Geometry mapping with `geomtools`

(Software/geomtools/GeometryMappingTutorial – version 0.2)

F. Mauger <`mauger@lpccaen.in2p3.fr`>

2011-12-06

Abstract

In this note, we explain the principles and tools for the creation and manipulation of numbering scheme policies and geometry mapping functionnalities within `geomtools`.

Contents

1 Introduction

To be done...

Subversion repository:
`https://nemo.lpc-caen.in2p3.fr/svn/geomtools/`
DocDB reference: `NemoDocDB-doc-1996`
References: see also *Geometry modelling with* ***`geomtools`*** (`NemoDocDB-doc-1995`)

2 Geometry identifiers

2.1 Presentation of the concept

A *geometry identifier*, also known as *GID* or *geom ID*, is an unique identifier associated to a geometry volume that is part of a geometry hierarchy. The figure 1 shows a virtual geometry setup made of a collection of dictinct volumes (A, B, C, D, E, F, G) placed in a reference frame (doted rectangle). Here some of the volumes (F, G) are included in another one (E); this implies a natural hierarchy relationship between these last 3 volumes : volume E is the mother of volumes F an G; volumes F and G are the daughters of volume E.

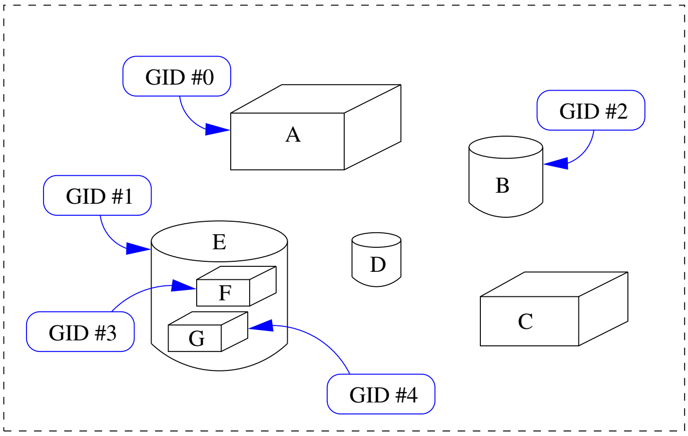
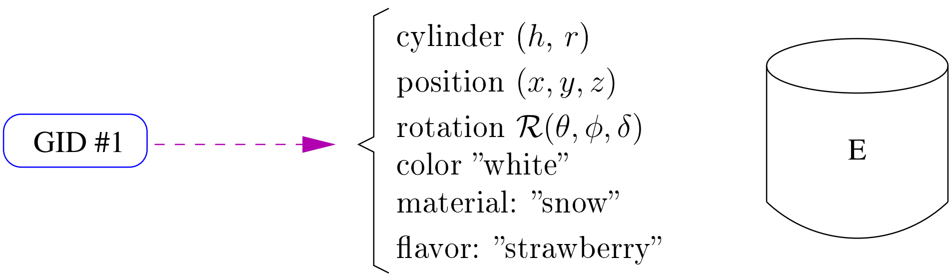


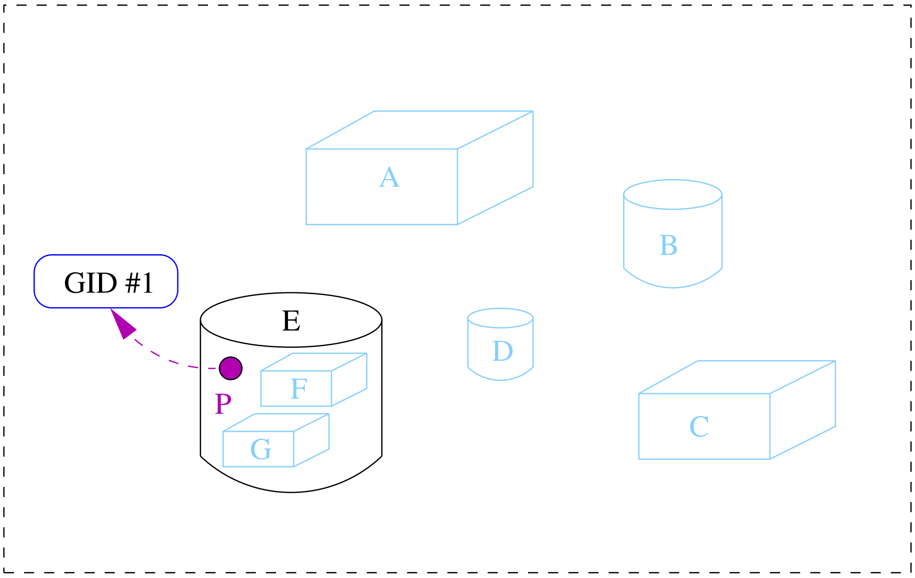
Figure 1: A simple geometry setup.

In an application that depends on and manipulates such a virtual geometry setup, it may be interesting to have access to an unified identification scheme for all or part of the volumes. In the present case, volumes labelled A, B, E, F and G are each associated to an unique *geometry identifier* (blue rounded boxes); on the other hand volumes C and D do not benefit of such association, because it may be not needed in the specific context of the application.
Within the `geomtools` program library, the association of a geometry volume placed in the virtual setup and its GID is called *geometry mapping*. Using such a concept, it is possible to implement some techniques/algorithms that enable for example to:

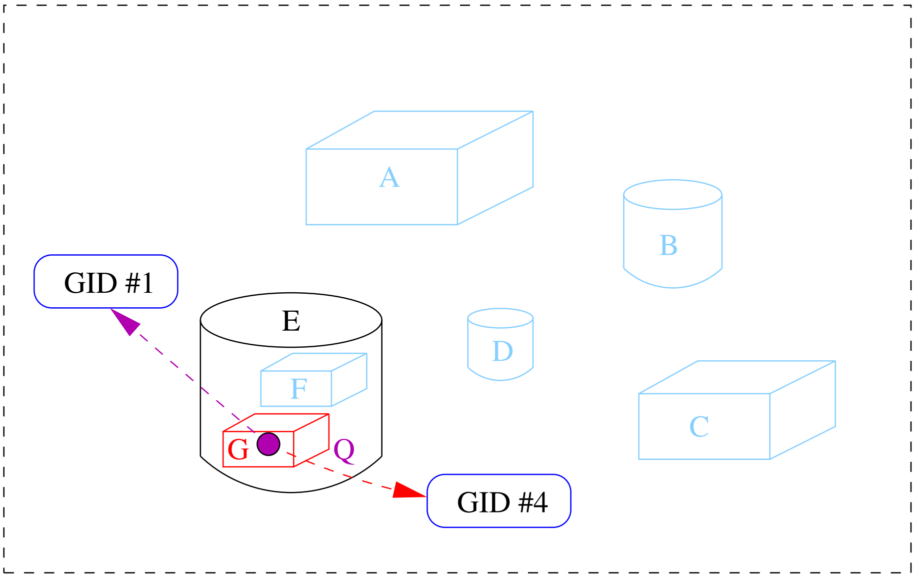
- retrieve the properties of a volume given its GID : placement (position and rotation matrix) in the reference frame (or some arbitrary relative frame), shape, color, material or any useful property in the context of the application.



- find/compute the GID associated to a volume that contains a given position P within the geometry setup :



These techniques can be implemented as *locator* algorithms.
But wait ! The following figure shows the case where the point Q is inside both volumes E **and** G ;



It means that, in general, the determination of the GID associated to the volume where a point lies in can be ambiguous. The final result depends on what the user/application expects or at which depth of the hierarchy the search is performed. In the present case, the ambiguity can be removed if one gives an additional information to the search *locator* algorithm. Such an information could be:

- the maximum depth of the volume hierarchy,
- a specific depth of the volume hierarchy,
- the *category* of the object which is expected to contain the given position Q.

2.2 Design of the `geomtools::geom_id` class

The above considerations give us some hints to design the implementation of a *geometry identifier* class. The minimal useful embedded information to uniquely describe a volume could imply:

- the specification of the *category* of object a volume belongs to: a physical volume is thus interpreted as an *instance* of the *geometry cateogory*,
- some informations that describe the full hierarchical path from the top level mother volume that contains the considered volume to the effective depth of the volume in this hierarchy.

This approach is used in `geomtools` to implement the `geomtools::geom_id` class.
A GID is thus composed of the following attributes :

- the *geometry category* is stored as an `integer` value for storage optimization and fast manipulation. This integer value is named the *type* of the GID (or *geometry type*). Conventionnaly the value `-1` means that the type is not defined (or not valid). Value `0` is reserved to the top level category of the geometry setup [1]. All values greater than `0` can be used to represent the *type* of some volume.
  It is the responsability of the application to manage the classification of different categories of volumes hosted in the setup and associate an unique type integer value to any given category.
- a collection of successive *addresses* that allows to identify/traverse the different levels of the hierarchy that host the corresponding volume. These addresses are implemented as a vector of integers. The size of this vector points out the position depth of the volume in some hierarchical relationship. Each *address* in the vector is an integer value with value `-1` if not defined (not valid) and `0` or more if valid. It is the responsability of the application to allocate a meaning to each address in the collection. It is expected that the ordering of addresses in the collection reflects some hierarchical placement relationship between the volume and its mother volume, grand-mother volume (if any)... this logics is managed by the application.

So, within `geomtools`, instances of the `geomtools::geom_id` class are represented using the following format :

`[TYPE:ADDR0.ADDR1.···.ADDRN]`

where `TYPE` is the value of the type (geometry category) and `ADDRN` are the values of each address at successive depths of the geometry hierarchy.
Examples : `[6:0]`, `[6:1]`, `[6:2]` `[12:2.0]`, `[12:2.1]`, `[18:2.0.3]`, `[18:2.0.4]`.
It should be mentionned that the `geomtools::geom_id` class is a low-level class used to store the raw informations corresponding to the geometry identifier associated to some volume. The choice for using integers as the basic support to store information has been made for convenience and performance both for storage and manipulation. All the *intelligence* that gives some meaning to the values used to store the *type*, as well as the interpretation of the vector of *addresses*, must be managed at higher level. The `geomtools::id_mgr` class is responsible for such features.

[1] in the terminology of the GEANT4 program library, it corresponds to the unique *world volume*

3 More concepts about the use of GIDs

3.1 A simple use case

In order to illustrate the basic concepts used to create and manipulate GIDs with the help of the `geomtools` program library, we present here a simple virtual domestic setup which accomodates several kinds of objects with some simple (and realistic) mother-to-daughter relationship between them.
We start first by the definition of the set of *geometry categories* the objects belongs to. For now, we have 10 of such *type* of objects :

- the *category* "`house`" is associated to the *type* value `1`,
- the *category* "`floor`" is associated to the *type* value `2`,
- the *category* "`room`" is associated to the *type* value `3`,
- the *category* "`table`" is associated to the *type* value `4`,
- the *category* "`chair`" is associated to the *type* value `6`,
- the *category* "`bed`" is associated to the *type* value `9`,
- the *category* "`cupboard`" is associated to the *type* value `12`.
- the *category* "`small_drawer`" is associated to the *type* value `34`.
- the *category* "`large_drawer`" is associated to the *type* value `35`.
- the *category* "`blanket`" is associated to the *type* value `74`.

Category	Type
"`world`"	0
"`house`"	1
"`floor`"	2
"`room`"	3
"`table`"	4
"`chair`"	6
"`bed`"	9
"`cupboard`"	12
"`small_drawer`"	34
"`large_drawer`"	35
"`blanket`"	74

Table 1: The lookup table for *geometry categories* and associated *types*.

The only constraint here is that both *type* values (implemented as integers) and *category* labels (implemented as character strings) are unique within a given context of an

application (i.e. the scope of a particular *geometry ID manager*, see below). We needs here some kind of look-up table with unique *key/value* pairs (table 1).
Then we give the rules that describes the hierarchical relationships between different categories of objects. In our present domestic example, we can make explicit the rules of our virtual domestic world :

- the whole setup (the top-level volume) contains one or more objects of the `"house"` category (*houses*) (here we exclude the case of a top level volume without any house in it : it is indeed of no interest !)
- a *house* contains at least one or several *floors*,
- a *floor* contains at least one or several *rooms*,
- a *room* contains zero or more objects of the following categories : `"chair"`, `"table"`, `"bed"`, `"cupboard"`,
- a *table* can have zero or only one *small drawer*,
- a *cupboard* can have zero or up to 4 *large drawers*,
- a *bed* can host zero or more *blankets*,
- a *blanket*, a *small drawer* or a *large drawer* cannot contains anything; there are the terminal leafs of the hierarchy (they cannot be considered as *containers*).

The figure 2 shows these various categories of objects.

3.2 Different kinds of hierarchical relationships

This domestic example is a good start to investigate the different placement relationships that can be identified in such a virtual model. Let's consider the *world* in figure 3.
We have here only one house with two floors. The ground floor contains two rooms, the first floor has one single room. The large room at ground floor contains three chairs and one table with one unique small drawer. The small room at ground floor has one single chair. The unique room at first floor contains one chair and one bed with one blanket on it. Looks like your place isn't it ?
Now we can make an exhaustive inventory of all the objects that belongs to this world. We can associate to each of them an unique *geometry identifier* :

- the GID of the unique house here has type value `1`. As our world can in principle host several houses, we must allocate a *house number* to this particuliar house. Let's chose house number `666` (the Devil's house !). So the GID of the house can be read : "I'm an object in category `"house"` and my address is fully defined by the `"house_number"`=666". This should lead to the following format: `[1:666]`. The depth of the addresses path of the GID (`666`) is only 1 because only one integer value is enough to distinguish this *Devil's house* from some possible other houses we could add in this virtual world (*Mary's place*, *The Red Lantern*. . .).

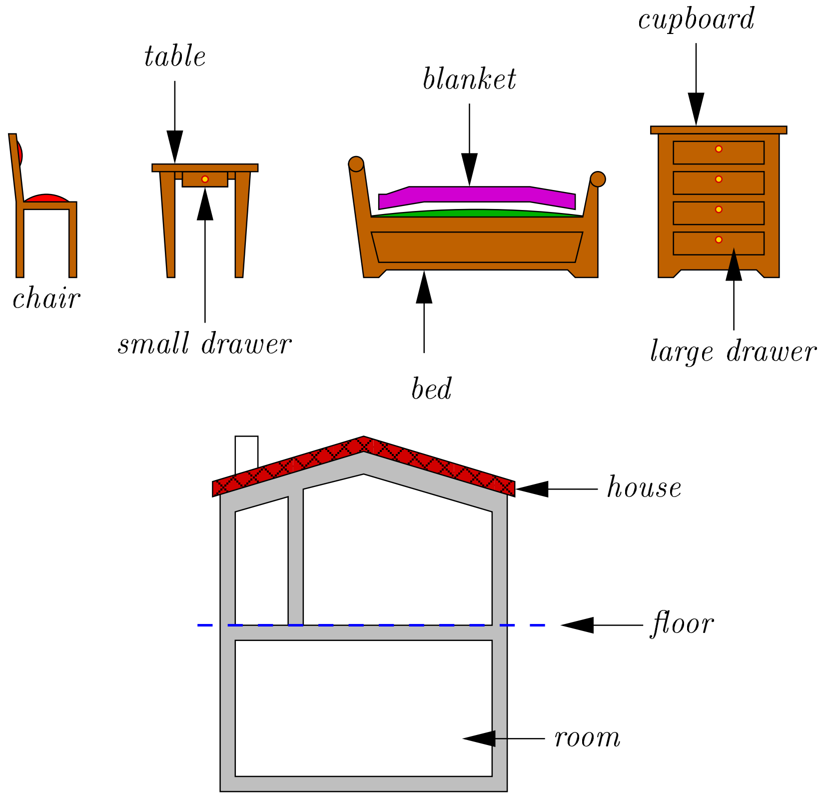


Figure 2: Various categories of objects in a domestic world.

- Let's now consider the ground and first floors. Of course these objects share the same *category*, both are *floors*. The only way to distinguish them is to allocate to them different addresses. The addresses path still has to reflect to fact that they both belong to the *Devil's house* (with `"house_number"`=666). So they must share this hierarchical information. Finally the minimal information needed to distinguish both floor is to allocate and additionnal *floor number* (thus another integer number).
  - The ground floor, or floor with number `0`, can be provided with the following GID: `[2:666.0]`, which reads : "I'm an object of type `2` (so my category is `"floor"`) and I belong to the house of which `"house_number"` is `666`. My `"floor_number"` is `0`"
  - Using a similar scheme, the first floor is given the following GID : `[2:666.1]`, which reads : "I'm an object of type `2` (so my category is `"floor"`) and I belong to the house of which `"house_number"` is `666`. My `"floor_number"` is `1`"

  So the depth of the addresses associated to a *floor* is 2. In the geometry mapping terminology in use in `geomtools`, we say that the `"floor"` category *extends* the `"house"` category (and its `"house_number"` address) *by* the additionnal `"floor_number"` address, leading to a two-levels (or depth) addressing scheme.
- Now we can play the same game for the *rooms*. It is obvious that all 3 rooms in

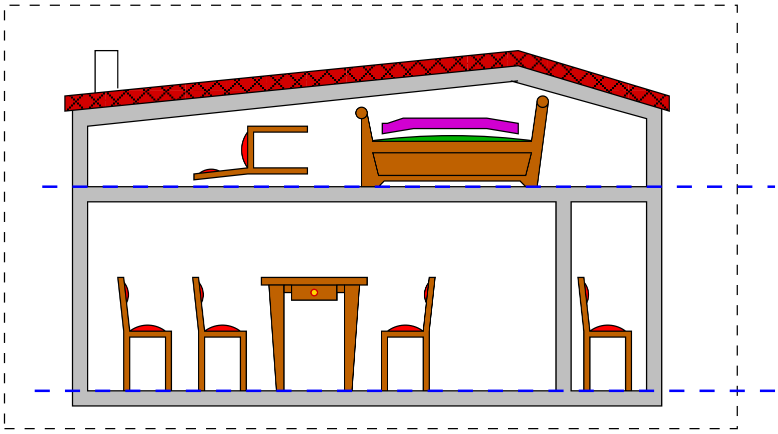


Figure 3: A simple domestic world.

the Devil's house share the same type (set conventionnaly at 3) and belong to the same house ("house_number"=666). However they can be distinguished by two different informations: their "floor_number" and a new level of address: their "room_number" with respect to the floor they lie in. This enables us to allocate a unique GID to each of them :

- Large room at ground floor: GID is [2:666.0.0] where we chose to start the numbering of rooms at this floor with "room_number"=0
- Small room at ground floor: GID is [2:666.0.1] where we chose the next available "room_number"=1 because the former room already used "room_number"=0.
- Unique room at first floor: GID is [2:666.1.0] where we chose to start the numbering of rooms at this a floor with "room_number"=0

So the depth of the addresses associated to a *room* is 3. Here we say that the "room" category *extends* the "floor" category (and its "house_number" and "floor_number" addresses) *by* the additionnal "floor_number" address, leading to a three-levels (or depth) addressing scheme.

- What about chairs, beds and tables ? Again as one room can contains several of these objects, we will have to add additionnal number along the addresses path to uniquely identify them with a GID. This gives the following collection of GID:

  [6:666.0.0.0] : the first chair in the large room at ground floor of the Devil's house where a "chair_number"=0 value has been appended to the addresses path of the mother room with GID [6:666.0.0],

  [6:666.0.0.1] : the second chair in the large room at ground floor of the Devil's house (appended "chair_number"=1),

  [6:666.0.0.2] : the third chair in the large room at ground floor of the Devil's house (appended "chair_number"=2),

  [4:666.0.0.0] : the unique table in the large room at ground floor of the Devil's house (appended "table_number"=0), [6:666.0.1.0] : the unique chair in the small room at ground floor of the Devil's house (appended "chair_number"=0),

[6:666.1.0.0] : the unique chair in the unique room at first floor of the Devil's house (appended "chair_number"=0),

[9:666.1.0.0] : the unique bed in the unique room at first floor of the Devil's house (appended "bed_number"=0),

Should we add another chair in the room at first floor and also a cupboard in the large room at ground floor (figure 4) ?

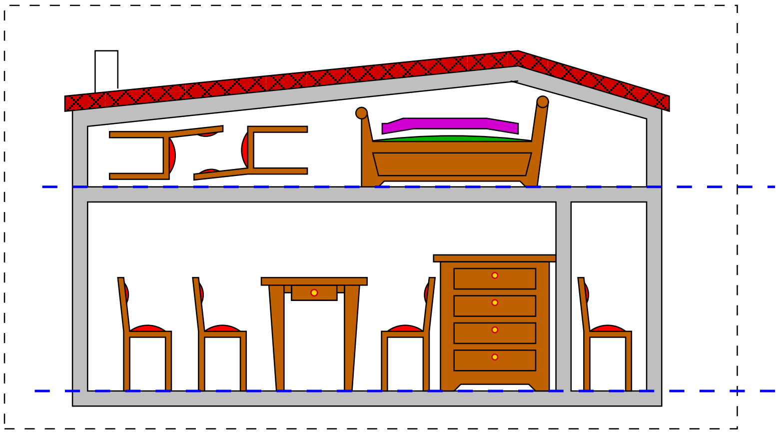


Figure 4: More objects in the domestic world.

No problem ! We allocate to this new chair the GID :

[6:666.1.0.1]

where we have appended the "chair_number"=1. We also allocate to the new cupboard the GID :

[12:666.0.0.0]

where we use the type 12 that corresponds to the "cupboard" category and extends the GID of its mother room (666.0.0) by a "cupboard_number". The "cupboard_number" arbitrarily starts at 0 (in case we would add more and more cupboards in this room : "cupboard_number"=1, 2...).

We have clearly identified one kind of hierarchy relationship between different *categories* of objects in term of the *extension* of the mother container's addresses path by some additionnal address value(s) (integer numbers). These appended numbers enable the distinction between different objects of the same type located in the same container object. This concept is closed to the *mother volume* concept in a hierachical geometry setup and is indeed well adapted to represent such geometry placement relationship.

Now let's consider another situation ! As the placement rules above state it : "a table object can have one and only one small drawer". What does it mean ? It means that to absolutely identify a specific drawer, we only need to know to which table it belongs. There is no need for additionnal information to *locate* it. We say in this case that the "small_drawer" category *inherits* the addressing scheme of the "table" category. So, in the case of the drawer plugged in the ”unique table in the large room at ground floor of the Devil's house”, we can build its GID :

[34:666.0.0.0]

which shares the same 4-levels addresses path of its host/mother table, but differs only by the type identifier (34 for the "small_drawer" category in place of 4 for the "table" category).
This has to be compared with the way we should treat the four "large_drawer" objects that belong to the newly added cupboard in the same room. In this case we must use a *extend by* relationship because an additionnal "drawer_number" is needed to distinguish the 4 possible drawers for a given cupboard. So, in this virtual world, the GID of a "small_drawer" has some 4-levels addresses path but the GID of a "large_drawer" has some 5-levels addresses path. This cleary shows that in this approach, the structure of the numbering scheme attached to an object is not an intrinsic property of the (geoemtrical) nature of the object but an intrinsic property of its relationship with its environment (the way it is placed in some mother object). We can imagine that there is no physical difference (shape, color, dimensions) between the model of the drawer inserted in a table and the model of the drawer inserted in a cupboard. They can have the same physical properties and thus share a common physical description.

3.3 Numbering scheme is an arbitrary choice within an application

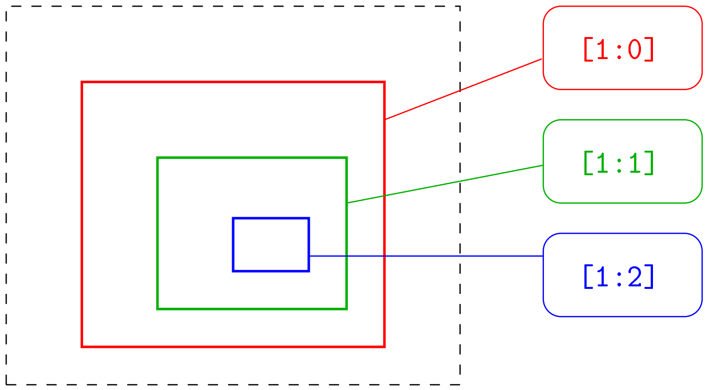
The above considerations enlight the fact that the so-called *geometry category* does not reflect the nature of a physical object with its physical properties but the way it is *inserted* and/or *filled* in its environment made of other objects.
This can be seen on figures below where we can attribute different numbering schemes to the same virtual world, depending on some arbitrary way of thinking :

- Case 1 : Here all box objects are considered to belong to the "any_box" category (with type=1).

Category	Type
"world"	0
"any_box"	1

  We don't take into account the mother/daughter relationship between the red and the green boxes, then between the green and the blue boxes. This corresponds to a *flat* numbering scheme where we don't need to know anything about the internal hierarchy of the geometry setup. Obviously, the numbering scheme does not need to rely on any specific *hierarchy rules* but a single one : the top-level volume (world) contains some objects of category "any_box".

  We get 3 GIDs with the same *type* and some distinct "object_number" (running from 0 to 2) :



To summarize this simple use case, the physical structure of the setup does rely on several levels of nested volumes, but we simply do not consider this geometrical fact if we choose this flat numbering scheme and build the objects' GIDs from this policy.

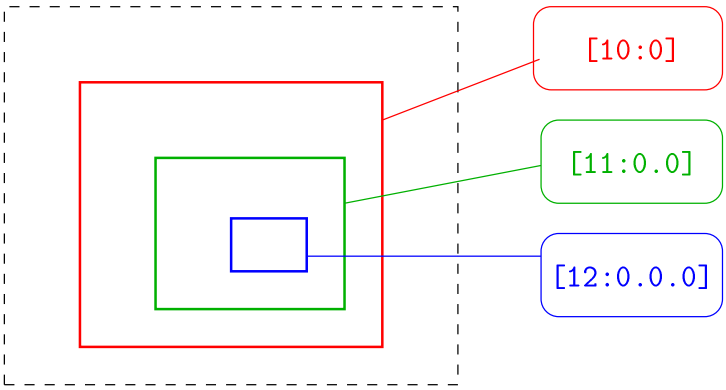
- Case 2 : Here we do take into account the geometry hierarchical relationships between the three box objects. The lookup table now defines 3 different *categories* with 3 associated *types* :

Category	Type
`"world"`	`0`
`"big_box"`	`10`
`"medium_box"`	`11`
`"small_box"`	`12`

Some *hierarchy rules* are also needed :

  - "the top-level volume (world) contains some objects of category `"big_box"`",
  - "an object of category `"big_box"` contains some objects of category `"medium_box"`",
  - "an object of category `"medium_box"` contains some objects of category `"small_box"`".
  - "an object of category `"small_box"` cannot contain anything.

The resulting GIDs associated to the 3 boxes are now :

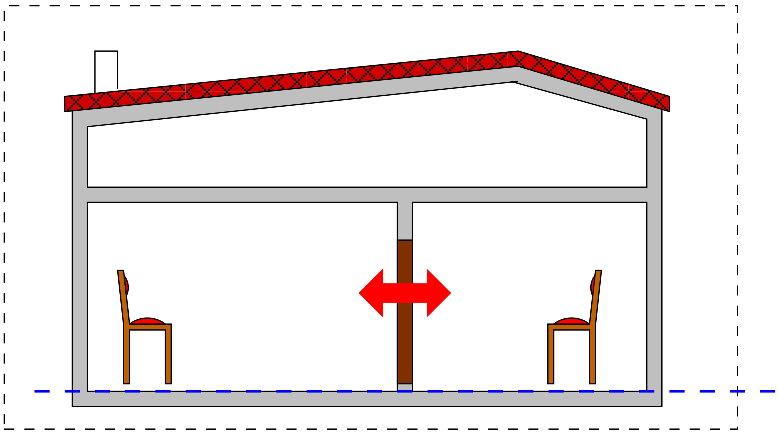


Here GIDs are not only distinguished by their *type* but also by the depth of their addresses pathes, with reflect the hierachical nature of their respective placements. Of course, this way of doing is richer than the *flat* numbering model.

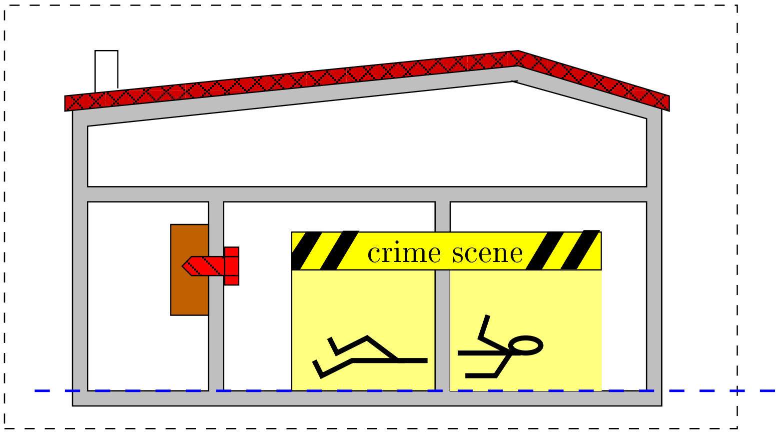
Finally, it turns out that the choice for the numbering scheme strategy (the *mapping*) is the affair of the user and/or application and is not fixed by the physical nature of the objects/components of the virtual model, though it is generally related to it.

3.4 Limitations of this approach

Although the above hierarchical approach is rather powerful and can be naturally implemented to design the numbering scheme of many different virtual geometry setups – usually designed/interpreted with the hierarchical (mother/daughter) approach – there are some (realistic) cases that cannot be addressed or with at least some difficulties or limitations.
The first of these cases is the *door case*. Let's consider again a domestic setup with two adjacent rooms on the same floor of some house.



In the real world, we generally use doors to move from one room to the other. In the physical world, a door can be considered as yet another kind of object in the domectic environment. For example, it is made of wood, has generally a rectangular shape, some dimensions, a mass, can be painted with some arbitrary colour... exactly like a chair. However the way a door is *inserted* within the *geometry* of our world is not obvious on the point of view of the natural *house/floor/room/furniture* hierarchy. The main question is : "Of which room a specific door belongs to ?". It was easy to answer this question for a chair... So we meet here some issues to use our hierarchy model and clearly we will have to invent new concepts to handle such a situation.
Another interesting case has to do with overlapping and/or complex assembly of objects. The figure below shows two of these special cases :



Here we have a red screw that has some parts of its volume in different rooms, another part is inserted in the grey region (house structure); its peak is even inserted in the brown box on the left. The other special case concerns the yellow region (labelled "crime scene" that is splitted in two distinct rooms. Because of the overlapping placements of these objects, it is again not possible to address the problem in term of a simple hierarchical relationship.
In software applications we use for geometry modelling (GEANT4 simulation, GDML modelization, visualization software), such complex situations are generally forbiden by the underlying geometry modeling engine, despite they are not rare in the real world.

4 The GID manager

4.1 Requirements to setup a numbering scheme policy

Within `geomtools`, a class named `geomtools::id_mgr` has been designed to activate a numbering scheme policy for a virtual geometry setup and enable the interpretation of the GID associated to some volumes in this setup. Following the concepts we have investigated in the previous section, a `geomtools::id_mgr` instance is initialized with some special directives that specify :

- a list of *geometry categories* with their uniquely associated *type* values and also a human readable label (the *category* name or label).
- for each *geometry category*, some rules that describes the hierarchical relationships (*inherit/extend to*) with the other categories of objects.

In the current implementation, these configuration informations are stored in a `datatools::utils::multi_properties` container [2] and used to create an internal lookup table (based on the `std::map` class). Each section of this multi-container concerns a specific *geometry category* of which the *name* is the primary key to access useful informations. The section stores some configuration parameters that reflect the *hierarchy rules* between categories.

4.2 Configuration file format

Practically, the `datatools::utils::multi_properties` configuration container is saved as an ASCII file using a human friendly readable format. Let's consider the domestic world explored in the previous sections to illustrate the syntax of this file !

First of all, the header of the file must contain the following meta comments:

```
#@description The description of geometry category in the domestic model
#@key_label   "category"
#@meta_label  "type"
```

Note that the description line is optionnal and the *key* and *label* ones are mandatory.

Then we must add the default top-level category, namely `"world"` which implements a only-one level addresses path with a value labelled `"world"`. The `type` is set at `0` by convention of the library :

```
[category="world" type="0"]
addresses : string[1] = "world"
```

This design could allow in the future to run simultaneously some *parallel* virtual world, each having a different `"world"` number, but sharing the same `"world"` category.

Now it's time to enter the description of the categories of domestic objects. We start with the `"house"` category to which we allocate the type `1` and a 1-level addresses path with a single `"house_number"` value:

[2]seed the "*Using container objects in* `datatool`" tutorial from the `datatools` program library.

```
[category="house" type="1"]
addresses : string[1] = "house_number"
```

Note that the category label "house", as well as the address label "house_number", will be usable by the user through a human friendly interface. This will allow people not to memorize all the integer numbers used in this system and will provide simple methods to retrieve useful geometry informations through character strings.

Now we are done with houses, let's define the rules for the "floor" category with type 2 :

```
[category="floor" type="2"]
extends : string    = "house"
by      : string[1] = "floor_number"
```

which tells (extends) that any object of the "floor" category is contained by another object of the "house" category. More, as any house can contains several floors, we need some additionnal one-level depth appended to the full addresses path. The by directive thus defines the additional "floor_number". Given this rule, the id_mgr instance will automatically use a two-level addresses path for any floor object: the first integer value is the "house_number" inherited from the mother house object, followed by a second integer which corresponds to the "floor_number". Example: [2:666.9] means the floor number 9 in house number 666.

The numbering scheme for the "room", "table", "chair", "bed" and "cupboard" categories is similar:

```
[category="room" type="3"]
extends : string    = "floor"
by      : string[1] = "room_number"
```

this gives GID like: [3:666.9.7] means the room number 7 on floor number 9 in house number 666.

```
[category="table" type="4"]
extends : string    = "room"
by      : string[1] = "table_number"

[category="chair" type="6"]
extends : string    = "room"
by      : string[1] = "chair_number"

[category="bed" type="9"]
extends : string    = "room"
by      : string[1] = "bed_number"

[category="cupboard" type="12"]
extends : string    = "room"
by      : string[1] = "cupboard_number"
```

leading respectively to GIDs like: [4:666.9.7.0], [6:666.9.7.2], [9:666.9.7.1] and [12:666.9.7.0].

The case of the "small_drawer" is special because there can be only one of such object that belongs to a given table. This is specified with:

```
[category="small_drawer" type="34"]
inherits : string    = "table"
```

and gives GIDs like: [34:666.9.7.0] for the unique drawer of table [4:666.9.7.0].

Drawer for cupboard use the *extend by* technique:

```
[category="large_drawer" type="35"]
extends : string    = "cupboard"
by      : string[1] = "drawer_number"
```

and gives GIDs like: [35:666.9.7.0.2] for the drawer numbered 2 in the [34:666.9.7.0] cupboard.

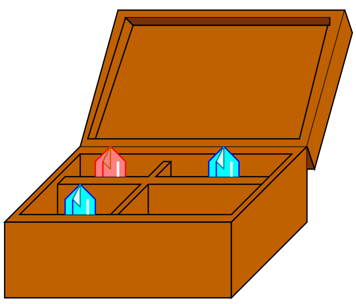
Of course more *category records* can be added to the file if it is needed:

```
[category="fork" type="105"]
extends : string    = "small_drawer"
by      : string[1] = "fork_number"

[category="spoon" type="106"]
extends : string    = "small_drawer"
by      : string[1] = "spoon_number"

[category="knife" type="107"]
extends : string    = "small_drawer"
by      : string[1] = "knife_number"
```

There is also a special useful case. Suppose we are allowed to place a special type of jewelry box in any drawer of a cupboard. We then write a new hierarchy rule :

```
[category="jewelry_box" type="110"]
extends : string    = "large_drawer"
by      : string[1] = "box_number"
```

So far so good. Assume now that each jewelry box can contains up to 4 jewels that can only be placed in one of the four available compartments :



As it can be seen on the above figure, each compartment can be naturally located in a abstract two-dimensional space using the value of its row number **and** the value of its column number. The addresses path of a given jewel will thus be determined first by the addresses path from the box it lies in, then with the information of both row and column numbers. Such a rule can be written with :

```
[category="jewel" type="111"]
extends : string    = "jewelry_box"
by      : string[2] = "row_number" "column_number"
```

where two additionnal numbers (with their human readable labels) have been appended (one-shot extension) to the address path. Of course the choice of ordering the row and column numbers is set conventionally. At least we know how to build the GID of any jewel. Example: `[111:666.9.7.0.2.0.1.0]` reads :
"I'm a (beautiful) jewel hidden in the top/left (`"row"`=`1`, `"column"`=`0`) compartment of the unique jewelry box stored in the drawer number `2` of the only cupboard in the room number `7` on the 9th floor of the Devil's house". Now you have its GID, I guess you are able to steal the treasure !

4.3 Code snippets

4.3.1 Initializing a GID manager object

The following sample program illustrates the initialization of a *GID manager*, using an instance of the `geomtools::id_mgr` class using the `"domestic_categories.lis"` configuration file.

```
// -*- mode: c++ ; -*-
// domestic_1.cxx

#include <cstdlib>
#include <iostream>
#include <exception>

#include <geomtools/id_mgr.h>

int main (void)
{
  int error_code = EXIT_SUCCESS;
  try
    {
      // Declare a GID manager :
      geomtools::id_mgr the_gid_manager;

      // Load the configuration file for categories :
      the_gid_manager.load ("domestic_categories.lis");

      // Print the GID manager's internal lookup table :
      the_gid_manager.tree_dump (std::clog);

    }
  catch (exception & x)
    {
      std::cerr << "error: " << x.what () << endl;
      error_code = EXIT_FAILURE;
    }
   return error_code;
}

// end of domestic_1.cxx
```

The "domestic_categories.lis" file contains the description of the domestic categories we have proposed in the previous section :

```
# domestic_categories.lis

#@description Description of geometry categories
#@key_label   "category"
#@meta_label  "type"

[category="world" type="0"]
addresses : string[1] = "world"

[category="house" type="1"]
addresses : string[1] = "house_number"

[category="floor" type="2"]
extends : string    = "house"
by      : string[1] = "floor_number"

[category="room" type="3"]
extends : string    = "floor"
by      : string[1] = "room_number"

[category="table" type="4"]
extends : string    = "room"
by      : string[1] = "table_number"

[category="chair" type="6"]
extends : string    = "room"
by      : string[1] = "chair_number"

[category="bed" type="9"]
extends : string    = "room"
by      : string[1] = "bed_number"

[category="cupboard" type="12"]
extends : string    = "room"
by      : string[1] = "cupboard_number"

[category="small_drawer" type="34"]
inherits : string    = "table"

[category="large_drawer" type="35"]
extends : string    = "cupboard"
by      : string[1] = "drawer_number"

[category="mailbox" type="40"]
extends : string    = "room"
by      : string[1] = "mailbox_number"

```

```
[category="mailcolumn" type="41"]
extends : string    = "mailbox"
by      : string[1] = "column"

[category="mailrow" type="42"]
extends : string    = "mailcolumn"
by      : string[1] = "row"

# end of domestic_categories.lis
```

The program first parses the file. It constructs an internal lookup table that stores all the informations needed to describe the hierarchical relationships between all kind of objects. Finally it prints the contents of the categories lookup table :

```
|-- Debug  : 0
`-- Categories      : [10]
    |-- Category : "bed"
    |   |-- Category  : "bed"
    |   |-- Type      : 9
    |   |-- Ancestors [3] : "house" "floor" "room"
    |   |-- Extends   : "room" by [1] :  "bed_number"
    |   `-- Addresses [4] : "house_number"
    :                         "floor_number"
    :                           "room_number"
    :                             "bed_number"
    |-- Category : "chair"
    |   |-- Category  : "chair"
    |   |-- Type      : 6
    |   |-- Ancestors [3] : "house" "floor" "room"
    |   |-- Extends   : "room" by [1] :  "chair_number"
    |   `-- Addresses [4] : "house_number"
    :                         "floor_number"
    :                           "room_number"
    :                             "chair_number"
    |-- Category : "cupboard"
    |   |-- Category  : "cupboard"
    |   |-- Type      : 12
    |   |-- Ancestors [3] : "house" "floor" "room"
    |   |-- Extends   : "room" by [1] :  "cupboard_number"
    |   `-- Addresses [4] : "house_number"
    :                         "floor_number"
    :                           "room_number"
    :                             "cupboard_number"
    |-- Category : "floor"
    |   |-- Category  : "floor"
    |   |-- Type      : 2
    |   |-- Ancestors [1] : "house"
    |   |-- Extends   : "house" by [1] :  "floor_number"
    |   `-- Addresses [2] : "house_number"
    :                         "floor_number"
    |-- Category : "house"
    |   |-- Category  : "house"
    |   |-- Type      : 1
    |   `-- Addresses [1] : "house_number"
    |-- Category : "large_drawer"
    |   |-- Category  : "large_drawer"
    |   |-- Type      : 35
    |   |-- Ancestors [4] : "house" "floor" "room" "cupboard"
    |   |-- Extends   : "cupboard" by [1] :  "drawer_number"
    |   `-- Addresses [5] : "house_number"
```

```
    :                          "floor_number"
    :                            "room_number"
    :                              "cupboard_number"
    :                                "drawer_number"
    |-- Category : "room"
    |   |-- Category  : "room"
    |   |-- Type      : 3
    |   |-- Ancestors [2] : "house" "floor"
    |   |-- Extends   : "floor" by [1] :  "room_number"
    |   `-- Addresses [3] : "house_number"
    :                          "floor_number"
    :                            "room_number"
    |-- Category : "small_drawer"
    |   |-- Category  : "small_drawer"
    |   |-- Type      : 34
    |   |-- Inherits  : "table"
    |   |-- Ancestors [4] : "house" "floor" "room" "table"
    |   `-- Addresses [4] : "house_number"
    :                          "floor_number"
    :                            "room_number"
    :                              "table_number"
    |-- Category : "table"
    |   |-- Category  : "table"
    |   |-- Type      : 4
    |   |-- Ancestors [3] : "house" "floor" "room"
    |   |-- Extends   : "room" by [1] :  "table_number"
    |   `-- Addresses [4] : "house_number"
    :                          "floor_number"
    :                            "room_number"
    :                              "table_number"
    `-- Category : "world"
        |-- Category  : "world"
        |-- Type      : 0
        `-- Addresses [1] : "world"
```

4.3.2 Creating some GIDs following the hierarchy rules of a GID manager object

This new sample program uses the previous "`domestic_categories.lis`" configuration file. It creates a specific GID in a given *category* ("`room`") through the human friendly interface of the `geomtools::id_mgr` class. Here a first GID is created from scratch then the GID of a parent object is automatically extracted and finally the GID of a daughter object in a given category is created :

```
// -*- mode: c++ ; -*-
// domestic_2.cxx

#include <cstdlib>
#include <iostream>
#include <exception>

#include <geomtools/id_mgr.h>

int main (void)
{
  int error_code = EXIT_SUCCESS;
  try
    {
      geomtools::id_mgr the_gid_manager;
      the_gid_manager.load ("domestic_categories.lis");

      // Construction of the GID of a 'room' object :
      geomtools::geom_id the_room_gid;
      if (the_gid_manager.has_category_info ("room"))
       {
         // Create a 'room' object from scratch :
         the_gid_manager.make_id ("room", the_room_gid);
         // Allocate the addresses path values at all depths :
         the_gid_manager.set (the_room_gid, "house_number", 666);
         the_gid_manager.set (the_room_gid, "floor_number", 0);
         the_gid_manager.set (the_room_gid, "room_number", 1);
      }

      // Extraction of the GID of the 'floor' mother object :
      geomtools::geom_id the_floor_gid;
      if (the_gid_manager.inherits (the_room_gid, "floor"))
        {
          // Create a mother object from scratch :
          the_gid_manager.make_id ("floor", the_floor_gid);
          // Extract the addresses path of the mother object,
          // removing the trailing "room_number" value:
          the_gid_manager.extract (the_room_gid, the_floor_gid);
        }


```

```
      // Construction of the GID of a 'chair' daughter object :
      geomtools::geom_id the_chair_gid;
      the_gid_manager.make_id ("chair", the_chair_gid);
      if (the_gid_manager.inherits (the_chair_gid, "room"))
        {
          // Inherits the addresses path of the mother :
          the_chair_gid.inherits_from (the_room_gid);
          // Appends an additionnal "chair_number" value
          // to the addresses path :
          the_gid_manager.set (the_chair_gid, "chair_number", 6);
        }

      std::clog << "Reference GID = " << the_room_gid
                << " in category '"
                << the_gid_manager.get_category (the_room_gid)
                << "'" << std::endl;
      std::clog << "Mother GID    = " << the_floor_gid
                << " in category '"
                << the_gid_manager.get_category (the_floor_gid)
                << "'" << std::endl;
      std::clog << "Daughter GID  = " << the_chair_gid
                << " in category '"
                << the_gid_manager.get_category (the_chair_gid)
                << "'" << std::endl;

    }
  catch (exception & x)
    {
      std::cerr << "error: " << x.what () << endl;
      error_code = EXIT_FAILURE;
    }
   return error_code;
}

// end of domestic_2.cxx
```

The program prints :

```
Reference GID = [3:666.0.1] in category 'room'
Mother GID    = [2:666.0] in category 'floor'
Daughter GID  = [6:666.0.1.6] in category 'chair'
```

4.3.3 Creating a large number of GIDs with optimized techniques

The next sample program still uses the previous "`domestic_categories.lis`" configuration file to initialize the GID manager. It accesses to the database of categories through a `geomtools::id_mgr::category_info` object fetched from the GID manager. It then uses the available informations about the hierarchy rules to manipulate GID objects at very low-level, i.e. by direct manipulation of the type value and the values/subaddress of the addresses path at any level :

```
// -*- mode: c++ ; -*-
// domestic_3.cxx

#include <cstdlib>
#include <iostream>
#include <exception>

#include <geomtools/id_mgr.h>

int main (void)
{
  int error_code = EXIT_SUCCESS;
  try
    {
      geomtools::id_mgr the_gid_manager;
      the_gid_manager.load ("domestic_categories.lis");

      if (the_gid_manager.has_category_info ("room"))
       {
         // Construction of the GID of a 'room' object
         // using the corresponding 'category_info' stored
         // in the lookup table.
         const geomtools::id_mgr::category_info & category_room_info
            = the_gid_manager.get_category_info ("room");

         // Get the type associated to the "room" category :
         int room_type = category_room_info.get_type ();

         // Get the indexes of various subaddresses that
         // compose the addresses path :
         int house_number_index =
           category_room_info.get_subaddress_index ("house_number");
         int floor_number_index =
           category_room_info.get_subaddress_index ("floor_number");
         int room_number_index =
           category_room_info.get_subaddress_index ("room_number");

         // Massive generation of GIDs of the "room" category
         // for 3 houses, each with 2 floors with 4 rooms per floor :
         for (int house_number = 666; house_number < 669; house_number++)
           {
```

```
              for (int floor_number = 0; floor_number < 2; floor_number++)
                {
                  for (int room_number = 0; room_number < 4; room_number++)
                    {
                      // Instantiate a GID :
                      geomtools::geom_id the_room_gid;

                      // Set the type of the GID :
                      the_room_gid.set_type (room_type);

                      // Allocate the addresses path values at all depths :
                      the_room_gid.set (house_number_index, house_number);
                      the_room_gid.set (floor_number_index, floor_number);
                      the_room_gid.set (room_number_index, room_number);

                      std::clog << "Room has GID = " << the_room_gid
                                << std::endl;
                    }
                }
            }
        }

    }
  catch (exception & x)
    {
      std::cerr << "error: " << x.what () << endl;
      error_code = EXIT_FAILURE;
    }
   return error_code;
}

// end of domestic_3.cxx
```

The program prints :

```
Room has GID = [3:666.0.0]
Room has GID = [3:666.0.1]
Room has GID = [3:666.0.2]
Room has GID = [3:666.0.3]
Room has GID = [3:666.1.0]
Room has GID = [3:666.1.1]
Room has GID = [3:666.1.2]
Room has GID = [3:666.1.3]
Room has GID = [3:667.0.0]
Room has GID = [3:667.0.1]
Room has GID = [3:667.0.2]
Room has GID = [3:667.0.3]
Room has GID = [3:667.1.0]
Room has GID = [3:667.1.1]
Room has GID = [3:667.1.2]
```

```
Room has GID = [3:667.1.3]
Room has GID = [3:668.0.0]
Room has GID = [3:668.0.1]
Room has GID = [3:668.0.2]
Room has GID = [3:668.0.3]
Room has GID = [3:668.1.0]
Room has GID = [3:668.1.1]
Room has GID = [3:668.1.2]
Room has GID = [3:668.1.3]
```

Such a technique should be favored when one needs to manipulate a large number of GIDs or even a few GIDs very frequently. Indeed the human-friendly methods provided by the GID manager (`geomtools::id_mgr`) class are based on searching algorithms in various associative containers keyed by `string` objects. If these methods are often used, some performance issues are expected. It is thus more efficient to directly use the integer values for *types* and *addresses* indexes, rather than human friendly string labels. Here the human-friendly methods are used once at the beginning of the program to retrieve useful addressing informations stored as integer values (category *types* and address index); then it is straightforward to reuse this addressing parameters a large number of times, without further request through the human friendly interface of the GID manager.

5 Geometry mapping and associated tools

In the previous section, we have presented the basic concepts (GID, hierarchy rules, GID manager) used in the `geomtools` library. Now it is time to introduce some high-level functionnalities that can be implemented on top of these low-level concepts : *geometry mapping* and *locators*.

5.1 Geometry mapping

The key concept of *geometry mapping* is to allow the users to benefit of some automated (or semi-automated) database of all (or part of) the objects that belongs to a geometry hierarchical setup. Such a database will naturally use the object's GID as the primary keys for accessing some meta-data associated to an object.

As seen in the previous sections, it is possible to define some non ambiguous *hierarchy rules* to reflect the mother/daughter relationships between objects of a virtual geometry setup. This is a task for the GID manager object, which is available in the library.

However, when designing the numbering scheme, there are still many arbitrary choices that have to be made by the architect/developper of the numbering scheme built on top of the geometry model : "Do we start the values of subaddresses from `0` or `1` when several replicates of some category are placed in a given mother volume ?", "What value is chosen for the subaddress of the left part of this tracking chamber : O or 1 ?", "What integer values are associated to the North, South, West and East directions ?"... So we need some additional conventions and rules to finalize the addressing scheme. Then we will be able to establish and build the full list of GIDs that makes sense in our application.

If we consider the above *domestic* virtual world, we have implicitely used such rules on top of the hierarchy rules. We now need some tools to automatically inform the geometry model of these rules. Let's build such a virtual setup with the tools provided by `geomtools`. Then we will see how to enrich this model with special *mapping directives*.

5.2 Mapping directives

Mapping directives explains the rules and conventions to be respected by dedicated algorithms that are responsible for the automatic generation of GIDs associated to the physical volumes of a geometry hierarchy. These algorithms first need a GID manager object (`geomtools::id_mgr` class) in order to know the layout of the numbering scheme to be applied to the geometry hierarchy. On a second step, mapping directives are used to build the GID associated to all physical volumes requested by users and associate each GID with some specific informations.

The `geomtools::mapping` class implement such an algorithm. Given a *model factory* and a *GID manager*, this *mapping algorithm* establishes a list of all requested GIDs and associates them with some useful geometry informations :

- the placement (position and rotation matrix) of the physical object in the *world* volume;
- a reference to the *logical volume* it refers to,

- a list of auxiliary properties.

This results in the creation of a – possibly large – dictionnary that contains the geometry information requested for some physical volumes. The GID of a given volume is used as the primary key to access the informations from this `GID database`.

Practically, the `geomtools` library proposes to enrich the description of the geometry models that take part to the geometry setup by adding some additionnal properties dedicated to the *mapping* functionnality.

From the *geometry modelling tutorial* we have learnt to write the description of a *geometry model*. In the following example, we recognize a few configuration directives for a *stacked* geometry model : material, length unit, description of the geometry models to be stacked along some axis (properties starting with the "`stacked.`" prefix). We also recognize visualization directives, starting with the "`visibility.`" prefix :

```
[name="stacked_box" type="geomtools::stacked_model"]
material.ref            : string  = "__default__"
length_unit             : string  = "cm"
stacked.axis            : string  = "z"
stacked.number_of_items : integer = 4
stacked.model_0         : string  = "blue_cylinder"
stacked.label_0         : string  = "stacked_0"
stacked.model_1         : string  = "huge_red_box"
stacked.label_1         : string  = "stacked_1"
...
visibility.hidden       : boolean   = 0
visibility.color        : string    = "grey"
...
```

The *mapping* directives will use a similar grammar. By convention, they will start with the "`mapping.`" prefix. We will see below what is the syntax for these directives. But first let's come back to some concept we have explored so far.

We have shown previously that the GID attached to a *physical volume* is not an intrinsic property of the associated *logical volume* from which the physical volume is built/modelled and thus not a property of the corresponding *geometry model*. The point here is that it is the action to place a logical volume in some mother volume that *instantiates* the physical volume. The GID is thus instantiated too at this step. It means that the description of the mother volume should contain the mapping directives for its daughter volumes.

Practically, there is only one useful mapping directive that is implemented in `geomtools`, namely "`mapping.daughter_id`". It is used in a parameterized way, i.e. is must be appended with some special string label (with an additionnal dot character):

```
mapping.daughter_id.<XXXXX> : string = "<MAPPING DIRECTIVE>"
```

where :

- `<XXXXX>` is a label that identifies non-ambiguously one of the daughter volumes contained in the current geometry model, i.e. the *mother*,

- `<MAPPING DIRECTIVE>` is a character string that gives the rules to build the GID associated to this daughter volume.

  The allowed syntaxes for `<MAPPING DIRECTIVE>` are:

  - for volumes belonging to some *inherited* geometry categories :

    `[<CATEGORY NAME>]`

  - for volumes belonging to some *extended* geometry categories :

    `[<CATEGORY NAME>:<SUBADDRESS NAME 1><OP 1><SUBADDRESS VALUE 1>(,<SUBADDRESS NAME 2><OP 2><SUBADDRESS VALUE 2>)]`

  where :

  - `<CATEGORY NAME>` is the name of a geometry category known by the *GID manager*
  - `<SUBADDRESS NAME 1>` is the name of a subaddress that is part of the addresses path description known by the *GID manager*
  - `<OP 1>` is the `=` ot the `+` operator,
  - `<SUBADDRESS VALUE 1>` is an integer value (>=0)
  - the`(,<...>)` indicates that the numbering scheme accept more similar rules.

Examples in the context of the domestic virtual setup :

- `[house:house_number=666]` : an object in the "`house`" category of which the "`house_number`" is set explicitely at value `666`,

- `[floor:floor_number+0]` : an object in the "`floor`" category of which the "`floor_number`" is autoincremented from starting value `0`; the `house_number` of the GID is supposed to be automatically set because the "`floor`" category extends the "`house`" so the GID of this floor object will use the `house_number` of the mother house.

- `[jewel:row_number=0,column_number=1]` : an object in the "`jewel`" category of which the "`row_number`" is set explicitely at value `0`, and the "`column_number`" is set explicitely at value `1`; here again the `box_number` of the GID is supposed to be automatically set because the "`jewel`" category extends the "`jewelry_box`" so the GID of this *jewel* object will use the `box_number` of the mother box.

- `[small_drawer]` : an object in the "`small_drawer`" category of which the "`table_number`" is automatically inherited from the GID of its mother table because the "`small_drawer`" category inherits the `table` category.

5.3 A toy model

Let's come back to the "Devil's house" ! In the geometry model shown on figure 5, we setup a virtual world which contains one single house with two floors.

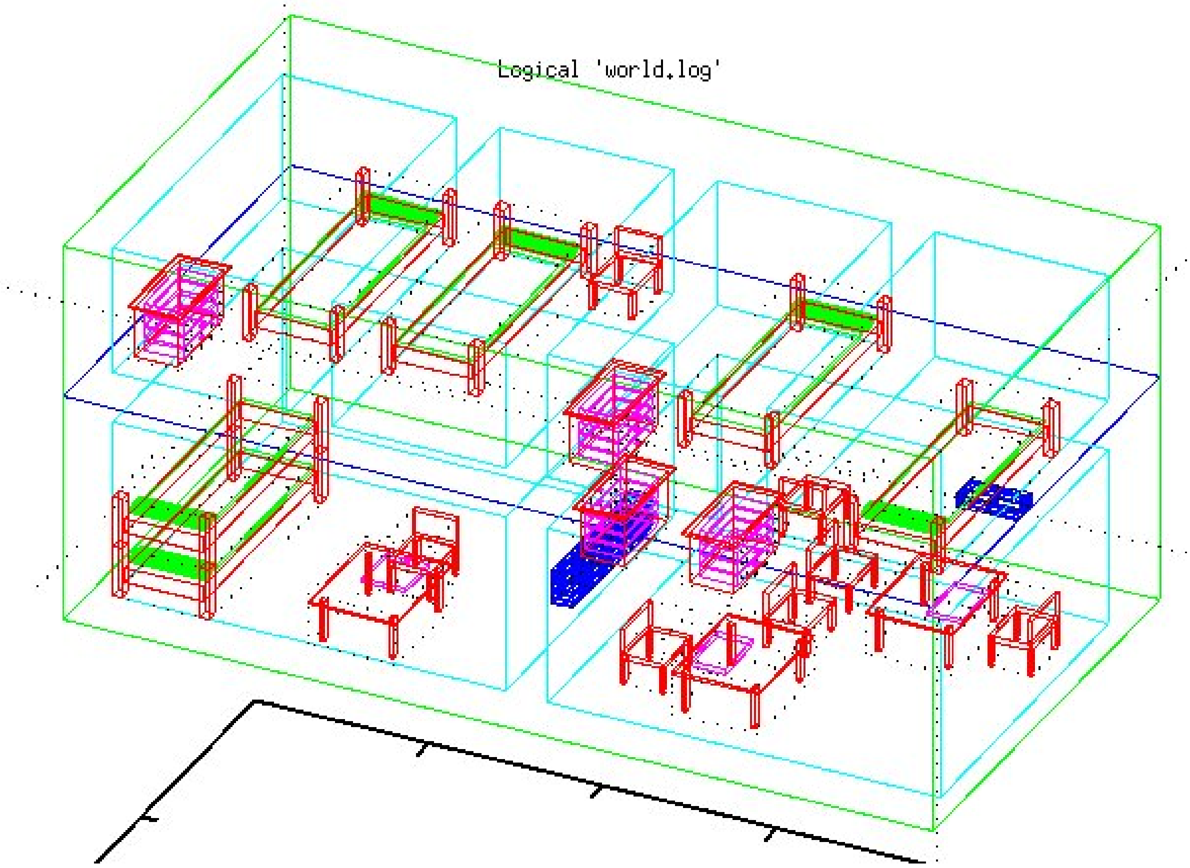


Figure 5: The Devil's house geometry model. Rooms display in cyan, furniture objects (bed, table, chair...) in red, drawer in magenta, mailboxes in blue.

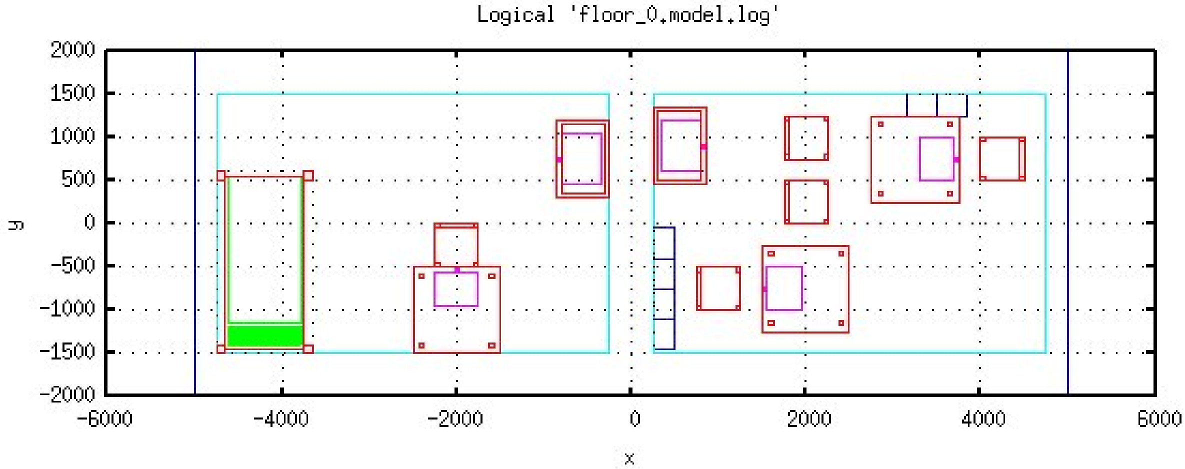


Figure 6: Top view of the ground floor of the Devil's house.

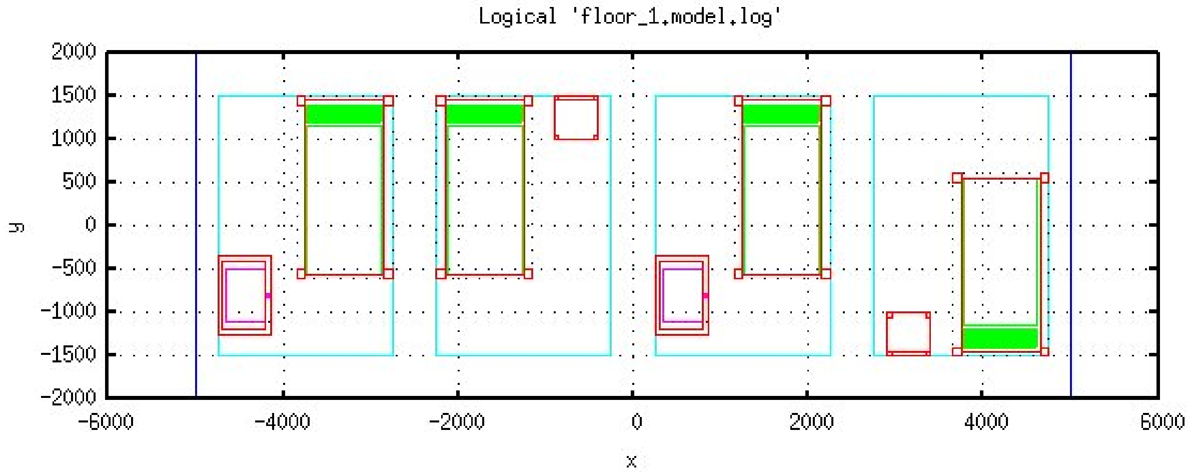


Figure 7: Top view of the first floor of the Devil's house.

5.3.1 Syntax

There are 2 rooms on the ground floor (figure 6) and 4 rooms on the first floor (figure 7). All rooms contain some furnitures like beds, chairs, cupboards, table, mailboxes. These objects can be automatically associated to GIDs through a geometry *mapping* algorithm. The list of categories is defined in the file shown in sample 1.

The geometry configuration corresponding file is shown in appendix A. Each section related to a geometry model with some daughter volumes to be identified with a GID is *decorated* with *mapping* directives. For example the sample 2 illustrates how one attributes a GID to the unique house of this world : this GID belongs to the "`house`" category and its one-level address is simply made of the house number (labelled "`house_number`" in the sample file 1) which is arbitrarily chosen to `666`.

Another similar example is shown on the sample 3. It sets the mapping directive for a *floor* models.

In the case of the replicated placement of some similar daughter volumes, another syntax is used as shown on sample 4. Here, the "`small_mailbox`" model being composed of the replication of 2 columns of compartments, the mapping directive "`column+0`" specifies that the "`column`" number is incremented automatically from the initial value `0`. The "`small_mailbox_column`" model uses a similar syntax ("`row+0`") to identify the `row` number of the mail compartments it contains.

Finally, another syntax is used to number the drawer associated to a given table (figure 8).

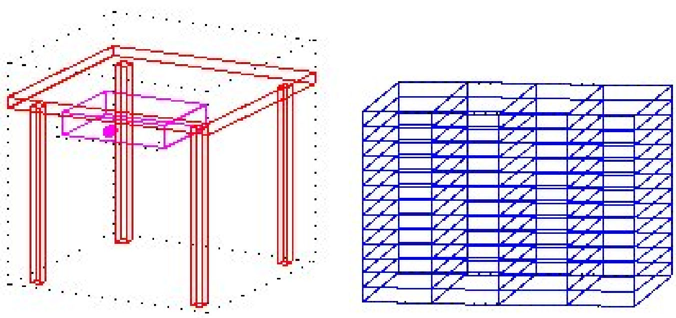


Figure 8: Left: the model of a table with its unique daughter drawer; right: the model of some mailboxes that corresponds to the replication of 4 columns each of 13 vertically replicated compartments.

This is shown on sample 5; it is here sufficient to specify the only category to which the drawer is associated, because it *inherits* the addresses path of its mother table.

```
#@description Description of geometry categories
#@key_label   "category"
#@meta_label  "type"

[category="world" type="0"]
addresses : string[1] = "world"

[category="house" type="1"]
addresses : string[1] = "house_number"

[category="floor" type="2"]
extends : string    = "house"
by      : string[1] = "floor_number"

[category="room" type="3"]
extends : string    = "floor"
by      : string[1] = "room_number"

[category="table" type="4"]
extends : string    = "room"
by      : string[1] = "table_number"

[category="chair" type="6"]
extends : string    = "room"
by      : string[1] = "chair_number"

[category="bed" type="9"]
extends : string    = "room"
by      : string[1] = "bed_number"

[category="cupboard" type="12"]
extends : string    = "room"
by      : string[1] = "cupboard_number"

[category="small_drawer" type="34"]
inherits : string    = "table"

[category="large_drawer" type="35"]
extends : string    = "cupboard"
by      : string[1] = "drawer_number"

[category="mailbox" type="40"]
extends : string    = "room"
by      : string[1] = "mailbox_number"

[category="mailcolumn" type="41"]
extends : string    = "mailbox"
by      : string[1] = "column"

[category="mailrow" type="42"]
extends : string    = "mailcolumn"
by      : string[1] = "row"
```

Sample 1: The geometry categories in the *Devil's house world*. Note that the categories are introduced from the top of the hierarchy to the final leaves.

```
[name="world" type="geomtools::simple_world_model"]
material.ref    : string = "vacuum"
setup.model     : string = "house.model"

angle_unit      : string = "degree"
setup.phi       : real   = 0.0
setup.theta     : real   = 0.0

length_unit     : string = "m"
setup.x         : real   = 0.0
setup.y         : real   = 0.0
setup.z         : real   = 0.0
world.x         : real   = 14.0
world.y         : real   = 10.0
world.z         : real   = 14.0

visibility.hidden            : boolean = 0
visibility.daughters.hidden : boolean = 0
```

Sample 2: The mapping directive for the *house* model placed in the *world* model.

```
[name="house.model" type="geomtools::stacked_model"]
length_unit       : string = "m"
material.ref      : string = "concrete"

visibility.hidden : boolean = 0
visibility.color  : string  = "green"

stacked.axis            : string  = "z"
stacked.number_of_items : integer = 2
stacked.label_1         : string  = "first_floor"
stacked.model_1         : string  = "floor_1.model"
stacked.label_0         : string  = "ground_floor"
stacked.model_0         : string  = "floor_0.model"

mapping.daughter_id.ground_floor : string = "[floor:floor_number=0]"
```

Sample 3: The mapping directives for the *floor* volumes contained in the *house* model.

```
[name="small_mailbox.model" type="geomtools::replicated_boxed_model"]
replicated.axis            : string  = "y"
replicated.number_of_items : integer = 2
replicated.model           : string  = "small_mailbox_column.model"
replicated.label           : string  = "mail_column"

visibility.hidden : boolean = 0
visibility.color  : string  = "grey"

mapping.daughter_id.mail_column : string = "[mailcolumn:column+0]"
```

Sample 4: The mapping directive for the *small mailbox column* volumes contained in the *small mailbox* model.

```
[name="table.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "box"
x            : real   = 100.0
y            : real   = 100.0
z            : real   = 100.0
length_unit  : string = "cm"
material.ref : string  = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "grey"

internal_item.labels : string[6] = \
                         "drawer" \
                         "leg_0" "leg_1" "leg_2" "leg_3" \
                         "plateau"

internal_item.model.drawer      : string  = "small_drawer.model"
internal_item.placement.drawer  : string  = "+27.5 0 +25 (cm)"

mapping.daughter_id.drawer : string = "[small_drawer]"
```

Sample 5: The mapping directive for the *small drawer* object contained in the *table* model.

5.3.2 Automatic GIDs generation

Once the geometry configuration file has been enriched with mapping directives, it is possible to ask a `geomtooms::mapping` algorithm to build the list of all (or part) of the GIDs corresponding to the objects of the hierarchy.

The sample program 1-2 illustrates the combined use of:

- a *geometry factory* which builds the virtual geometry setup from a geometry description file; the list of *geometry models* handled by the factory is given in sample 6
- a *GID manager* which handles a list of geometry categories loaded from a category description file,
- a *mapping* algorithm which reads the mapping directives associated to the models and create a dictionary of *geometry informations* associated to all geometry objects requested by some criteria; here we choose to :
  - not limit the generation of the dictionary to some maximum depth in the hierarchy:

    ```
    the_mapping.set_max_depth (geomtools::mapping::NO_MAX_DEPTH);
    ```

  - not generate the entries that correspond to the objects identified as mail compartments (the "`mailrow`" category) and columns of mail compartments (the "`mailcolumn`" category) :

    ```
    the_mapping.add_excluded ("mailrow");
    the_mapping.add_excluded ("mailcolumn");
    ```

    The list of GIDs associated to the generated *geometry information* entries is given id sample 7.
- a *Gnuplot drawer* which display a 2D or 3D view of the virtual geometry setup.

```
#include <cstdlib>
#include <iostream>
#include <stdexcept>

#include <geomtools/model_factory.h>
#include <geomtools/gnuplot_drawer.h>
#include <geomtools/id_mgr.h>
#include <geomtools/mapping.h>
#include <geomtools/placement.h>

int main (int argc_, char ** argv_)
{
  int error_code = EXIT_SUCCESS;
  try
    {
      std::string view_mode;
      std::string geom_file;
      std::string category_file;
      std::string model_name;
      int iarg = 1;
      while (iarg < argc_)
        {
          std::string token = argv_[iarg];
          if (token == "-3d") view_mode = geomtools::gnuplot_drawer::VIEW_3D;
          else if (token == "-xy") view_mode = geomtools::gnuplot_drawer::VIEW_2D_XY;
          else if (token == "-xz") view_mode = geomtools::gnuplot_drawer::VIEW_2D_XZ;
          else if (token == "-yz") view_mode = geomtools::gnuplot_drawer::VIEW_2D_YZ;
          else if (geom_file.empty ()) geom_file = token;
          else if (category_file.empty ()) category_file = token;
          else if (model_name.empty ()) model_name = token;
          ++iarg;
        }
      if (view_mode.empty ()) view_mode = geomtools::gnuplot_drawer::VIEW_3D;
      if (geom_file.empty ()) throw std::logic_error ("Missing .geom filename !");
      if (category_file.empty ()) throw std::logic_error ("Missing category filename !");
      if (model_name.empty ()) model_name = "world";

      // Declare a model factory :
      geomtools::model_factory the_model_factory;
      the_model_factory.load (geom_file);
      the_model_factory.lock ();

      // Print the list of geometry models :
      clog << "List of models: " << endl;
      int count = 0;
      for (geomtools::models_col_t::const_iterator i
             = the_model_factory.get_models ().begin ();
           i != the_model_factory.get_models ().end ();
           i++)
        {
          std::clog << "  " << i->second->get_name () << std::endl;
        }
      std::clog << std::endl;
```

Program 1: A program with embedded *geometry mapping* (part 1/2).

```
      // Declare the GID manager :
      geomtools::id_mgr the_gid_manager;
      the_gid_manager.load (category_file);

      // Declare the mapping manager :
      geomtools::mapping the_mapping;
      the_mapping.set_id_manager (the_gid_manager);
      // Do not limit the depth of the numbering scheme trhough the hierarchy :
      the_mapping.set_max_depth (geomtools::mapping::NO_MAX_DEPTH);
      // Do not generate GID entries for some specific geometry categories :
      the_mapping.add_excluded ("mailrow");
      the_mapping.add_excluded ("mailcolumn");
      the_mapping.build_from (the_model_factory);
      std::clog << "Congratulations ! Mapping has been built !" << std::endl;
      std::clog << "List of available GIDs : " << std::endl;
      for (geomtools::geom_info_dict_t::const_iterator i
             = the_mapping.get_geom_infos ().begin ();
           i != the_mapping.get_geom_infos ().end ();
           ++i)
        {
          std::cout << "  Geom ID : " << i->first << std::endl;
        }

      // Declare a Gnuplot renderer :
      geomtools::gnuplot_drawer GPD;
      GPD.set_view (view_mode);
      geomtools::placement reference_placement;
      reference_placement.set (0 * CLHEP::m, 0 * CLHEP::m, 0 * CLHEP::m,
                               0 * CLHEP::degree, 0 * CLHEP::degree, 0);
      GPD.draw (the_model_factory,
                model_name,
                reference_placement,
                geomtools::gnuplot_drawer::DISPLAY_LEVEL_NO_LIMIT);

    }
  catch (exception & x)
    {
      std::cerr << "error: " << x.what () << std::endl;
      error_code = EXIT_FAILURE;
    }
   return error_code;
}

// end of gmanager.cxx
```

Program 2: A program with embedded *geometry mapping* (part 2/2).

```
bed.model
bed_blanket.model
bed_body.model
bed_leg.model
bolster.model
chair.model
chair_back.model
chair_leg.model
chair_seat.model
cupboard.model
cupboard_body.model
cupboard_plateau.model
drawer_handle.model
floor_0.model
floor_1.model
house.model
large_drawer.model
large_drawer_box.model
large_mail_column.model
large_mailbox.model
mail_box.model
room_0.model
room_0_with_tables.model
room_1.model
room_1_with_bed.model
small_drawer.model
small_drawer_box.model
small_mailbox.model
small_mailbox_column.model
table.model
table_leg.model
table_plateau.model
twin_bed.model
world
```

Sample 6: The list of *geometry models* handled by the factory in program 1.

```
Geom ID : [0:0]
Geom ID : [1:666]
Geom ID : [2:666.0]
Geom ID : [2:666.1]
Geom ID : [3:666.0.0]
Geom ID : [3:666.0.1]
Geom ID : [3:666.1.0]
Geom ID : [3:666.1.1]
Geom ID : [3:666.1.2]
Geom ID : [3:666.1.3]
Geom ID : [4:666.0.1.0]
Geom ID : [4:666.0.1.1]
Geom ID : [6:666.0.1.0]
Geom ID : [6:666.0.1.1]
Geom ID : [6:666.0.1.2]
Geom ID : [6:666.0.1.3]
Geom ID : [6:666.1.1.0]
Geom ID : [6:666.1.3.0]
Geom ID : [9:666.0.0.0]
Geom ID : [9:666.0.0.1]
Geom ID : [9:666.1.0.0]
Geom ID : [9:666.1.1.0]
Geom ID : [9:666.1.2.0]
Geom ID : [9:666.1.3.0]
Geom ID : [12:666.0.0.0]
Geom ID : [12:666.0.1.0]
Geom ID : [12:666.1.0.0]
Geom ID : [12:666.1.2.0]
Geom ID : [34:666.0.1.0]
Geom ID : [34:666.0.1.1]
Geom ID : [35:666.0.0.0.0]
Geom ID : [35:666.0.0.0.1]
Geom ID : [35:666.0.0.0.2]
Geom ID : [35:666.0.0.0.3]
Geom ID : [35:666.0.1.0.0]
Geom ID : [35:666.0.1.0.1]
Geom ID : [35:666.0.1.0.2]
Geom ID : [35:666.0.1.0.3]
Geom ID : [35:666.1.0.0.0]
Geom ID : [35:666.1.0.0.1]
Geom ID : [35:666.1.0.0.2]
Geom ID : [35:666.1.0.0.3]
Geom ID : [35:666.1.2.0.0]
Geom ID : [35:666.1.2.0.1]
Geom ID : [35:666.1.2.0.2]
Geom ID : [35:666.1.2.0.3]
Geom ID : [40:666.0.1.7]
Geom ID : [40:666.0.1.9]
```

Sample 7: The list of *geometry information* entries in program 2. Here the GID corresponding to the "`mailrow`" (type=42) and "`mailcolumn`" (type=41) have not been generated because of a the use of some special mapping exclusion directives (see text).

6 Using geometry information from the mapping

To be done.

7 Conclusion

To be done...

A The toy model geometry configuration file

This is the geometry configuration file for the toy model from section 5.3 :

```
# -*- mode: conf-unix; -*-
# domestic_models.geom

#@description List of domestic geometry models
#@key_label   "name"
#@meta_label  "type"

#################################################################

[name="chair_leg.model" type="geomtools::simple_shaped_model"]
shape_type   : string =  "box"
x            : real   =  5.0
y            : real   =  5.0
z            : real   = 50.0
length_unit  : string = "cm"
material.ref : string  = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "red"

[name="chair_back.model" type="geomtools::simple_shaped_model"]
shape_type   : string =  "box"
x            : real   =  5.0
y            : real   = 50.0
z            : real   = 50.0
length_unit  : string = "cm"
material.ref : string  = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "red"

[name="chair_seat.model" type="geomtools::simple_shaped_model"]
shape_type   : string =  "box"
x            : real   = 50.0
y            : real   = 50.0
z            : real   = 10.0
length_unit  : string = "cm"
material.ref : string  = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "red"

[name="chair.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "box"
x            : real   = 50.0
y            : real   = 50.0
z            : real   = 110.0
length_unit  : string = "cm"
material.ref : string  = "air"

visibility.hidden : boolean = 0
visibility.color  : string  = "grey"
```

```
internal_item.labels : string[6] = \
                         "leg_0" "leg_1" "leg_2" "leg_3" \
                         "seat" \
                         "back"

internal_item.model.leg_0       : string  = "chair_leg.model"
internal_item.placement.leg_0   : string  = "+22.5 +22.5 -30 (cm)"

internal_item.model.leg_1       : string  = "chair_leg.model"
internal_item.placement.leg_1   : string  = "+22.5 -22.5 -30 (cm)"

internal_item.model.leg_2       : string  = "chair_leg.model"
internal_item.placement.leg_2   : string  = "-22.5 -22.5 -30 (cm)"

internal_item.model.leg_3       : string  = "chair_leg.model"
internal_item.placement.leg_3   : string  = "-22.5 +22.5 -30 (cm)"

internal_item.model.seat        : string  = "chair_seat.model"
internal_item.placement.seat    : string  = "0 0 0 (cm)"

internal_item.model.back        : string  = "chair_back.model"
internal_item.placement.back    : string  = "-22.5 0 +30 (cm)"


#################################################################

[name="drawer_handle.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "cylinder"
r            : real   = 2.0
z            : real   = 5.0
length_unit  : string = "cm"
material.ref : string = "aluminium"

visibility.hidden : boolean = 0
visibility.color  : string  = "magenta"

[name="small_drawer_box.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "box"
x            : real   = 40.0
y            : real   = 50.0
z            : real   = 10.0
length_unit  : string = "cm"
material.ref : string = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "magenta"

[name="small_drawer.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "box"
x            : real   = 45.0
y            : real   = 50.0
z            : real   = 10.0
length_unit  : string = "cm"
material.ref : string = "air"

```

```
visibility.hidden : boolean = 0
visibility.color  : string  = "grey"

internal_item.labels           : string[2] = "box" "handle"
internal_item.model.box        : string    = "small_drawer_box.model"
internal_item.placement.box    : string    = "-2.5 0 0 (cm)"
internal_item.model.handle     : string    = "drawer_handle.model"
internal_item.placement.handle : string    = "20 0 0 (cm) @ 0 90 (degree)"

###################################################################

[name="bed_blanket.model" type="geomtools::simple_shaped_model"]
shape_type   : string =  "box"
x            : real   = 170.0
y            : real   = 85.0
z            : real   = 10.0
length_unit  : string = "cm"
material.ref : string  = "whool"

visibility.hidden : boolean = 0
visibility.color  : string  = "green"

###################################################################

[name="bed_leg.model" type="geomtools::simple_shaped_model"]
shape_type   : string =  "box"
x            : real   = 10.0
y            : real   = 10.0
z            : real   = 100.0
length_unit  : string = "cm"
material.ref : string  = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "red"

[name="bed_body.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "box"
x            : real   = 200.0
y            : real   = 90.0
z            : real   = 30.0
length_unit  : string = "cm"
material.ref : string = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "red"

[name="bolster.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "cylinder"
r            : real   = 10.0
z            : real   = 85.0
length_unit  : string = "cm"
material.ref : string = "cotton"

visibility.hidden : boolean = 0
visibility.color  : string  = "green"

```

```
[name="bed.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "box"
x            : real   = 210.0
y            : real   = 110.0
z            : real   = 100.0
length_unit  : string = "cm"
material.ref : string  = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "grey"

internal_item.labels : string[7] = \
                         "leg_0" "leg_1" "leg_2" "leg_3" \
                         "body" "bolster" "blanket"

internal_item.model.leg_0       : string  = "bed_leg.model"
internal_item.placement.leg_0   : string  = "+100 +50 0 (cm)"

internal_item.model.leg_1       : string  = "bed_leg.model"
internal_item.placement.leg_1   : string  = "+100 -50 0 (cm)"

internal_item.model.leg_2       : string  = "bed_leg.model"
internal_item.placement.leg_2   : string  = "-100 -50 0 (cm)"

internal_item.model.leg_3       : string  = "bed_leg.model"
internal_item.placement.leg_3   : string  = "-100 +50 0 (cm)"

internal_item.model.body        : string  = "bed_body.model"
internal_item.placement.body    : string  = "0 0 -10 (cm)"

internal_item.model.bolster        : string  = "bolster.model"
internal_item.placement.bolster    : string  = "+85 0 +15 (cm) @ 90 90 (degree)"

internal_item.model.blanket        : string  = "bed_blanket.model"
internal_item.placement.blanket    : string  = "-15 0 +10 (cm)"

[name="twin_bed.model" type="geomtools::replicated_boxed_model"]
replicated.axis            : string  = "z"
replicated.number_of_items : integer = 2
replicated.model           : string  = "bed.model"
replicated.label           : string  = "stacked_bed"

visibility.hidden : boolean = 0
visibility.color  : string  = "grey"

mapping.daughter_id.stacked_bed : string = "[bed:bed_number+0]"


###################################################################

[name="table_leg.model" type="geomtools::simple_shaped_model"]
shape_type   : string =  "box"
x            : real   =  5.0
y            : real   =  5.0
z            : real   = 80.0
length_unit  : string = "cm"
```

```
material.ref : string  = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "red"


[name="table_plateau.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "box"
x            : real   = 100.0
y            : real   = 100.0
z            : real   = 5.0
length_unit  : string = "cm"
material.ref : string  = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "red"


[name="table.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "box"
x            : real   = 100.0
y            : real   = 100.0
z            : real   = 100.0
length_unit  : string = "cm"
material.ref : string  = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "grey"

internal_item.labels : string[6] = \
                         "drawer" \
                         "leg_0" "leg_1" "leg_2" "leg_3" \
                         "plateau"

internal_item.model.drawer      : string  = "small_drawer.model"
internal_item.placement.drawer  : string  = "+27.5 0 +25 (cm)"

mapping.daughter_id.drawer : string = "[small_drawer]"

internal_item.model.leg_0       : string  = "table_leg.model"
internal_item.placement.leg_0   : string  = "+40 +40 -10 (cm)"

internal_item.model.leg_1       : string  = "table_leg.model"
internal_item.placement.leg_1   : string  = "+40 -40 -10 (cm)"

internal_item.model.leg_2       : string  = "table_leg.model"
internal_item.placement.leg_2   : string  = "-40 -40 -10 (cm)"

internal_item.model.leg_3       : string  = "table_leg.model"
internal_item.placement.leg_3   : string  = "-40 +40 -10 (cm)"

internal_item.model.plateau     : string  = "table_plateau.model"
internal_item.placement.plateau : string  = "0 0 +32.5 (cm)"
```

```
##################################################################

[name="large_drawer_box.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "box"
x            : real   = 45.0
y            : real   = 60.0
z            : real   = 20.0
length_unit  : string = "cm"
material.ref : string = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "magenta"

[name="large_drawer.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "box"
x            : real   = 50.0
y            : real   = 60.0
z            : real   = 20.0
length_unit  : string = "cm"
material.ref : string = "air"

visibility.hidden : boolean = 0
visibility.color  : string  = "grey"

internal_item.labels           : string[2] = "box" "handle"
internal_item.model.box        : string    = "large_drawer_box.model"
internal_item.placement.box    : string    = "-2.5 0 0 (cm)"
internal_item.model.handle     : string    = "drawer_handle.model"
internal_item.placement.handle : string    = "22.5 0 0 (cm) @ 0 90 (degree)"

##################################################################

[name="cupboard_body.model" type="geomtools::simple_shaped_model"]
shape_type   : string =  "box"
x            : real   =  50.0
y            : real   =  80.0
z            : real   = 100.0
length_unit  : string = "cm"
material.ref : string = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "red"


[name="cupboard_plateau.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "box"
x            : real   = 60.0
y            : real   = 90.0
z            : real   = 5.0
length_unit  : string = "cm"
material.ref : string = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "red"


```

```
[name="cupboard.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "box"
x            : real   = 60.0
y            : real   = 90.0
z            : real   = 100.0
length_unit  : string = "cm"
material.ref : string  = "wood"

visibility.hidden : boolean = 0
visibility.color  : string  = "grey"

internal_item.labels : string[6] =  \
                         "body"     \
                         "plateau"  \
                         "drawer_0" \
                         "drawer_1" \
                         "drawer_2" \
                         "drawer_3"

internal_item.model.body        : string  = "cupboard_body.model"
internal_item.placement.body    : string  = "0 0 0 (cm)"

internal_item.model.plateau     : string  = "cupboard_plateau.model"
internal_item.placement.plateau : string  = "0 0 +52.5 (cm)"

internal_item.model.drawer_0      : string  = "large_drawer.model"
internal_item.placement.drawer_0  : string  = "5 0 -37.5 (cm)"

internal_item.model.drawer_1      : string  = "large_drawer.model"
internal_item.placement.drawer_1  : string  = "5 0 -12.5 (cm)"

internal_item.model.drawer_2      : string  = "large_drawer.model"
internal_item.placement.drawer_2  : string  = "5 0 +12.5 (cm)"

internal_item.model.drawer_3      : string  = "large_drawer.model"
internal_item.placement.drawer_3  : string  = "5 0 +37.5 (cm)"

mapping.daughter_id.drawer_0 : string = "[large_drawer:drawer_number=0]"
mapping.daughter_id.drawer_1 : string = "[large_drawer:drawer_number=1]"
mapping.daughter_id.drawer_2 : string = "[large_drawer:drawer_number=2]"
mapping.daughter_id.drawer_3 : string = "[large_drawer:drawer_number=3]"

###################################################################

[name="mail_box.model" type="geomtools::simple_shaped_model"]
shape_type   : string = "box"
x            : real   = 25.0
y            : real   = 35.0
z            : real   =  5.0
length_unit  : string = "cm"
material.ref : string = "aluminium"

visibility.hidden : boolean = 0
visibility.color  : string  = "blue"

[name="large_mail_column.model" type="geomtools::replicated_boxed_model"]
```

```
replicated.axis            : string  = "z"
replicated.number_of_items : integer = 13
replicated.model           : string  = "mail_box.model"
replicated.label           : string  = "mail_box"

visibility.hidden : boolean = 0
visibility.color  : string  = "grey"

mapping.daughter_id.mail_box : string = "[mailrow:row+0]"

[name="small_mailbox_column.model" type="geomtools::replicated_boxed_model"]
replicated.axis            : string  = "z"
replicated.number_of_items : integer = 5
replicated.model           : string  = "mail_box.model"
replicated.label           : string  = "mail_box"

visibility.hidden : boolean = 0
visibility.color  : string  = "grey"

mapping.daughter_id.mail_box : string = "[mailrow:row+0]"

[name="large_mailbox.model" type="geomtools::replicated_boxed_model"]
replicated.axis            : string  = "y"
replicated.number_of_items : integer = 4
replicated.model           : string  = "large_mail_column.model"
replicated.label           : string  = "mail_column"

visibility.hidden : boolean = 0
visibility.color  : string  = "grey"

mapping.daughter_id.mail_column : string = "[mailcolumn:column+0]"

[name="small_mailbox.model" type="geomtools::replicated_boxed_model"]
replicated.axis            : string  = "y"
replicated.number_of_items : integer = 2
replicated.model           : string  = "small_mailbox_column.model"
replicated.label           : string  = "mail_column"

visibility.hidden : boolean = 0
visibility.color  : string  = "grey"

mapping.daughter_id.mail_column : string = "[mailcolumn:column+0]"

###################################################################

[name="room_0.model" type="geomtools::simple_shaped_model"]
shape_type        : string = "box"
x                 : real   = 4.5
y                 : real   = 3.0
z                 : real   = 3.5
length_unit       : string = "m"
material.ref      : string = "air"

visibility.hidden : boolean = 0
visibility.color  : string  = "cyan"

```

```
internal_item.labels : string[4] = \
  "beds_a" "cupboard_a" "table_a" "chair_a"

internal_item.model.beds_a      : string  = "twin_bed.model"
internal_item.placement.beds_a  : string  = "-1.70 -0.45 -0.75 (m) / z 270 (degree)"

internal_item.model.cupboard_a     : string  = "cupboard.model"
internal_item.placement.cupboard_a : string  = "+1.95 +0.75 -1.2 (m) / z 180 (degree)"

internal_item.model.table_a     : string  = "table.model"
internal_item.placement.table_a  : string  = "0.5 -1.0 -1.25 (m) / z 90 (degree)"

internal_item.model.chair_a      : string  = "chair.model"
internal_item.placement.chair_a  : string  = "0.5 -0.25 -1.2 (m) / z 270 (degree)"

#mapping.daughter_id.cupboard_X : string = "[cupboard:cupboard_number=0]"
mapping.daughter_id.cupboard_a : string = "[cupboard:cupboard_number=0]"

[name="room_0_with_tables.model" type="geomtools::simple_shaped_model"]
shape_type        : string = "box"
x                 : real   = 4.5
y                 : real   = 3.0
z                 : real   = 3.5
length_unit       : string = "m"
material.ref      : string = "air"

visibility.hidden : boolean = 0
visibility.color  : string  = "cyan"

internal_item.labels : string[9] = \
  "table_U" \
  "table_V" \
  "chair_A" \
  "chair_B" \
  "chair_C" \
  "chair_D" \
  "cupboard_X" \
  "mailbox_Y" \
  "mailbox_Z"

internal_item.model.table_U      : string  = "table.model"
internal_item.placement.table_U  : string  = "-0.5 -0.75 -1.25 (m) / z 180 (degree)"

internal_item.model.table_V      : string  = "table.model"
internal_item.placement.table_V  : string  = "0.75 +0.75 -1.25 (m)"

internal_item.model.chair_A      : string  = "chair.model"
internal_item.placement.chair_A  : string  = "-1.5 -0.75 -1.2 (m)"

internal_item.model.chair_B      : string  = "chair.model"
internal_item.placement.chair_B  : string  = "+1.75 +0.75 -1.2 (m) / z 180 (degree)"

internal_item.model.chair_C      : string  = "chair.model"
internal_item.placement.chair_C  : string  = "-0.5 +1.0 -1.2 (m)"

internal_item.model.chair_D      : string  = "chair.model"
```

```
internal_item.placement.chair_D  : string  = "-0.5 +0.25 -1.2 (m)"

internal_item.model.cupboard_X     : string  = "cupboard.model"
internal_item.placement.cupboard_X : string  = "-1.95 +0.9 -1.2 (m)"

internal_item.model.mailbox_Y      : string  = "large_mailbox.model"
internal_item.placement.mailbox_Y  : string  = "-2.125 -0.75 +0.5 (m)"

internal_item.model.mailbox_Z      : string  = "small_mailbox.model"
internal_item.placement.mailbox_Z  : string  = "1. 1.375 +0.35 (m)/ z 90 (degree)"

mapping.daughter_id.table_U : string = "[table:table_number=0]"
mapping.daughter_id.table_V : string = "[table:table_number=1]"
mapping.daughter_id.chair_A : string = "[chair:chair_number=0]"
mapping.daughter_id.chair_B : string = "[chair:chair_number=1]"
mapping.daughter_id.chair_C : string = "[chair:chair_number=2]"
mapping.daughter_id.chair_D : string = "[chair:chair_number=3]"
mapping.daughter_id.cupboard_X : string = "[cupboard:cupboard_number=0]"
mapping.daughter_id.mailbox_Y  : string = "[mailbox:mailbox_number=9]"
mapping.daughter_id.mailbox_Z  : string = "[mailbox:mailbox_number=7]"

[name="room_1.model" type="geomtools::simple_shaped_model"]
shape_type        : string = "box"
x                 : real   = 2.0
y                 : real   = 3.0
z                 : real   = 2.5
length_unit       : string = "m"
material.ref      : string = "air"

visibility.hidden : boolean = 0
visibility.color  : string  = "cyan"

internal_item.labels : string[2] = \
  "cupboard_a" \
  "bed_a"

internal_item.model.cupboard_a    : string  = "cupboard.model"
internal_item.placement.cupboard_a : string  = "-0.70 -0.8 -0.75 (m)"

internal_item.model.bed_a      : string  = "bed.model"
internal_item.placement.bed_a  : string  = "+0.45 0.45 -0.750 (m) / z 90 (degree)"

mapping.daughter_id.bed_a   : string = "[bed:bed_number=0]"
mapping.daughter_id.cupboard_a : string = "[cupboard:cupboard_number=0]"

[name="room_1_with_bed.model" type="geomtools::simple_shaped_model"]
shape_type        : string = "box"
x                 : real   = 2.0
y                 : real   = 3.0
z                 : real   = 2.5
length_unit       : string = "m"
material.ref      : string = "air"

visibility.hidden : boolean = 0
visibility.color  : string  = "cyan"

```

```
internal_item.labels : string[2] = "bed_a" "chair_a"

internal_item.model.bed_a      : string  = "bed.model"
internal_item.placement.bed_a  : string  = "-0.45 0.45 -0.750 (m) / z 90 (degree)"

internal_item.model.chair_a      : string  = "chair.model"
internal_item.placement.chair_a  : string  = "+0.6 1.25 -0.70 (m) / z 270 (degree)"

mapping.daughter_id.bed_a   : string = "[bed:bed_number=0]"
mapping.daughter_id.chair_a : string = "[chair:chair_number=0]"

#######################################################################

[name="floor_0.model" type="geomtools::simple_shaped_model"]
shape_type        : string = "box"
x                 : real   = 10.0
y                 : real   = 4.0
z                 : real   = 4.5
length_unit       : string = "m"
material.ref      : string = "concrete"

visibility.hidden : boolean = 0
visibility.color  : string  = "blue"

internal_item.labels : string[2] = "room_Y" "room_Z"

internal_item.model.room_Y       : string  = "room_0.model"
internal_item.placement.room_Y   : string  = "-2.5 0 -0.5 (m)"

internal_item.model.room_Z       : string  = "room_0_with_tables.model"
internal_item.placement.room_Z   : string  = "+2.5 0 -0.5 (m)"

mapping.daughter_id.room_Y : string = "[room:room_number=0]"
mapping.daughter_id.room_Z : string = "[room:room_number=1]"

[name="floor_1.model" type="geomtools::simple_shaped_model"]
shape_type        : string = "box"
x                 : real   = 10.0
y                 : real   = 4.0
z                 : real   = 3.0
length_unit       : string = "m"
material.ref      : string = "concrete"

visibility.hidden : boolean = 0
visibility.color  : string  = "blue"

internal_item.labels : string[4] = "room_A" "room_B" "room_C" "room_D"

internal_item.model.room_A       : string  = "room_1.model"
internal_item.placement.room_A   : string  = "-3.75 0 -0.25 (m)"

internal_item.model.room_B       : string  = "room_1_with_bed.model"
internal_item.placement.room_B   : string  = "-1.25 0 -0.25 (m)"

internal_item.model.room_C       : string  = "room_1.model"
internal_item.placement.room_C   : string  = "+1.25 0 -0.25 (m)"
```

```
internal_item.model.room_D       : string  = "room_1_with_bed.model"
internal_item.placement.room_D   : string  = "+3.75 0 -0.25 (m) / z 180 (degree)"

mapping.daughter_id.room_A : string = "[room:room_number=0]"
mapping.daughter_id.room_B : string = "[room:room_number=1]"
mapping.daughter_id.room_C : string = "[room:room_number=2]"
mapping.daughter_id.room_D : string = "[room:room_number=3]"

[name="house.model" type="geomtools::stacked_model"]
length_unit       : string = "m"
material.ref      : string = "concrete"

visibility.hidden : boolean = 0
visibility.color  : string  = "green"

stacked.axis            : string  = "z"
stacked.number_of_items : integer = 2
stacked.label_1         : string  = "first_floor"
stacked.model_1         : string  = "floor_1.model"
stacked.label_0         : string  = "ground_floor"
stacked.model_0         : string  = "floor_0.model"

mapping.daughter_id.ground_floor : string = "[floor:floor_number=0]"
mapping.daughter_id.first_floor  : string = "[floor:floor_number=1]"

[name="world" type="geomtools::simple_world_model"]
material.ref    : string = "vacuum"
setup.model     : string = "house.model"

angle_unit      : string = "degree"
setup.phi       : real   = 0.0
setup.theta     : real   = 0.0

length_unit     : string = "m"
setup.x         : real   = 0.0
setup.y         : real   = 0.0
setup.z         : real   = 0.0
world.x         : real   = 14.0
world.y         : real   = 10.0
world.z         : real   = 14.0

visibility.hidden           : boolean = 0
visibility.daughters.hidden : boolean = 0

mapping.daughter_id.setup : string = "[house:house_number=666]"

# end of domestic_models.geom
```