to define optional output
where messages should go.

public System.IO.TextWriter Output { get; set; }


Model on which the parser operates
```

```
public XbimModel Model { get; }

Variables which are result of the parsing process.
It can be either list of selected objects or new objects assigned to the variables.

public XbimVariables Results { get; }
```

### 5.1.3 Methods

```
Set source for scanning and parsing

public void SetSource(string source)


Set source for scanning and parsing

public void SetSource(System.IO.Stream source)


Set source for scanning and parsing

public void SetSource(IList<string> source)

Performs only scan of the source and returns list of string
representation of Tokens. This is mainly for debugging.

public IEnumerable<string> ScanOnly()


The main function used to perform parsing of the query.
Returns false only if something serious happen during
parsing process. However it is quite possible that some errors occured.
So, make sure to check Errors if there are any.

public bool Parse()


The main function used to perform parsing of the query.
Returns false only if something serious happen during
parsing process. However it is quite possible that some errors occured.
So, make sure to check Errors if there are any.

public bool Parse(string source)


The main function used to perform parsing of the query.
Returns false only if something serious happen during
parsing process. However it is quite possible that some errors occured.
So, make sure to check Errors if there are any.

public bool Parse(System.IO.Stream source)


The main function used to perform parsing of the query.
Returns false only if something serious happen during
parsing process. However it is quite possible that some errors occured.
So, make sure to check Errors if there are any.

public bool Parse(IList<string> source)
```

## 5.2 Static extensions

Static extension is C# feature which makes it possible to extend existing objects, possibly from different name spaces and assemblies with new methods where you can use these methods as if they

were defined in the definition of the object itself. *Xbim.Script* assembly implements several static extensions which make use of this feature and script functionality. All you need to do to have an access to these methods is to use *Xbim.Script* assembly and name space.

First and most useful extension extends *IfcGroup* so that it can contain embedded script which defines objects which belong to this group. This can be used for the automated classification of the object for example or it can be applied on several models which are referenced in the main model. Execution of the embedded script works as a late evaluation so it will return all objects which satisfy the conditions at the moment of the call. Group members are not defined by the *IfcRelAssignsToGroup* so you have to perform additional operations if you want to make it persistent. It might appear as an interesting feature of the BQL that embedded script can be defined within the BQL script but we don't want to get this text too complicated.

Static extensions of the *IfcGroup* are listed bellow:

| *IfcGroup* | |
|---|---|
| `SetScript(string script)` | Sets the script to the predefined pSet. This can be used for a late evaluation of the group members |
| `string GetScript()` | Gets the script from predefined pSet and property. |
| `bool HasScript()` | Indicates whether the script is defined for this group |
| `IEnumerable<IfcObjectDefinition> ExecuteScript()` | This function will execute the script if there is any defined. If no script is defined this will return empty set. |
| `List<string> GetLastScriptErrors()` | Gets last errors from execution of the script. You should always check this ater you try to get elements using 'ExecuteScript()'. This function is based on a static field so it may give a wrong results in multithreaded environment. |

# 6 Literature and links

[1] Gardens Point Parser Generator (http://gppg.codeplex.com/)

[2] Gardens Point LEX (http://gplex.codeplex.com/)

[3] IFC2x Edition 3 Technical Corrigendum 1 (http://www.buildingsmart-tech.org/ifc/IFC2x3/TC1/html/)

[4] International Framework for Dictionaries (http://www.ifd-library.org/index.php)

[5] The xBIM Toolkit official website (http://xbim.codeplex.com/)

## Appendix A - Language syntax

```
expressions
      : expressions expression
      | expression
      ;

expression
      : selection ';'
      | creation ';'
      | addition ';'
      | attr_setting ';'
      | variables_actions ';'
      | model_actions ';'
      | error
      ;

attr_setting
      : SET value_setting_list FOR IDENTIFIER
      | SET value_setting_list FOR IDENTIFIER IN PROPERTY_SET STRING
      ;

value_setting_list
      : value_setting_list ',' value_setting
      | value_setting
      ;

value_setting
      : attribute TO value
      | STRING TO value
      | MATERIAL TO IDENTIFIER
      ;

value
      : STRING
      | BOOLEAN
      | INTEGER
      | DOUBLE
      | NONDEF
      ;

num_value
      : DOUBLE
      | INTEGER
      ;

model_actions
      : OPEN MODEL FROM FILE STRING
      | CLOSE MODEL
      | VALIDATE MODEL
      | SAVE MODEL TO FILE STRING
      | ADD REFERENCE MODEL STRING WHERE ORGANIZATION OP_EQ STRING OP_AND OWNER
OP_EQ STRING
      /* | COPY IDENTIFIER TO MODEL STRING
      ;

variables_actions
      : DUMP IDENTIFIER
      | CLEAR IDENTIFIER
      | COUNT IDENTIFIER
      | DUMP string_list FROM IDENTIFIER
      | DUMP string_list FROM IDENTIFIER TO FILE STRING
```

```
      ;

string_list
      : string_list ',' STRING
      | string_list ',' attribute
      | STRING
      | attribute
      ;

selection
      : SELECT selection_statement
      | IDENTIFIER op_bool selection_statement
      ;

selection_statement
      : EVERY object
      | EVERY object STRING
      | EVERY object WHERE conditions_set
      ;

creation
      : CREATE creation_statement
      | CREATE CLASSIFICATION STRING
      | IDENTIFIER OP_EQ creation_statement
      ;

creation_statement
      : NEW object STRING
      | NEW object WITH_NAME STRING
      | NEW object WITH_NAME STRING OP_AND DESCRIPTION STRING
      | NEW MATERIAL LAYER_SET STRING ':' layers
      ;

layers
      : layers ',' layer
      | layer
      ;

layer
      : STRING num_value
      ;
```

## Appendix B – Lexical categories

**Identifiers**

```
"$"[a-z$][a-z0-9_]*.........................IDENTIFIER
```

**Operators**

```
"=".........................................|
"equals" |
"is" |
"is equal to"...............................OP_EQ

"!=" |
"is not equal to" |
"is not" |
"does not equal" |
"doesn't equal" |
"doesn't"...................................OP_NEQ

">" |
"is greater than"...........................OP_GT

"<".........................................|
"is less than"..............................OP_LT

">=" |
"is greater than or equal to"...............OP_GTE

"<=" |
"is less than or equal to"..................OP_LTQ

"&&" |
"and"........................................OP_AND

"||" |
"or".........................................OP_OR

"~"..........................................|
"contains" |
"is like"...................................OP_CONTAINS

"!~" |
"does not contain" |
"doesn't contain" |
"is not like" |
"isn't like"................................OP_NOT_CONTAINS

"the same"..................................THE_SAME
"deleted"...................................DELETED
"inserted"..................................INSERTED
"edited"....................................EDITED
```

**Keywords**

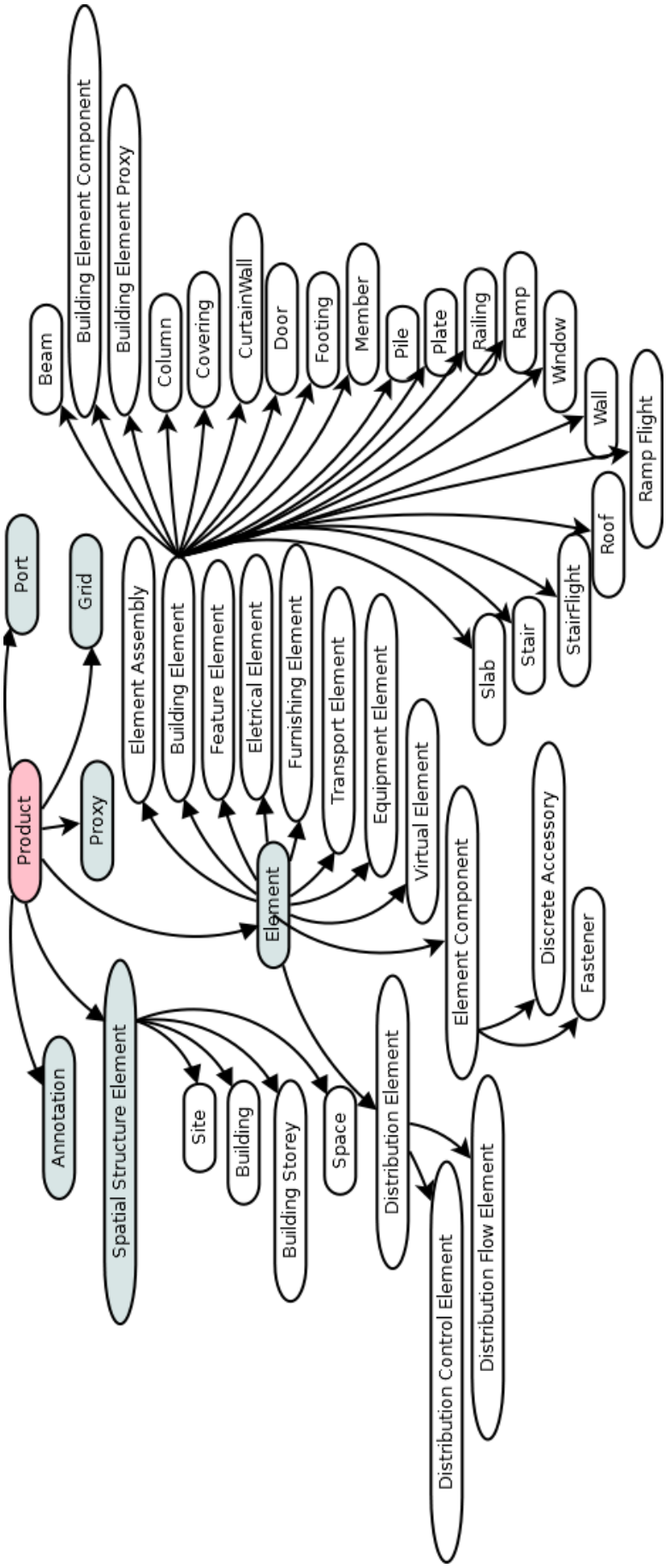
```
"select"....................................SELECT
"set".......................................SET
"for".......................................FOR
"where".....................................WHERE
"create"....................................CREATE
"with name" |
"called"....................................WITH_NAME
"description" |
"described as"..............................DESCRIPTION
"new".......................................NEW
"add".......................................ADD
```

```
"to"....................................TO
"remove"................................REMOVE
"from"..................................FROM
"export" |
"dump"..................................DUMP
"count".................................COUNT
"clear".................................CLEAR
"open"..................................OPEN
"close".................................CLOSE
"save"..................................SAVE
"in"....................................IN
"it"....................................IT
"every".................................EVERY
"validate"..............................VALIDATE
"copy"..................................COPY
"property set" |
"property_set" |
"propertyset"...........................PROPERTY_SET

"name"..................................NAME
........................................
"predefinedtype" |
"predefined_type" |
"predefined type".......................PREDEFINED_TYPE
"type" |
"family"................................TYPE
"material"..............................MATERIAL
"thickness".............................THICKNESS
"file"..................................FILE
"model".................................MODEL
"reference".............................REFERENCE
"classification"........................CLASSIFICATION
"group".................................GROUP
"organization"..........................ORGANIZATION
"owner".................................OWNER
"layer set" |
"layer_set".............................|
"layerset"..............................LAYER_SET

"null" |
"undefined" |
"unknown"...............................NONDEF

"defined"...............................DEFINED
```

## Appendix C - IFC products

Annotation
Beam
Building
Building Element
Building Element Component
Building Element Part
Building Element Proxy
Building Storey
Column
Covering
Curtain Wall
Discrete Accessory
Distribution Control Element
Distribution Element
Distribution Flow Element
Distribution Port
Door
Electric Distribution Point
Electric Flow Storage Device Type
Electrical Element
Element
Element Assembly
Element Component
Energy Conversion Device
Equipment Element
Fastener
Feature Element
Feature Element Addition
Feature Element Subtraction
Flow Controller
Flow Fitting
Flow Moving Device
Flow Segment
Flow Storage Device
Flow Terminal
Flow Treatment Device
Footing
Furnishing Element
Grid
Member
Opening Element
Pile
Plate
Port
Product
Projection Element
Proxy
Railing
Ramp
Ramp Flight
Reinforcing Bar
Reinforcing Element
Reinforcing Mesh
Roof
Site
Slab
Space
Spatial Structure Element
Stair
Stair Flight
Structural Action
Structural Activity
Structural Connection
Structural Curve Connection
Structural Curve Member
Structural Curve Member Varying
Structural Item
Structural Linear Action
Structural Linear Action Varying
Structural Member
Structural Planar Action
Structural Planar Action Varying
Structural Point Action
Structural Point Connection
Structural Point Reaction
Structural Reaction
Structural Surface Connection
Structural Surface Member
Structural Surface Member Varying
Tendon
Tendon Anchor
Transport Element
Virtual Element
Wall
Wall Standard Case
Window

## Appendix D – Product inheritance hierarchy



## Appendix E - Sample script for automated NRM classification

```
Open model from file 'd:\CODE\Sample_IFC\Simple_house.ifc';
Create classification NRM;

//**********************  External Walls ******************************
$extWallsGroup = every system where description = 'External Walls';

//select elements
$extWalls is every IfcCurtainWall;

$extWalls is every wall where
      name contains 'Ext' or
      Function is 1 or
      IsExternal is true or
      type name contains 'Ext' or
      type function is 1 or
      type 'IsExternal' is true
      ;

$extWalls is every covering where
      predefined type is cladding or
      predefined type is insulation or
      type predefined type is cladding
      ;

$extWalls is every member where
      type predefined type is 'STUD'
      ;

$extWalls is every plate where
      type predefined type is 'CURTAIN_PANEL'
      ;

Add $extWalls to $extWallsGroup;


//**********************  Basement Retaining Walls  ************************
$basementWallsGroup is every system where description is 'Basement Retaining
Walls';

$basementWalls is every wall where
      name contains 'Retaining' or
      type name contains 'Retaining'
      ;

Add $basementWalls to $basementWallsGroup;

//***************************  Foundations  ******************************
$foundationsGroup = every system where description is 'Foundations';

$foundations is every IfcWall where
      name contains 'Footing' or
      name contains 'Foundation' or
      type name contains 'Footing' or
      type name contains 'Foundation'
      ;

Add $foundations to $foundationsGroup;

//******************  Internal Walls and Partitions  ***********************
$intWallsGroup is every system where description contains 'Internal Walls and
```

```
Partitions';

$intWalls is every wall where group description is not 'External Walls';

Add $intWalls to $intWallsGroup;

//************************  Ground Floor Construction  ***********************
$groundGroup = every group where description is 'Ground Floor Construction';

$ground is every slab where
      predefined type is 'SLAB' or
      name contains 'GS' or
      name contains 'GND' or
      function is 1 or
      IsExternal is true or
      type predefined type is 'SLAB' or
      type name contains 'GS' or
      type name contains 'GND'
      ;

Add $ground to $groundGroup;

//************************  Upper Floors  ****************************
$upperFloorGroup is every system where description is 'Upper Floors';

$upperFloor is every slab where
      predefined type is 'FLOOR' or
      name contains 'Susp' or
      function is 0 or
      IsExternal is false or
      type predefined type is 'FLOOR' or
      type name contains 'Susp'
      ;

Add $upperFloor to $upperFloorGroup;

//******************  Roof  ***************************************
$roofGroup is every group where description is 'Roof';

$roof is every slab where
      predefined type is 'ROOF' or
      name contains 'Roof' or
      'Name' is 'Roof' or
      type predefined type is 'ROOF'
      ;

$roof is every FlowSegment where
      type predefined type is 'GUTTER' or
      name contains 'gutter'
      ;

Add $roof to $roofGroup;

//************************  Windows and External Doors  ********************
$winDoorGroup is every group where description is 'Windows and External Doors';

$windows is every window;

$doors is every door where
      name contains 'Ext' or
      description contains 'Ext' or
      IsExternal is true or
      Function = 1 or
      Reference contains 'Ext' or
```

```
      type name contains 'Ext' or
      type description contains 'Ext' or
      type IsExternal is true or
      type Function = 1
      ;

Add $windows to $winDoorGroup;
Add $doors to $winDoorGroup;

//*****************  Internal Doors  *********************************
$intDoorGroup is every group where description is 'Internal Doors';

$intDoors is every door where
      group description is not 'Internal Walls and Partitions'
      ;

Add $intDoors to $intDoorGroup;

//**********************  Stairs and Ramps  *********************************
$stairsGroup is every group where description is 'Stairs and Ramps';

$stair is every stair;
$stair is every stair_flight;
$stair is every railing where
      predefined type is 'HANDRAIL' or
      predefined type is 'GUARDRAIL' or
      predefined type is 'NOTDEFINED' or
      name contains 'Handrail' or
      type predefined type is 'HANDRAIL' or
      type predefined type is 'GUARDRAIL' or
      type predefined type is 'NOTDEFINED' or
      type name contains 'Handrail'
      ;

$stair is every member where
      name contains 'stringer' or
      description contains 'stringer' or
      type name contains 'stringer' or
      type description contains 'stringer' or
      type predefined_type is 'STRINGER'
      ;

$stair is every building_element_proxy where
      name contains 'riser' or
      name contains 'tread' or
      type name contains 'riser' or
      type name contains 'tread'
      ;

$stair is every ramp_flight where function is 0;

Add $stair to $stairsGroup;

//**************************  Frame  *********************************
$framingGroup is every group where description is 'Frame';

$framing is every column;

Add $framing to $framingGroup;

//**********************  Ceiling finishes  **************************
$ceilingFinishGroup is every group where description is 'Ceiling finishes';

$ceilingFinish is every covering where
```

```
      predefined type is 'CEILING' or
      type predefined type is 'CEILING'
      ;

Add $ceilingFinish to $ceilingFinishGroup;

//**************  Roads, Paths, Pavings and Surfacings  *****************
$roadsGroup is every group where description is 'Roads, Paths, Pavings and
Surfacings';

$roads is every ramp_flight where function is not 0;

Add $roads to $roadsGroup;

//************  General fittings, furnishings and equipment  ***************
$furnishingGroup is every group where description is 'General fittings,
furnishings and equipment';

$furniture is every IfcFurnishingElement;

Add $furniture to $furnishingGroup;

Save model to file 'Result.ifc';
```