Plug-out – Tutorial

This tutorial is written to help you to write a *plug-out*, only few features available for the *plug-out* are exposed here, the full description of the API this the modeler is given here. A chapter of the reference manual is dedicated to the *plug-outs*.

Goal

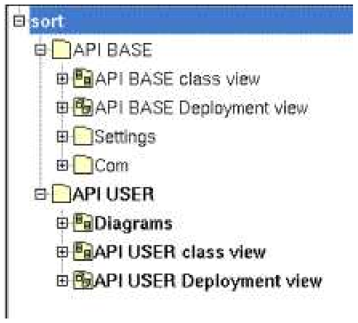
I describe here how to write a *plug-out* both in C++ and Java sorting the elements in the browser to have in order :

- in a *package* : *sub-packages*, *use case views, class views, component views, deployment views, generalizations, dependencies*
- in a *use case view* : *sub-use case views, use case diagrams, sequence diagrams, collaboration diagrams, use cases, actors*
- in a *class view : class diagrams, sequence diagrams, collaboration diagrams, state machines, classes*
- in a *component view : components diagrams, components*
- in a *deployment view : deployment diagrams, nodes, artifacts*
- in a *use case* : *use case diagrams, sequence diagrams, collaboration diagrams, sub-use cases, actors, generalizations, dependencies*
- in a *class (actor)* or a *state machine* : nothing is re-ordered, you have to do it yourself as an exercise :-), don't forget that the order of the members in a class is followed by the code generator.

In a same family the alphabetic order is followed.

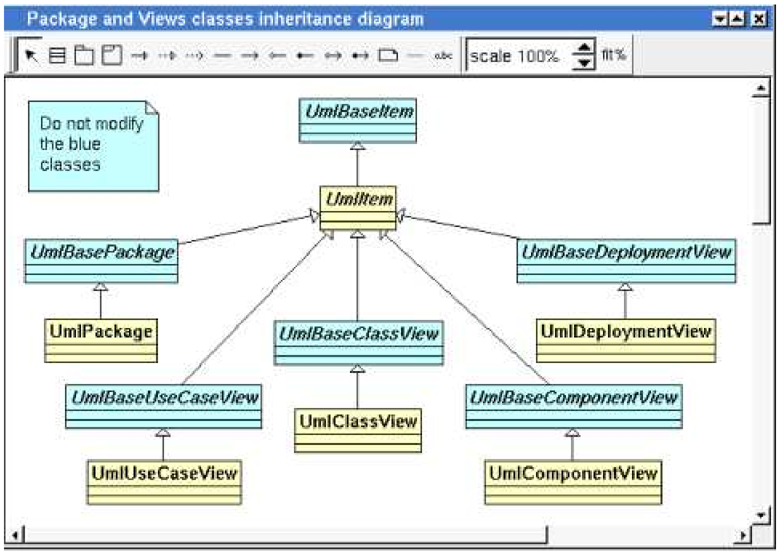
Create a *plug-out*

This *plug-out* is written from scratch, not from an other one. Start BOUML and load the *plug-out* named *empty* part of the distribution. BOUML ask you immediately to save it, protecting the definition of *empty*, of course I name it *sort* :



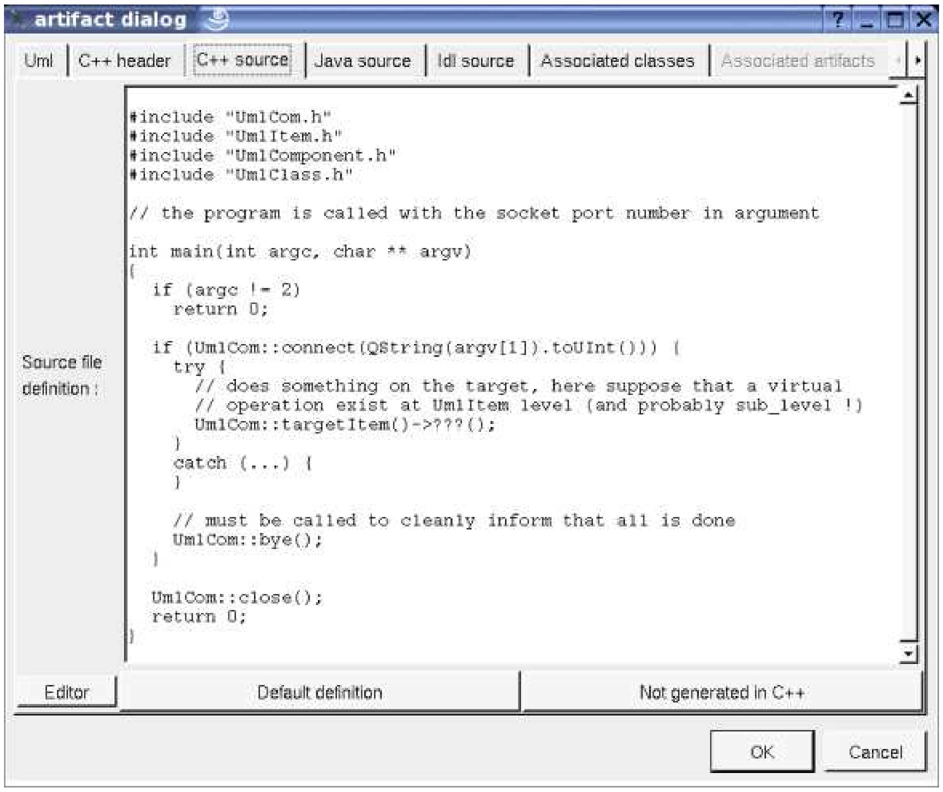
As you can see the project contains two main *packages*, the first one defines *system base classes* and associated *artifacts* supporting the API with the modeler, these classes are read-only. The second *package* contains predefined *user classes* and associated *artifacts*, generally a user class inherits a *system* class.

The classes mainly correspond to the browser elements and the settings, for instance a *package* is managed by the user class *UmlPackage* inheriting the *system* class *UmlBasePackage* etc ..., for instance :

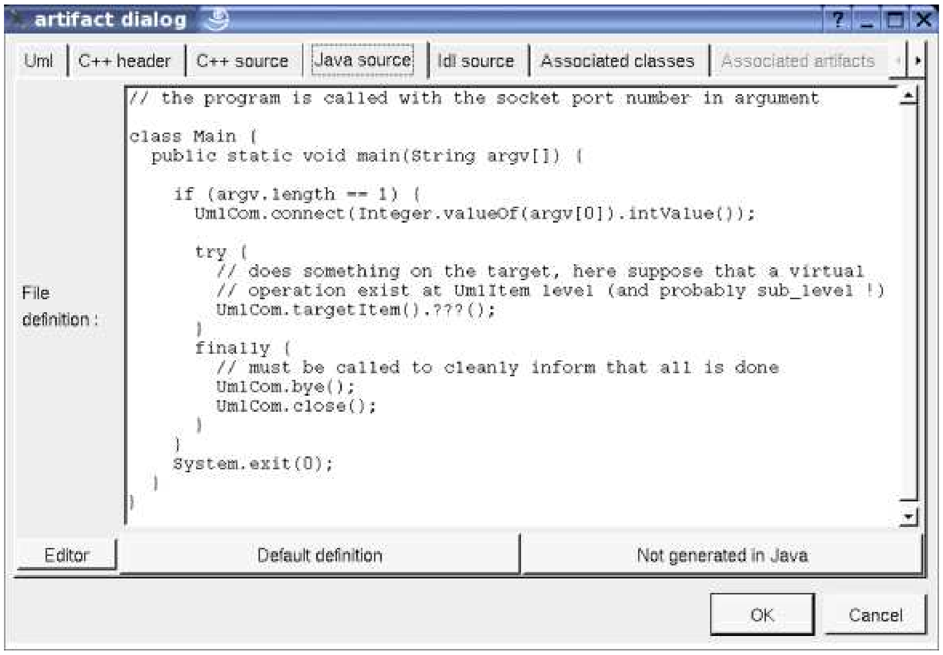


Execution beginning

As usual the execution starts in the *main*, and a *plug-out* is always applied on a browser element. A default definition is given in the *artifact main* defined in *API USER Deployment view* :



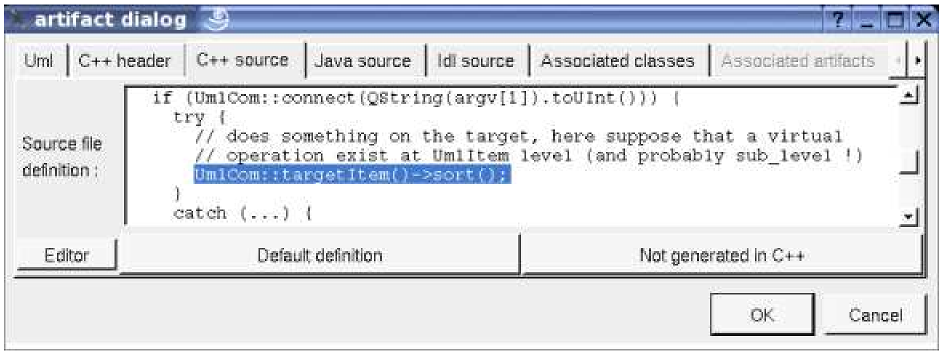
```
#include "UmlCom.h"
#include "UmlItem.h"
#include "UmlComponent.h"
#include "UmlClass.h"

// the program is called with the socket port number in argument

int main(int argc, char ** argv)
{
  if (argc != 2)
    return 0;

  if (UmlCom::connect(QString(argv[1]).toUInt())) {
    try {
      // does something on the target, here suppose that a virtual
      // operation exist at UmlItem level (and probably sub_level !)
      UmlCom::targetItem()->???();
    }
    catch (...) {
    }

    // must be called to cleanly inform that all is done
    UmlCom::bye();
  }

  UmlCom::close();
  return 0;
}
```



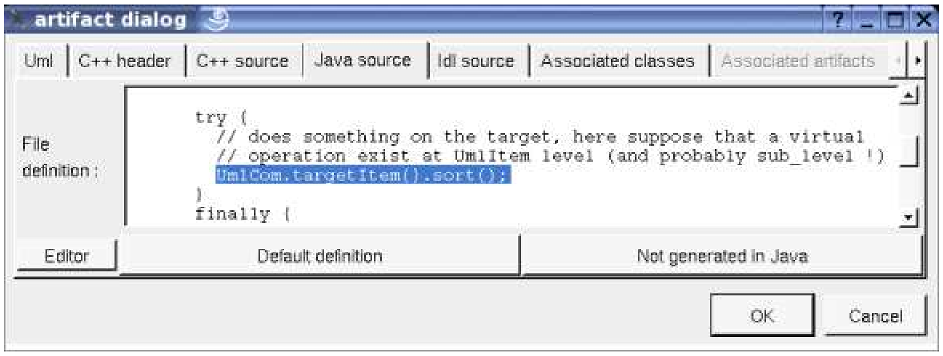
```
// the program is called with the socket port number in argument

class Main {
  public static void main(String argv[]) {

    if (argv.length == 1) {
      UmlCom.connect(Integer.valueOf(argv[0]).intValue());

      try {
        // does something on the target, here suppose that a virtual
        // operation exist at UmlItem level (and probably sub_level !)
        UmlCom.targetItem().???();
      }
      finally {
        // must be called to cleanly inform that all is done
        UmlCom.bye();
        UmlCom.close();
      }
    }
    System.exit(0);
  }
}
```

Our goal is to sort all the children of the browser element on which the *plug-out* is applied, this element is get using the operation *targetItem* defined on *UmlCom* and returning an *UmlItem* which is the base type of all the browser elements, this operation is already called in the default definitions of the *main*.

The *plug-out* only have interest when it is applied on a *package*, any *view* or on a *use case*, and it will be configured in BOUML for these kinds of elements, but because he configuration may be wrong I have to protect the other cases, so I decide to define the operation *sort* on *UmlItem* indicating the error, and to refine *sort* on *UmlPackage, UmlUseCaseView, UmlClassView, UmlComponentView, UmlDeploymentView* and *UmlUseCase.* The *main* just have to apply *sort* on the result of *targetItem* :



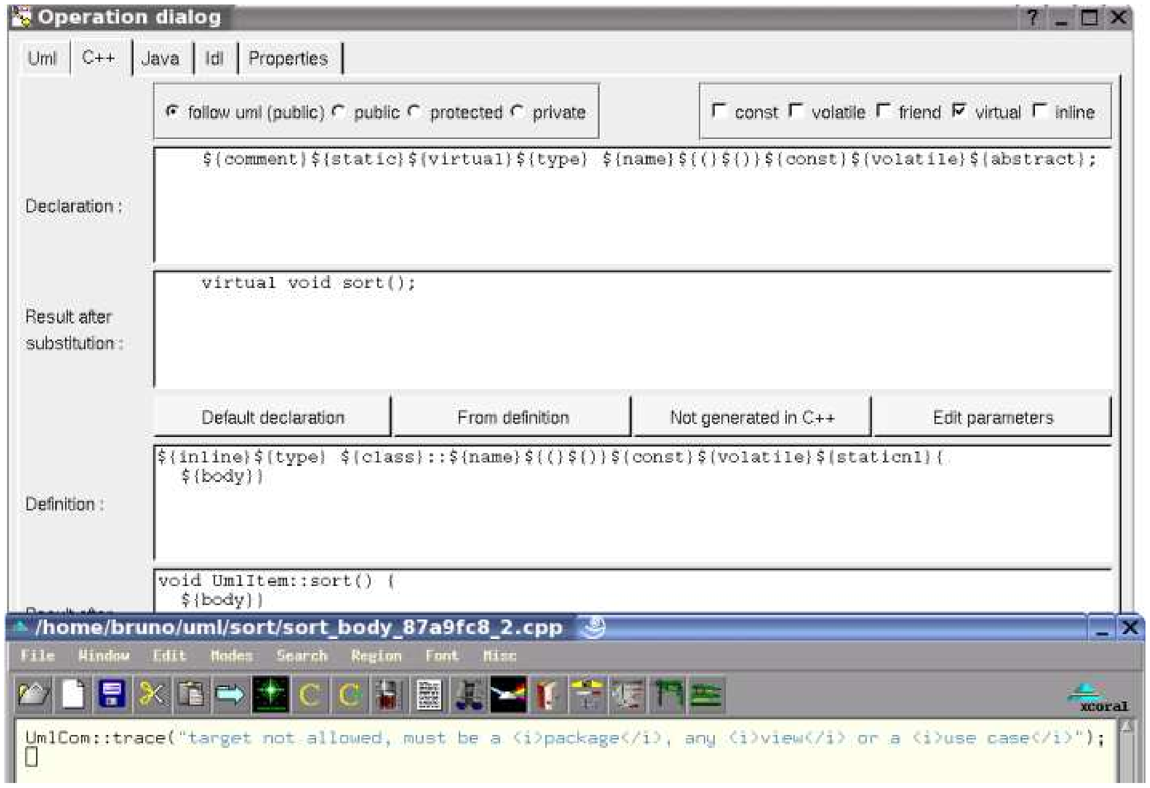
```
  if (UmlCom::connect(QString(argv[1]).toUInt())) {
    try {
      // does something on the target, here suppose that a virtual
      // operation exist at UmlItem level (and probably sub_level !)
      UmlCom::targetItem()->sort();
    }
    catch (...) {
```



```
      try {
        // does something on the target, here suppose that a virtual
        // operation exist at UmlItem level (and probably sub_level !)
        UmlCom.targetItem().sort();
      }
      finally {
```

Error on wrong element

Now I define the operation *sort* on *UmlItem*, to indicate the error to the user I use the operation *message* defined on *UmlCom*, this message will be written in the *trace window* and must be a valid HTML block. *sort* return nothing (*void*) and doesn't have parameters, in C++ it is obviously *virtual* to refine it.



● follow uml (public) ○ public ○ protected ○ private

☐ const ☐ volatile ☐ friend ☑ virtual ☐ inline

Declaration :

```
    ${comment}${static}${virtual}${type} ${name}${(}${)}${const}${volatile}${abstract};
```

Result after substitution :

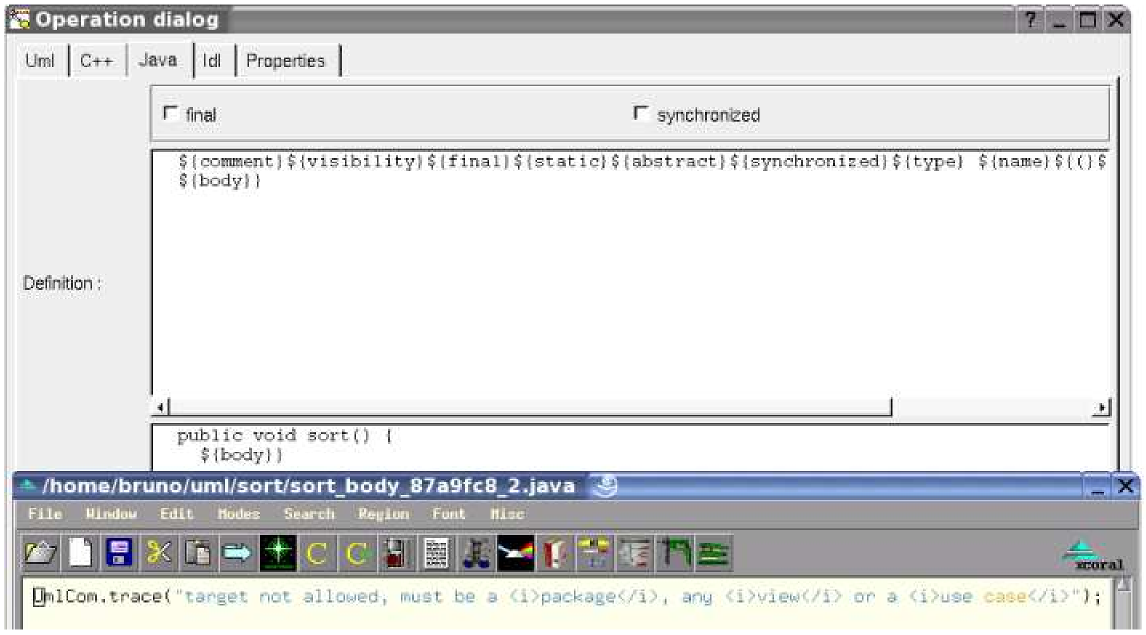
```
    virtual void sort();
```

Definition :

```
${inline}${type} ${class}::${name}${(}${)}${const}${volatile}${staticnl}{
  ${body}}
```

```
void UmlItem::sort() {
  ${body}}
```

```
UmlCom::trace("target not allowed, must be a <i>package</i>, any <i>view</i> or a <i>use case</i>");
```



☐ final ☐ synchronized

Definition :

```
  ${comment}${visibility}${final}${static}${abstract}${synchronized}${type} ${name}${(}$
  ${body}}
```

```
  public void sort() {
    ${body}}
```

```
UmlCom.trace("target not allowed, must be a <i>package</i>, any <i>view</i> or a <i>use case</i>");
```

In C++ the *artifact* must be modified to add *#include "UmlCom.h"* in the source, to find quickly the artifact use the class menu entry *select associated artifact*, to come back use the *artifact* menu entry *select the associated class*.

Sort the children

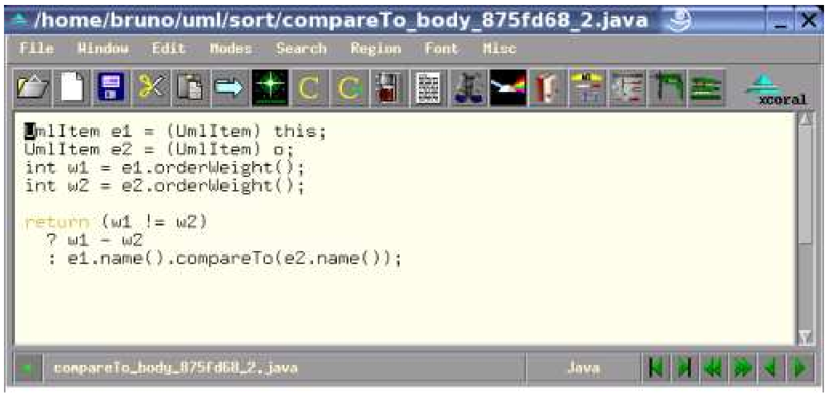
For all the right targets the same thing must be done : to sort the children, I decide this is done by the operation *sortChildren* defined on *UmlItem*, thanks to that I just have to call *sortChildren* in the operation *sort* defined on *UmlPackage, UmlUseCaseView, UmlClassView, UmlComponentView, UmlDeploymentView* and *UmlUseCase*. So I define *sort* on *UmlPackage* just calling *sortChildren* and to quickly have the same definition for the other classes I mark (mouse left click with the *control* key down) *sort* on *UmlPackage* and I use the menu entry *duplicate marked into* on *UmlUseCaseView* etc ...

To sort I have two possibilities : to write the algorithm myself or to use a library, even implementing *quicksort* is trivial I choose the second way because this is more interesting in this tutorial. In Java a *sort* operation is defined on the arrays of elements implementing *Comparable*, in C++ we can use the *sort* operation defined on the *QVector* defining a sub class defining *compareItems*. In the two cases the sort in first done in the *plug-out* memory then the browser elements will be moved accordingly.

Sort in Java

Because we sort instances of several classes I decide to implement *Comparable* by *UmlItem*, because only one class and operation must be defined I don't use *Java catalog* and I do all by hand :

- I create a *package* named *aux* under the project
- in *aux* I create a *class view* named *aux*
- in this *class view* I create the *class Comparable* with the stereotype *interface*, I say it is external in Java to avoid code generation warning, and even this is not mandatory I define the *abstract* operation *compareTo*.
- I create a diagram to allow *UmlItem* to realize *Comparable*, I say the realization in not implemented in C++
- I define *compareTo* on *UmlItem*, for instance using *add inherited operation*, I say the operation is not implemented in C++
- the comparison must first take into account the class of an element then may be its name, for that I define the operation *orderWeight* returning an int valuing 0 for *UmllItem* (the value is not relevant, this operation will not be called)
- the name of a browser element is returned by the operation *name*, so the definition of *int compareTo(Object o)* is :

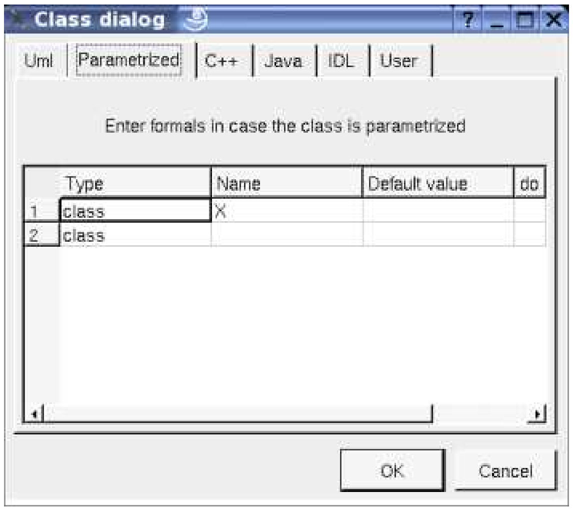

/home/bruno/uml/sort/compareTo_body_875fd68_2.java
File Window Edit Modes Search Region Font Misc
xcoral

```
UmlItem e1 = (UmlItem) this;
UmlItem e2 = (UmlItem) o;
int w1 = e1.orderWeight();
int w2 = e2.orderWeight();

return (w1 != w2)
  ? w1 - w2
  : e1.name().compareTo(e2.name());
```

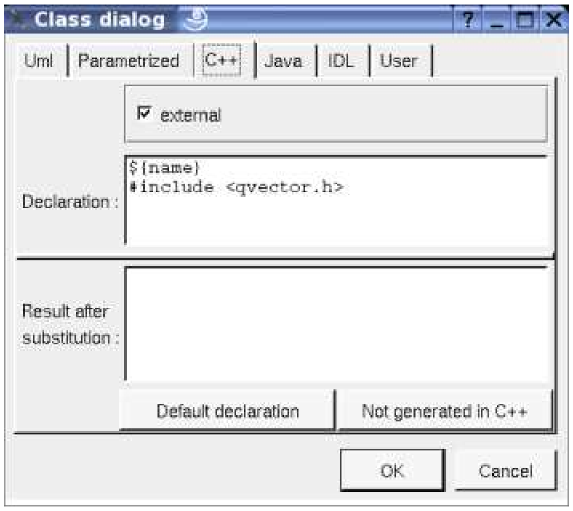
compareTo_body_875fd68_2.java
Java

Sort in C++/Qt

- I say *orderWeight* is defined *virtual* in C++ on *UmlItem* and it returns also 0 in C++
- in the *class view aux* I create the parametrized class *QVector*, I say it is not implemented in Java, in C++ this is a *external* class having this definition :



	Type	Name	Default value	do
1	class	X		
2	class			

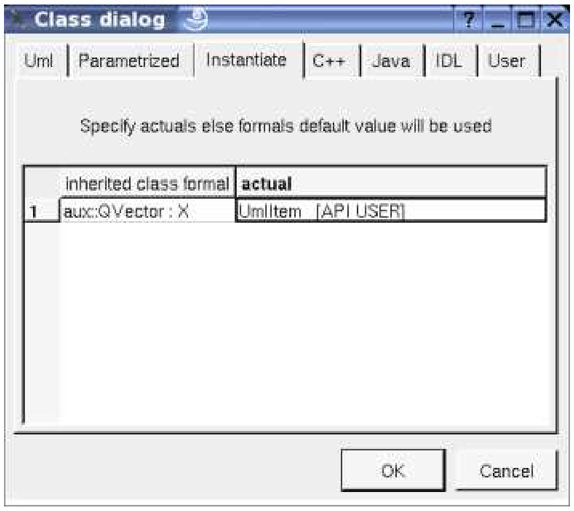


☑ external

Declaration :

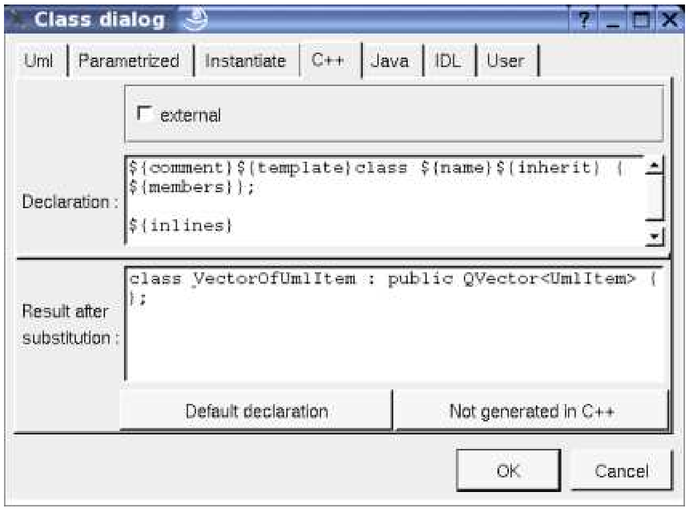
```
${name}
#include <qvector.h>
```

- even it is not mandatory on it I define the operation *int compareItems(QCollection::Item d1, Qcollection::Item d2)*
- I define the class *VectorOfUmlItem* inheriting *QVector,* I say this class must not be defined in Java, in C++ its definition is (the inheritance must be defined before the edition of the class to set the *actual*) :



Specify actuals else formals default value will be used

	inherited class formal	actual
1	aux::QVector : X	UmlItem [API USER]



☐ external

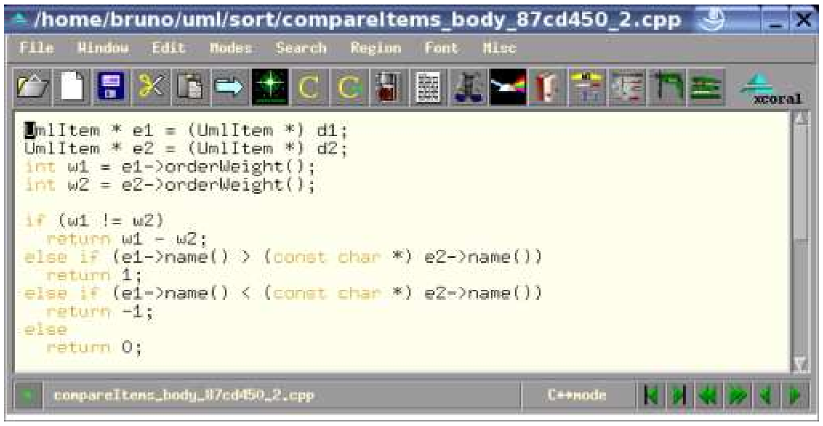
Declaration :

```
${comment}${template}class ${name}${inherit} {
${members}};

${inlines}
```

Result after substitution :

```
class VectorOfUmlItem : public QVector<UmlItem> {
};
```

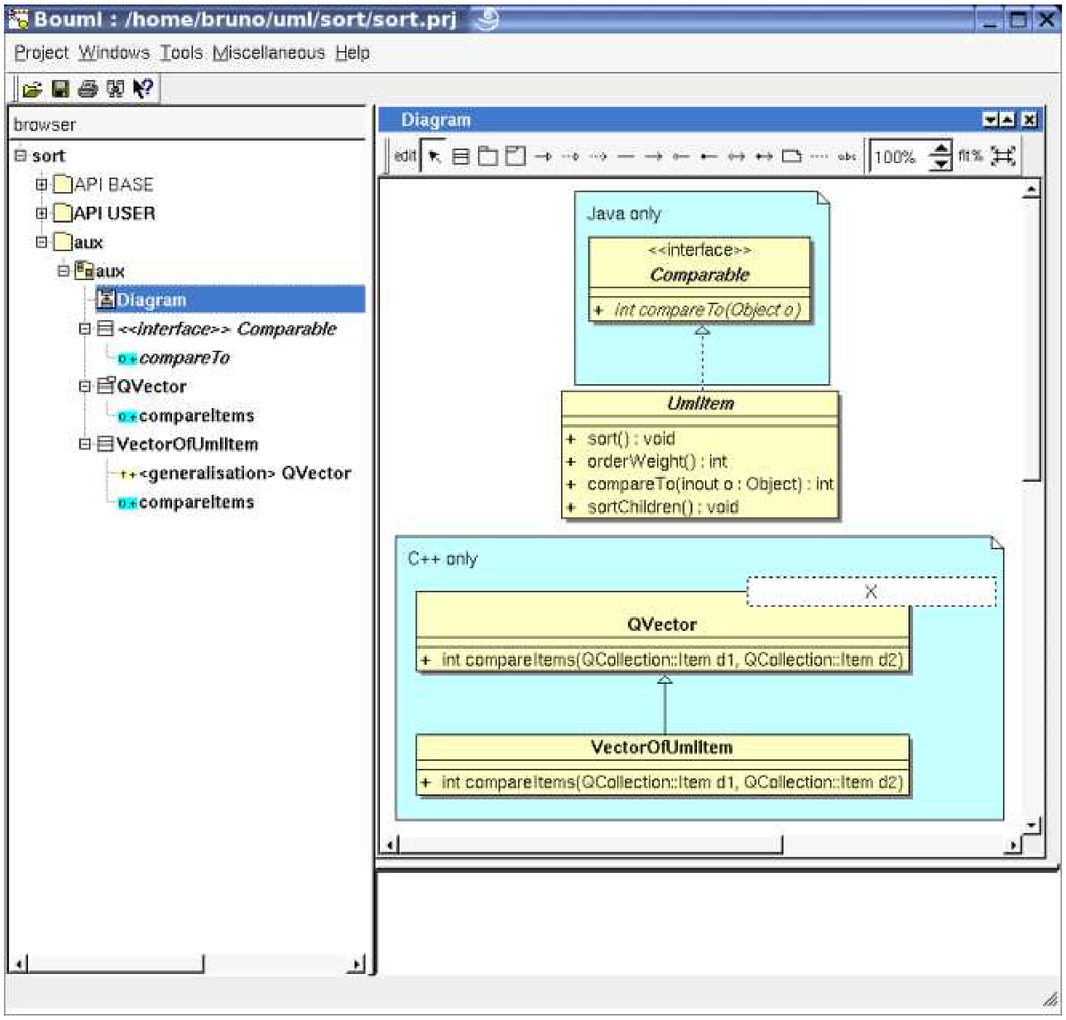
- on *VectorOfUmlItem* I define *compareItems* like this :


/home/bruno/uml/sort/compareItems_body_87cd450_2.cpp
File
Window
Edit
Modes
Search
Region
Font
Misc
xcoral
compareItems_body_87cd450_2.cpp
C++mode

```
UmlItem * e1 = (UmlItem *) d1;
UmlItem * e2 = (UmlItem *) d2;
int w1 = e1->orderWeight();
int w2 = e2->orderWeight();

if (w1 != w2)
  return w1 - w2;
else if (e1->name() > (const char *) e2->name())
  return 1;
else if (e1->name() < (const char *) e2->name())
  return -1;
else
  return 0;
```

- To not have to create an *artifact* for it I edit the *artifact UmlItem* and I add *VectorOfUmlItem* in the associated class list

Finally :



Operation orderWeight

Now we have to define *orderWeight*, to have the desired order the returned value may be :

- *UmlPackage (package)* : 1
- *UmlUseCaseView (use case view)* : 2
- *UmlClassView (class view)* : 3
- *UmlComponentView (component view*): 4
- *UmlDeploymentView (deployment view*): 5
- *UmlUseCaseDiagram (use case diagram)*, *UmlClassDiagram* (*class diagram), UmlComponentDiagram (components diagram), UmlDeploymentDiagram (deployment diagram)* : 6
- *UmlSequenceDiagram (sequence diagram)* : 7
- *UmlCollaborationDiagram (collaboration diagram)* : 8
- *UmlUseCase (use case), UmlState (state machine), UmlComponent (component), UmlNode (node)* : 9
- *UmlClass (class), UmlArtifact (artifact)* : 10
- *UmlRelation* and *UmlNcRelation* of kind *generalization* (*relationKind()* valuing *aGeneralisation*) : 11
- *UmlRelation* and *UmlNcRelation* of kind *dependency* (*relationKind()* valuing *aDependency*) : 12

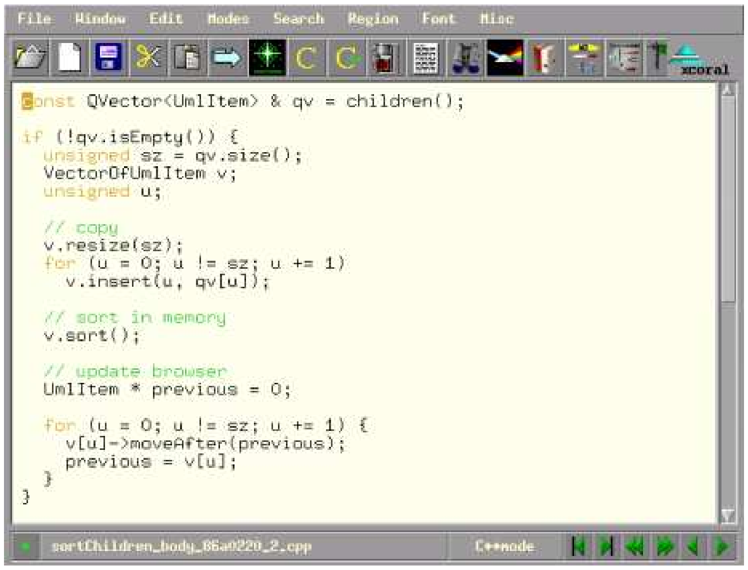
I define the operations on the right class with the right returned value, of course using the duplication on the operation.

sortChildren

The operation *children()* defined on *UmlBaseItem* returns a vector of *UmlItems.*

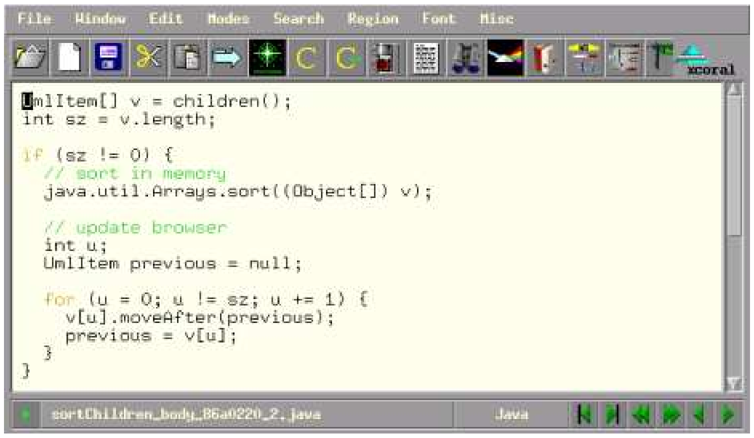
To change the order in the browser the operation *moveAfter* is defined on UmlBaseItem. It is only possible to change the order, not to move an element from its parent into an other one. If the parameter is null the element is moved to be the first child of its parent, else it is moved to be placed after the parameter (of course the parameter and the element must have the same parent else nothing is done).

The definition of *sortChildren* in C++ is :


File Window Edit Modes Search Region Font Misc
xcoral
sortChildren_body_86a0220_2.cpp
C++mode

```
const QVector<UmlItem> & qv = children();

if (!qv.isEmpty()) {
  unsigned sz = qv.size();
  VectorOfUmlItem v;
  unsigned u;

  // copy
  v.resize(sz);
  for (u = 0; u != sz; u += 1)
    v.insert(u, qv[u]);

  // sort in memory
  v.sort();

  // update browser
  UmlItem * previous = 0;

  for (u = 0; u != sz; u += 1) {
    v[u]->moveAfter(previous);
    previous = v[u];
  }
}
```

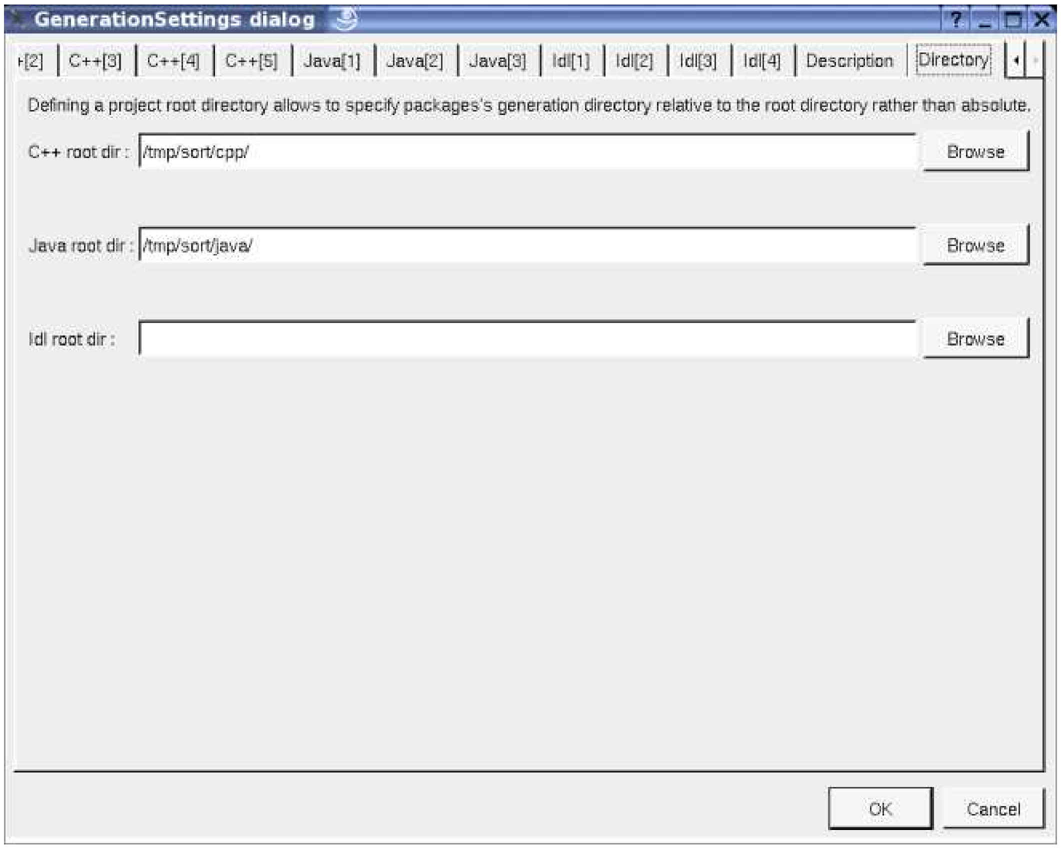
The definition of *sortChildren* in Java is :


File Window Edit Modes Search Region Font Misc
xcoral
sortChildren_body_86a0220_2.java
Java

```
UmlItem[] v = children();
int sz = v.length;

if (sz != 0) {
  // sort in memory
  java.util.Arrays.sort((Object[]) v);

  // update browser
  int u;
  UmlItem previous = null;

  for (u = 0; u != sz; u += 1) {
    v[u].moveAfter(previous);
    previous = v[u];
  }
}
```

Generate code

To generate the code we have to set where the sources must be placed, edit the *generation settings* for instance to have :


GenerationSettings dialog
+[2] C++[3] C++[4] C++[5] Java[1] Java[2] Java[3] Idl[1] Idl[2] Idl[3] Idl[4] Description Directory
Browse
OK
Cancel

Defining a project root directory allows to specify packages's generation directory relative to the root directory rather than absolute.

C++ root dir : /tmp/sort/cpp/

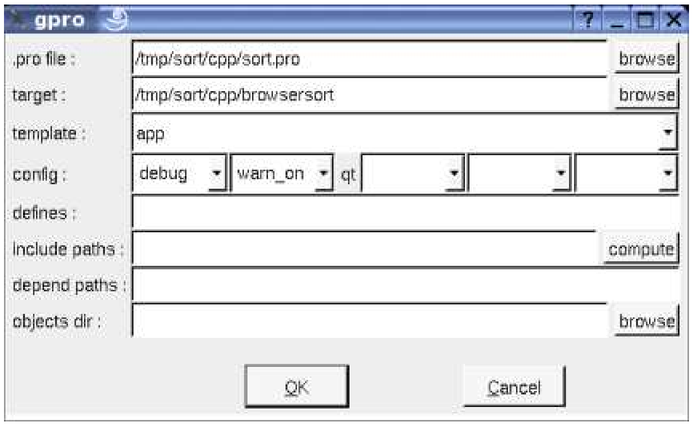
Java root dir : /tmp/sort/java/

Idl root dir :

Ask for the C++ and/or Java generation using the menu *Tools* or through the menu of the *project* (all must be generated)

C++ compilation

To compile the C++ definition under Linux or MacOS X we need a Makefile, for that the better is to produce the *.pro* file applying the *plug-out genpro* on the *artifact* named *executable*, because *sort* doesn't use data specific to C++ Java or Idl we can remove the *defines* (the executable will be smaller), the name of the executable is *browsersort* :



.pro file : /tmp/sort/cpp/sort.pro

target : /tmp/sort/cpp/browsersort

template : app

config : debug warn_on qt

defines :

include paths :

depend paths :

objects dir :

Use *qmake* to produce the Makefile, then *make* to compile.

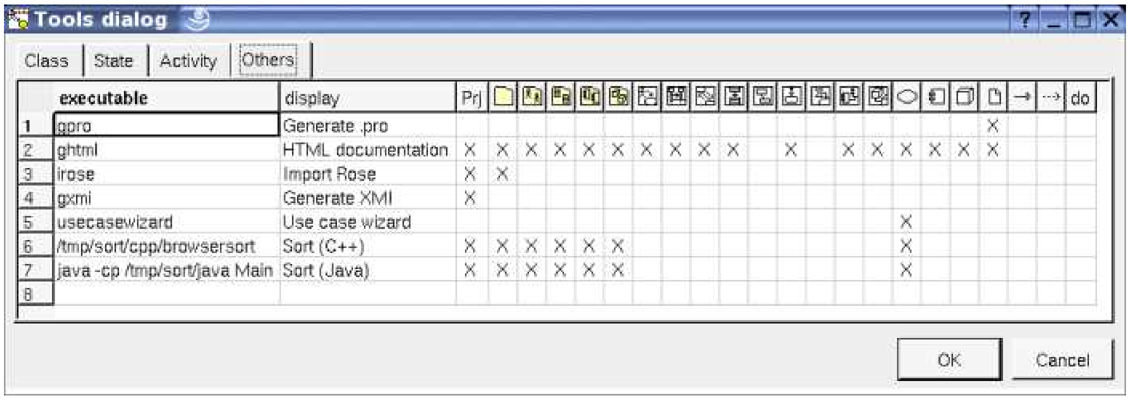
Java compilation

To compile the java files, go in the directory where the files are and do *javac *.java* (don't compile file by file because some files associated to the API contain several classes).

All is done, don't forget to save your project :-)

Use the plug-out

To use the *plug-out* in any project you must declare it, call the *Tools settings* dialog in the menu *Tools*, here I declare the C++ and Java implementation :



	executable	display	Prj																⬭				→	⇢	do
1	gpro	Generate .pro																				X			
2	ghtml	HTML documentation	X	X	X	X	X	X	X	X	X	X		X		X	X	X	X	X	X			
3	irose	Import Rose	X	X																				
4	gxmi	Generate XMI	X																					
5	usecasewizard	Use case wizard																X						
6	/tmp/sort/cpp/browsersort	Sort (C++)	X	X	X	X	X	X										X						
7	java -cp /tmp/sort/java Main	Sort (Java)	X	X	X	X	X	X										X						
8																								

Now the sub-menu *tool* on the *project*, a *package,* all the *views* and a *use case* propose to sort.

Easy isn't it ? Do not hesitate to write your *plug-outs*, look at the already defined ones to have examples !

Happy modeling !

Bruno Pagès