

Les classes et la génération de code sous Bouml

par Bruno Pagès ([Bouml](#))

Date de publication : 20/10/2007

Dernière mise à jour :



*Ce tutoriel écrit en 2008 décrit à une très vieille version de BOUML et n'a pas été mis à jour. Il est préférable d'utiliser les tutoriels vidéo disponibles sur la page **documentation***

Ce tutoriel est le second sur BOUML, un modelleur UML2 gratuit fonctionnant sous *Windows, Linux, Solaris, MacOS X* disponible [ici](#). Ce tutoriel porte sur la définition de classes et la génération de code avec BOUML.

I - Rappel.....	3
II - Définir des classes.....	4
II-A - Sous package, class view, diagrammes de classe, classes.....	4
II-B - Ajouter des attributs et opérations.....	6
II-C - Éditer un attribut.....	7
II-D - Modifier le dessin d'une classe.....	13
II-E - Relations.....	13
III - Classes spéciales.....	26
III-A - Définir des structures.....	26
III-B - Définir des unions.....	26
III-C - Définir des typedefs.....	26
III-D - Définir des énumérations.....	27
III-E - Définir des interfaces.....	30
III-F - Définir des exceptions.....	30
III-G - Templates et Génériques.....	31
IV - Génération de code, deployment view, artifact.....	32
V - Conclusion.....	35
V-A - Epilogue.....	35
V-B - Liens.....	35
V-C - Remerciements.....	35

I - Rappel

 *Le tutoriel Premiers pas avec BOUML doit être lu avant ce tutoriel.*

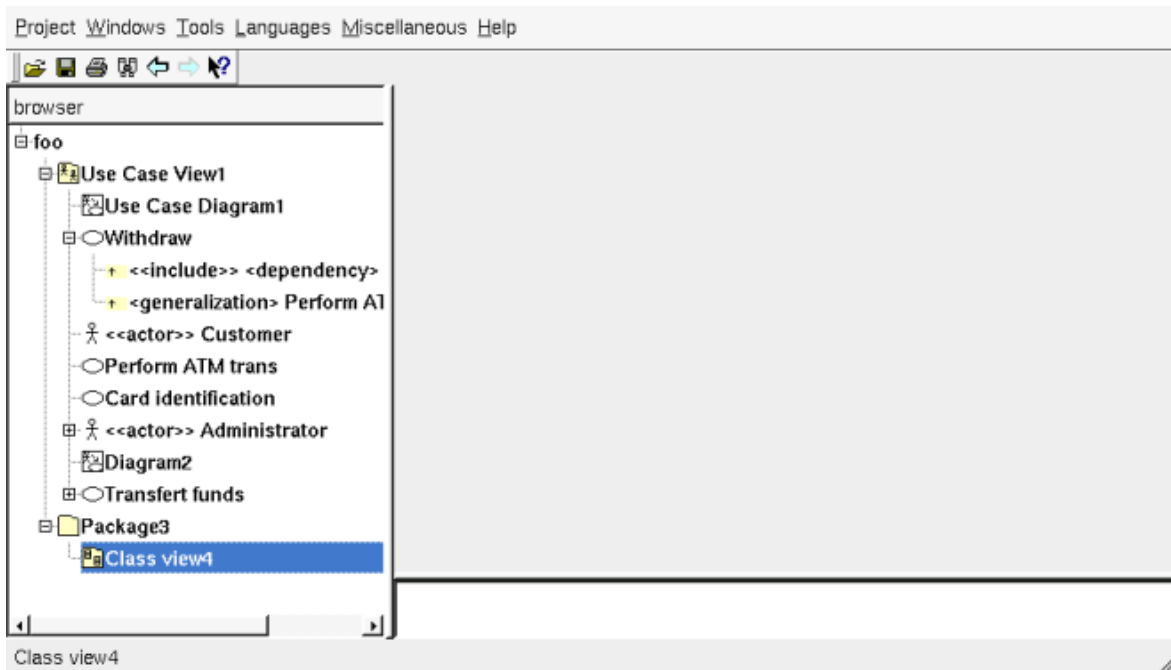
En suivant le premier tutoriel, nous avons créé un projet, des **use cases** et des acteurs, le tout étant présenté dans des **use case diagrams**.

Rechargez ce même projet, pour pouvoir le compléter.

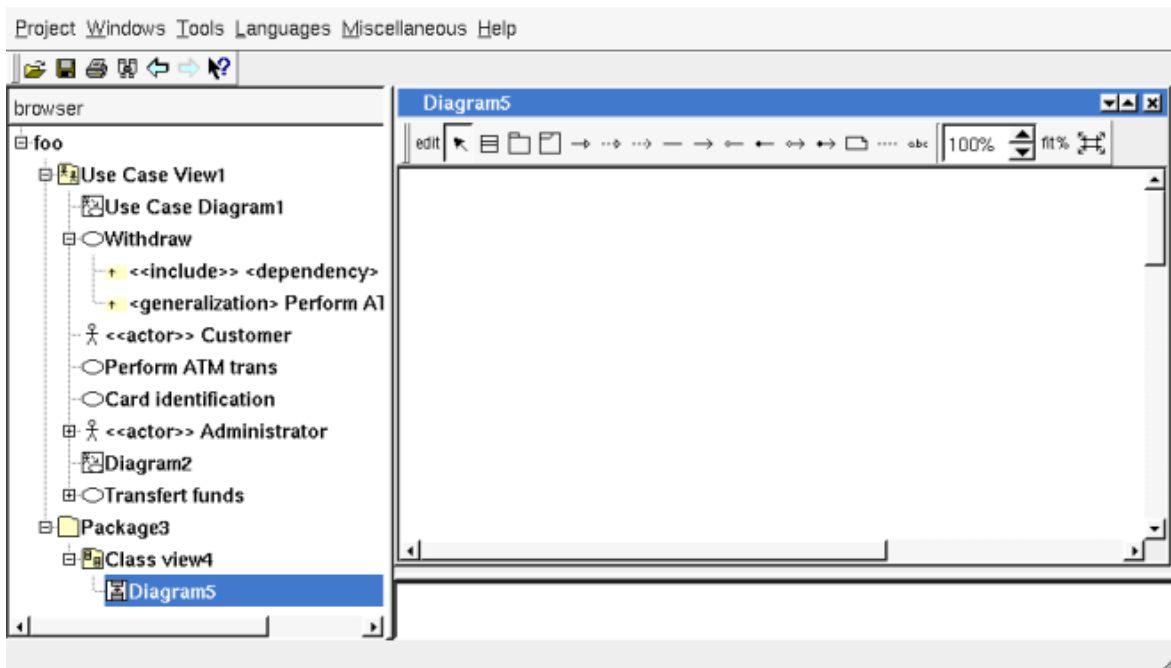
II - Définir des classes

II-A - Sous package, class view, diagrammes de classe, classes

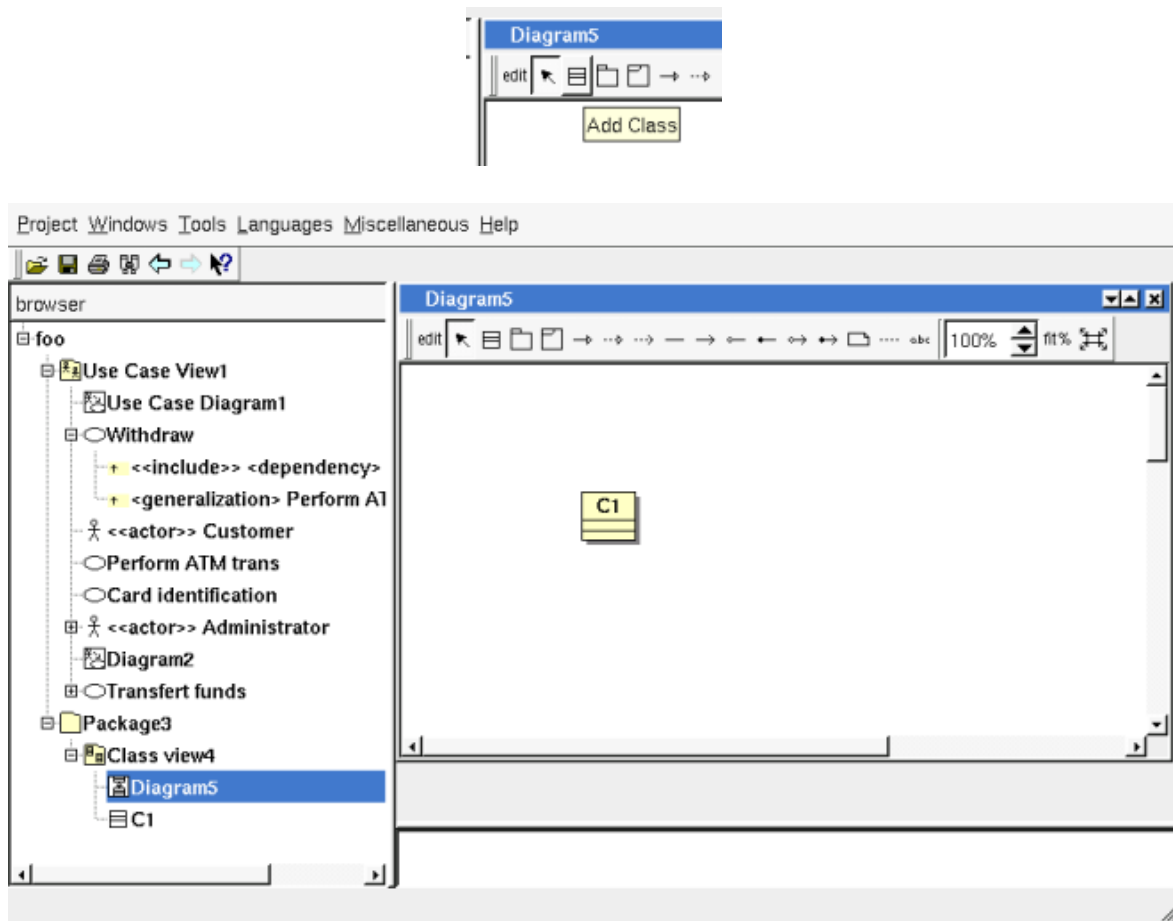
Nous voulons définir des classes dans le but de générer du code, pour cela nous devons premièrement créer une *class view*, et pour changer nous n'allons pas la placer directement dans le *package* projet. Pour cela affichons le menu du projet (clic droit sur *foo* dans le *browser*) choisissons *new package*, et nommons le *package Package3*. Appelez le menu de *Package3* et choisissez *new class view*, et nommons la *Class view4* :



Créons un diagramme de classes *via* le menu de *Class view4* et ouvrons ce nouveau diagramme :



Appuyez sur le bouton représentant une classe puis cliquez dans le diagramme (il est aussi possible de créer la classe *via* le menu de la *class view* dans le *browser*), appelez cette classe C1:



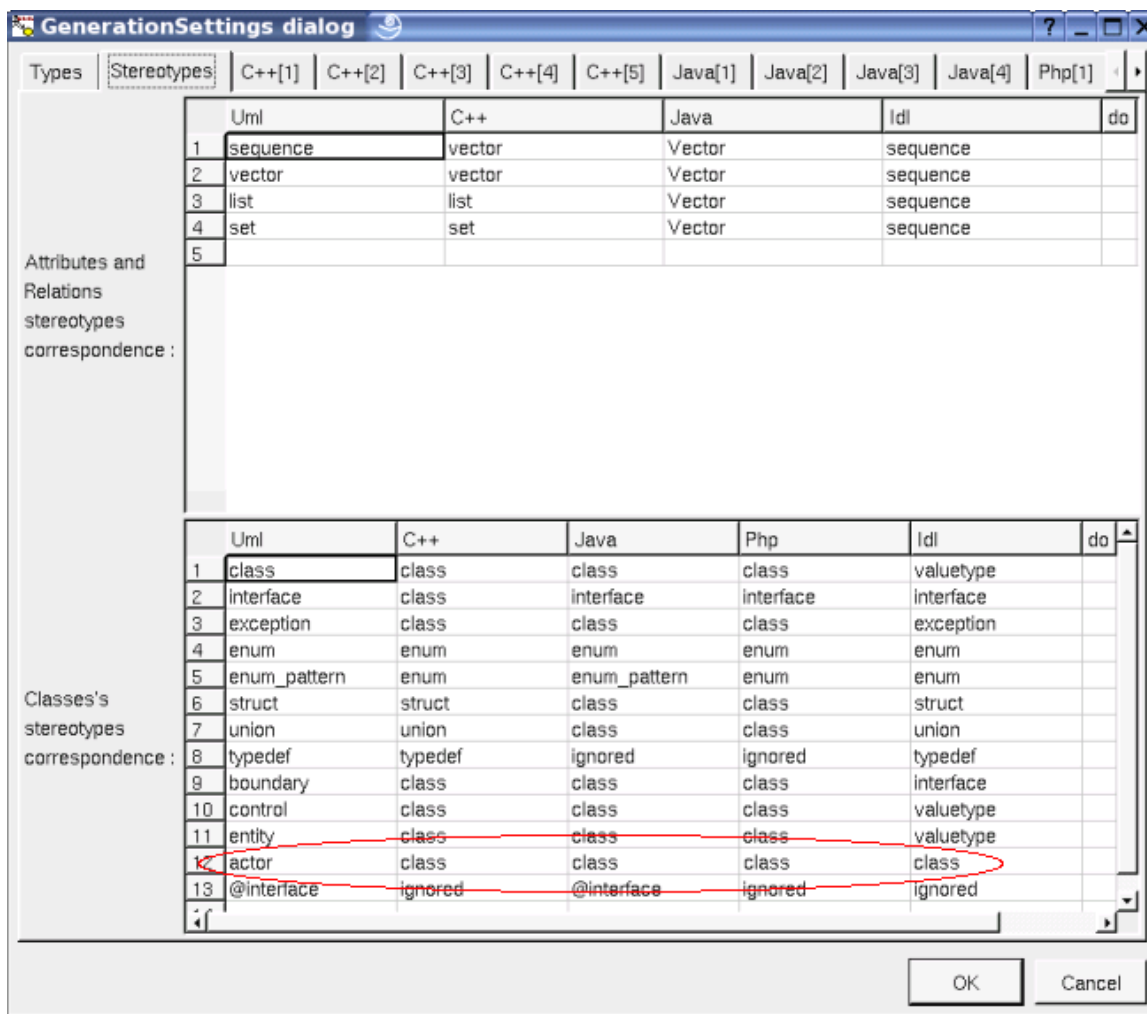
Parce les acteurs sont des classes, ceux-ci peuvent être ajoutés dans un diagramme de classe, *via* un *drag and drop* du *browser* dans le diagramme, ou en utilisant le bouton représentant une classe et en choisissant l'une d'elle dans la liste proposée, par exemple *Administrator*. Vous pouvez choisir un élément dans une liste défilante (*combo box*) soit en la parcourant *via* les flèches soit en tapant le premier caractère de son nom.

! Malheureusement la sélection d'un élément *via* son nom en tapant un caractère dans une liste défilante (*combo box*) ne différencie pas les majuscules et minuscules, je n'ai rien pu faire contre car cela est profondément enfoui dans Qt et dépend en plus de la version de Qt.

L'acteur est dessiné comme un acteur et non une classe standard, c'est ce qui se passe par défaut pour les classes ayant le stéréotype *actor*. Ceci n'est pas une obligation et peut être modifié *via* les *drawing settings* de la classe dans le diagramme, des dessins particuliers sont également utilisés pour les classes stéréotypées *control*, *boundary* et *entity*. On obtient donc :

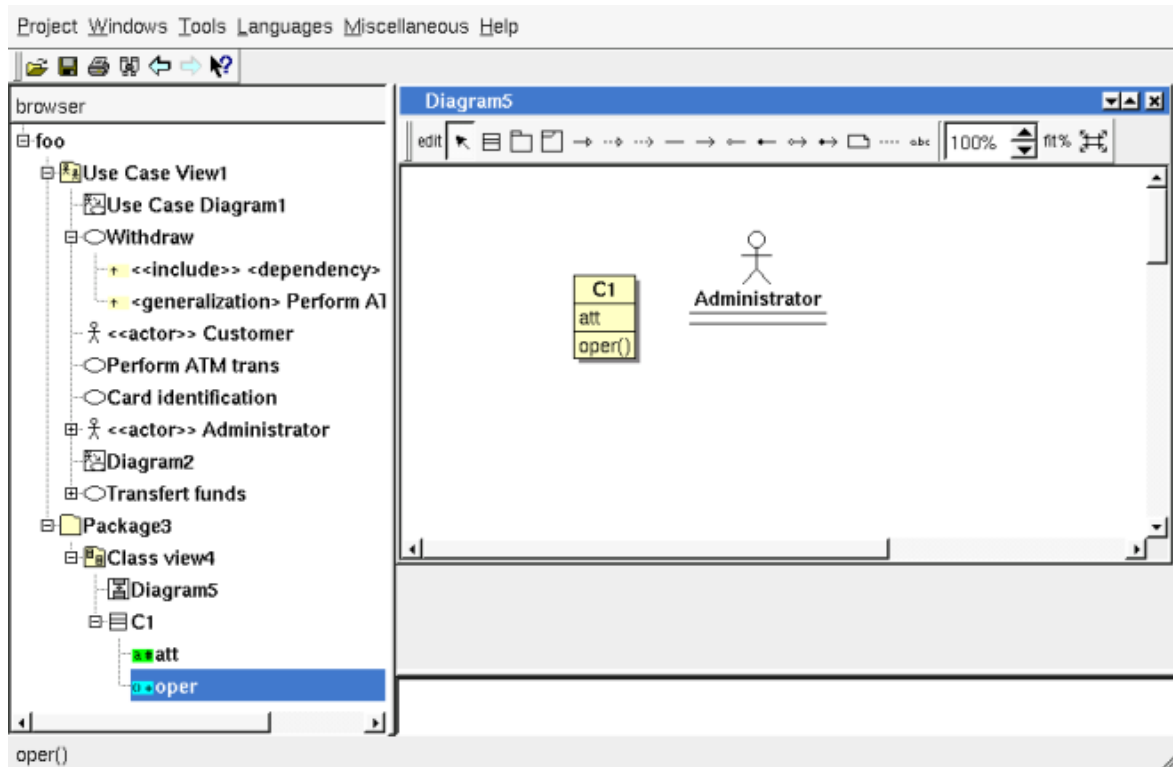


Cependant, dans le mode par défaut les acteurs ne sont pas utilisés pour la génération de code, en effet par défaut le stéréotype de classe *actor* produit *ignored* dans les différents langages cibles. Pour continuer sans cette limitation, editez les *generation settings* allez dans l'intercalaire *Stereotypes* et remplacer *ignored* par *class* pour *actor* :

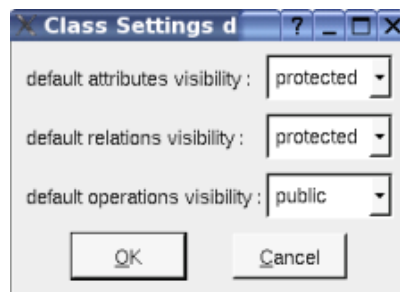


II-B - Ajouter des attributs et opérations

Nous voulons définir l'attribut *att*, et l'opération *oper* pour la classe *C1*. Appelez le menu de la classe, choisissez *add attribute* puis *add operation*. Lorsque vous ajoutez un membre de classe *via* le diagramme l'élément est automatiquement édité, ce n'est pas le cas *via* le *browser*. Les nouveaux éléments sont visibles dans le *browser* et le diagramme :



Comme vous le voyez *att* a la visibilité *protected* et l'opération est publique, car telles sont les visibilités par défaut définies via les *class settings*. Les *class settings* peuvent être positionnés au niveau de chaque *package* et *class view* et s'appliquent au sous-niveau qui ont la valeur *default* en suivant le même principe que celui des *drawing settings* comme cela a été décrit dans le précédent tutoriel.



II-C - Éditer un attribut

Décidons que l'attribut *att* est un *int*, pour cela il faut l'éditer, ce qui peut être fait de différentes façons : double-clic dessus dans le *browser*, ou via son menu dans le *browser*, ou via le menu de *C1* dans le diagramme en choisissant *edit attribute* puis *att*. On obtient :


Le premier intercalaire concerne les propriétés UML, comme vous le voyez, par défaut un attribut est un membre d'instance (pas un membre de classe), il est non volatile et non constant. Les autres intercalaires sont spécifiques à chaque langage de génération : avec BOUML vous pouvez définir des éléments simultanément en C++, Java Php et IDL. Par exemple le générateur de document HTML est un *plug-out* dont l'implémentation est faite en C++ et en Java, vous pouvez donc le modifier pour vos besoins propres dans votre langage préféré.

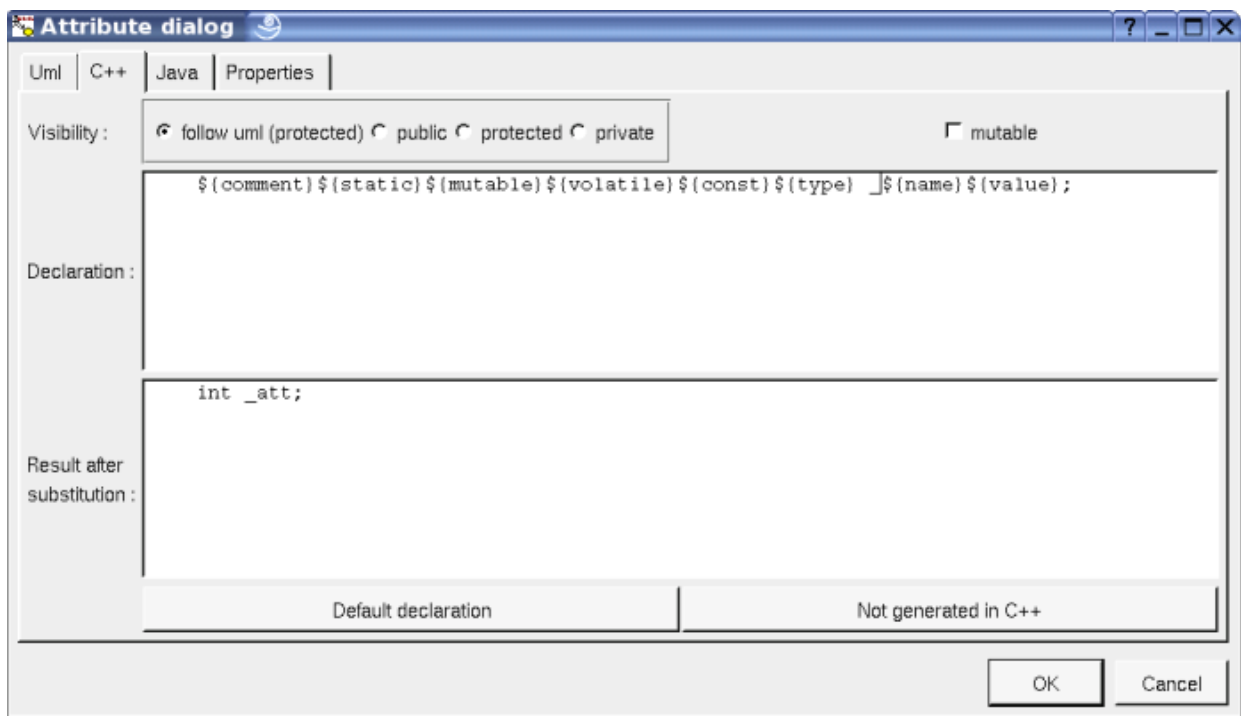
Changez le type en *int*, soit en choisissant parmi les types prédéfinis (modifiables *via* les *generation settings*) ou en tapant *int*.

Allez dans l'intercalaire C++ (en cliquant sur C++) :

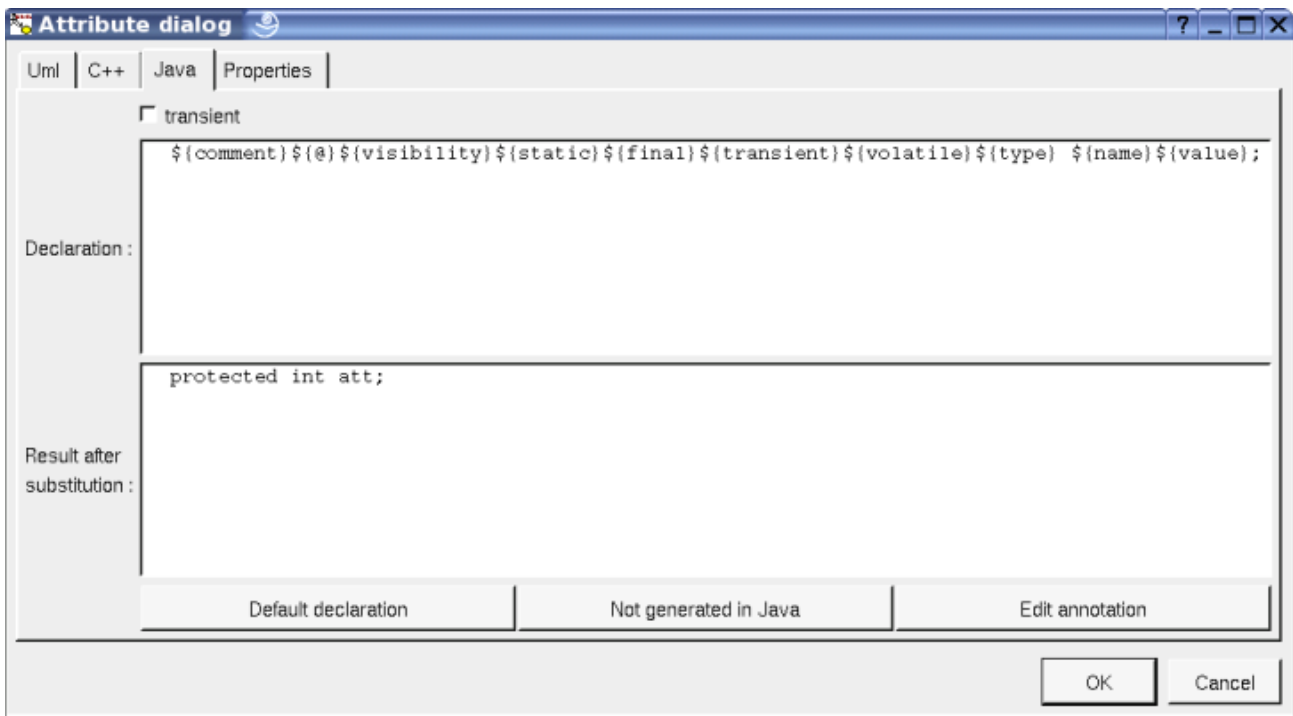
Comme vous le voyez, la visibilité en C++ est celle indiquée au niveau UML, mais ce n'est pas une obligation et BOUML vous permet de ne pas avoir la même visibilité en C++, *Java* et *IDL* si vous le souhaitez. En C++, un attribut peut de plus être *mutable*.

Le texte affiché devant *Declaration* est éditable, pas celui en face de *result after substitution* qui montre ce que sera le code produit par le générateur C++ (sauf si vous modifiez ce dernier !). Exception faite des mots clefs signalés par *{}* et les propriétés utilisateur (**tagged values**) signalées par *@{}*, tous les autres caractères sont générés inchangés, y compris les sauts de ligne. Il n'est pas très difficile de comprendre que *{type}* est remplacé par *int* et *{name}* par *att*, ces valeurs étant définies au niveau UML. Cela veut également dire que si vous remplacez *{type}* par *aze* le type généré pour *att* en C++ sera *aze*, etc

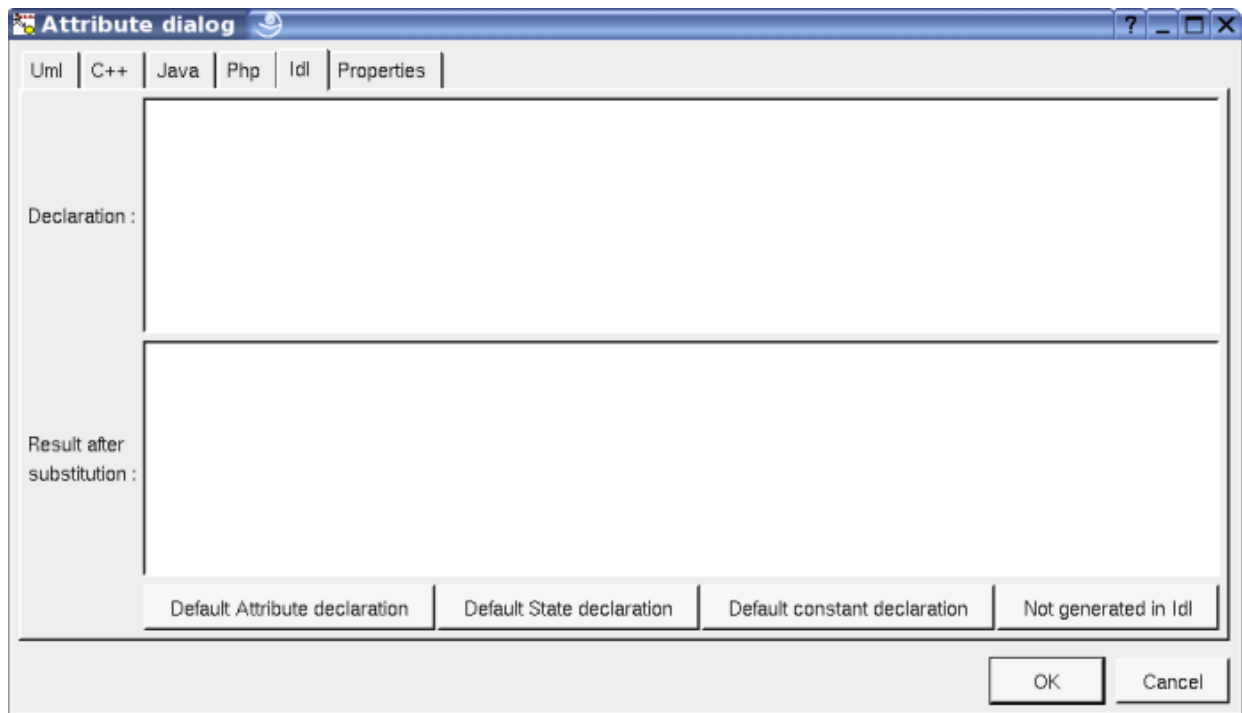
 *Peut être avez-vous des règles de codage demandant qu'en C++ le nom d'un attribut commence toujours par '_' ? Bien sûr, vous pouvez renommer l'attribut `_att`, mais voir le '_' dans le diagramme de classe n'est pas très joli, le mieux est d'ajouter le '_' avant *{name}* :*



Allons dans l'intercalaire *Java*, le principe est le même, appliqué au langage *Java* :



La définition pour Php et Idl n'est pas accessible. Pourquoi ? Rappelez-vous, au début de l'histoire (dans le premier tutoriel) je vous ai demandé de cocher C++ et Java dans le menu *Langages*, mais ni *Php* ni *IDL*. Fermons le dialogue via *ok*, cochons *Php* et *IDL* dans le menu *Languages*, rééditons l'attribut et allons dans l'intercalaire IDL :

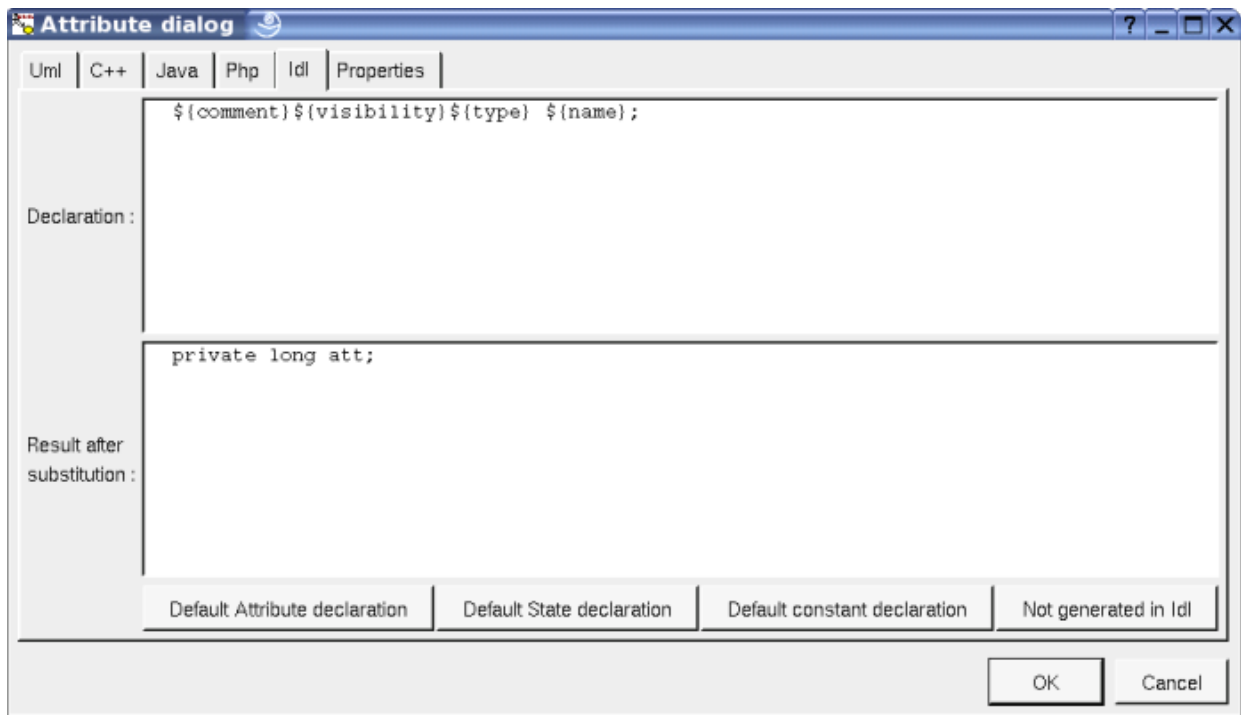


La définition est vide car IDL n'était pas coché lors de la création de l'attribut.



*Dans le cas présent mis à part, ne cochez pas inutilement les langages pour lesquels vous ne générerez pas de code dans le menu *Languages*, sinon votre modèle prendra plus de place que nécessaire en mémoire.*

Appuyons sur le bouton *Default State declaration*, on obtient :

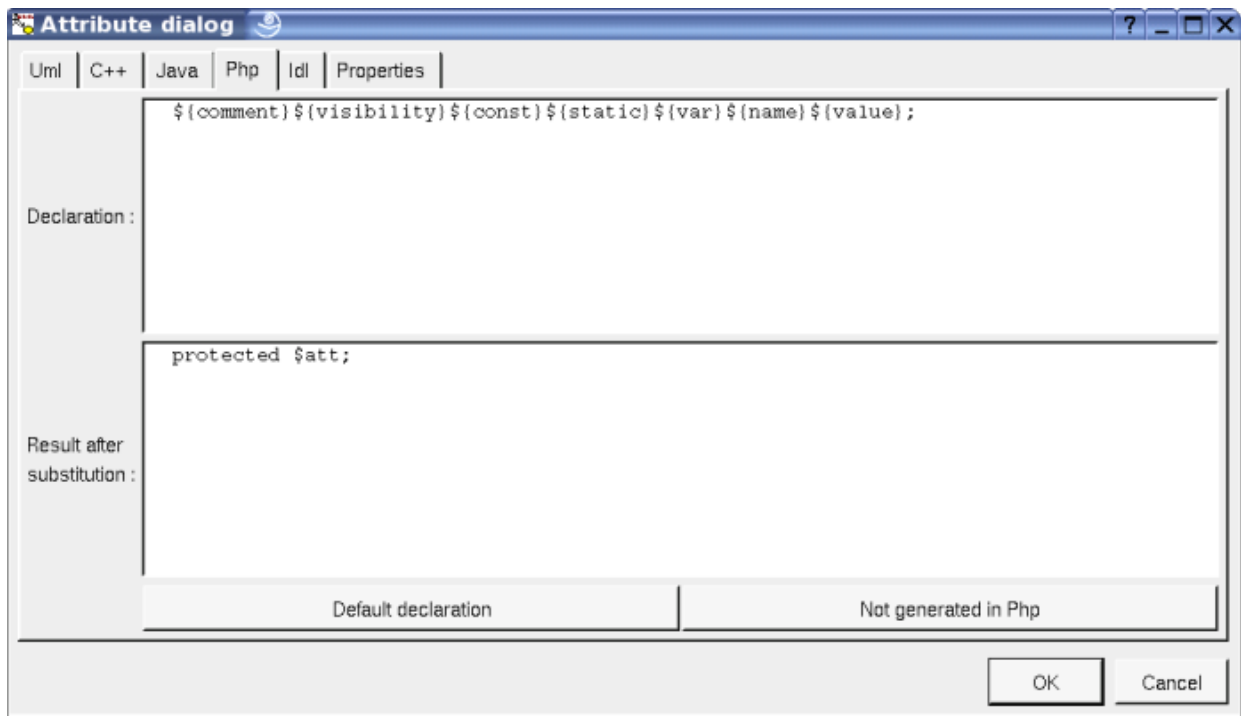


Premièrement la visibilité est *privé* et non *protégé* car cette dernière n'existe pas en *IDL*.

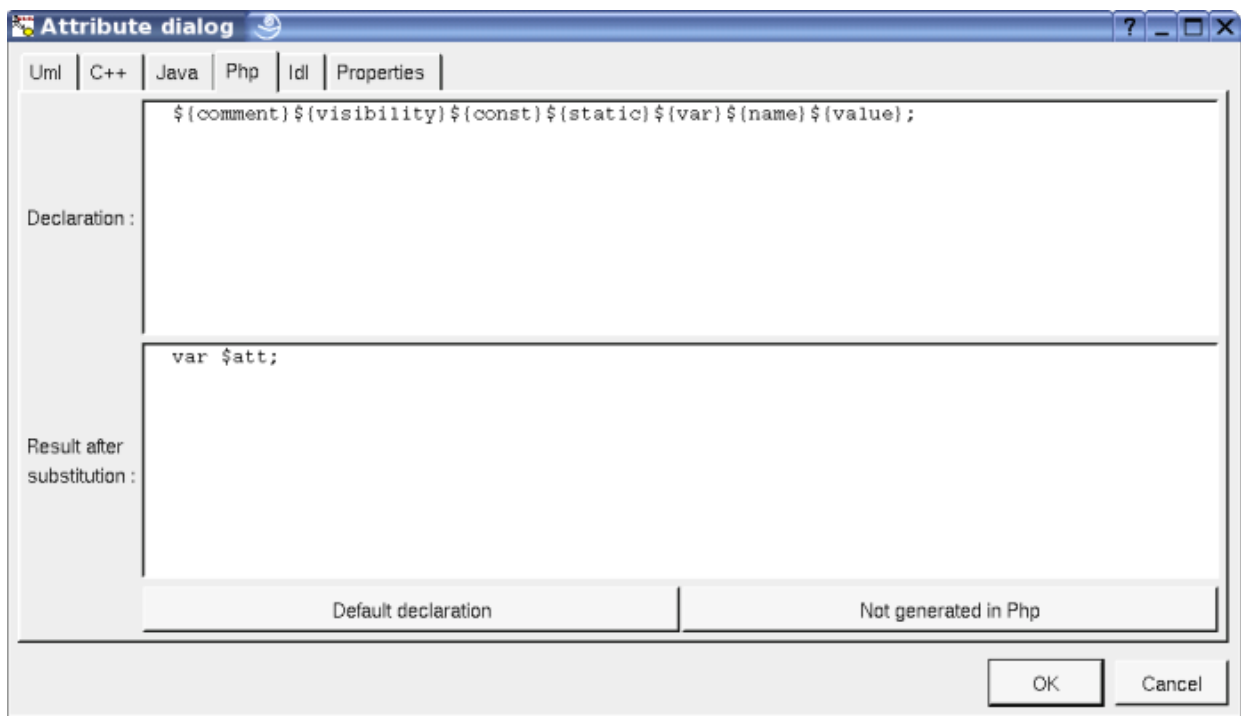
D'autre part *\${type}* produit *long* plutôt que *int*, ceci permet de ne pas produire un code *IDL* faux et par défaut un *int* est traduit par *long* en *IDL*, magique, et surtout pratique !

Il y a d'autres conversions automatiquement faites par BOUML, par exemple si le type est *any* en UML vous aurez *void ** en C++, *Object* en Java et *any* en *IDL*. Bien évidemment ces conversions ne sont pas faites en dur et imposées par BOUML, elles sont définies et modifiables *via* les *generation settings*.


Allons dans l'intercalaire *Php*, la définition est vide comme pour *Idl*, appuyons sur le bouton *Default declaration*, on obtient :



C'est une définition pour Php 5, si vous voulez produire du code pour Php 4 le mieux est de changer la visibilité (intercalaire UML) en *package*, et la définition devient :



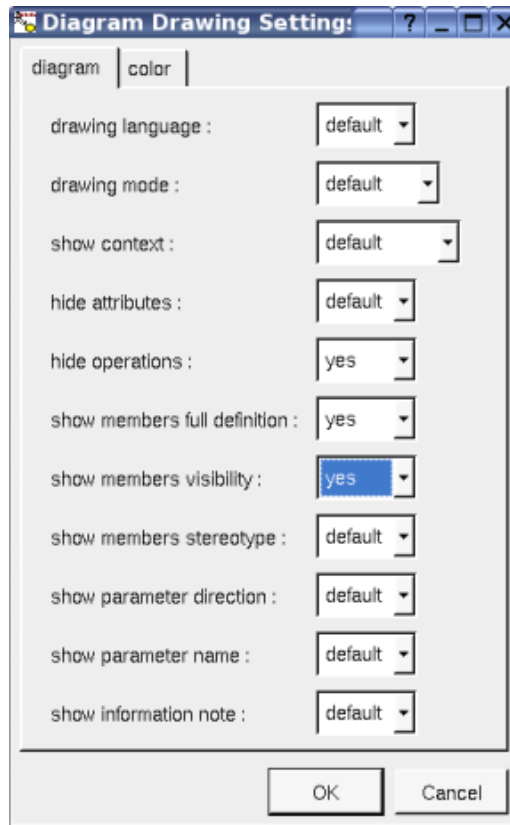
Les définitions par défaut, par exemple celles des attributs, sont également modifiables *via* les *generation settings*, cela vous permet par exemple d'ajouter le '_' au nom de l'attribut en C++ comme ci dessus dans la définition par défaut et non de le faire sur chaque attribut, ce qui serait particulièrement pénible. Je vous renvoie au [manuel de référence](#) pour plus de détails.

 **Quand vous changez les generation settings, les définitions des éléments existants ne sont heureusement pas modifiées.**

Faites *ok* pour valider les changements.

II-D - Modifier le dessin d'une classe

Par défaut dans un diagramme de classe seul le nom des attributs et opérations est visible, pas leur type ni leur visibilité. Appeler le menu de la classe dans le diagramme et changer trois *drawing settings* pour obtenir :



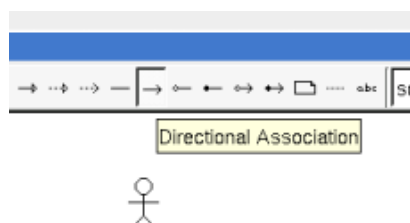
Le dessin de la classe devient :

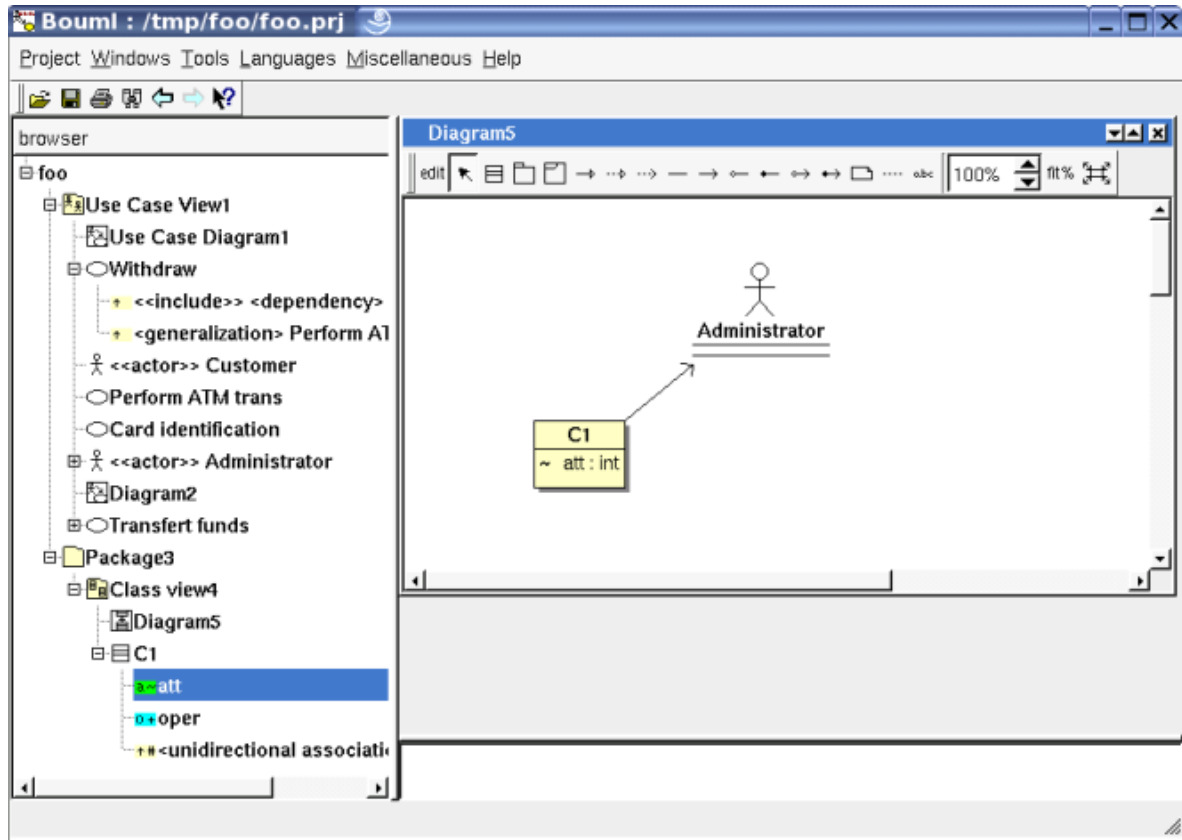


N'hésitez à essayer les autres drawing settings, et lisez le **manuel de référence** pour plus de détails.

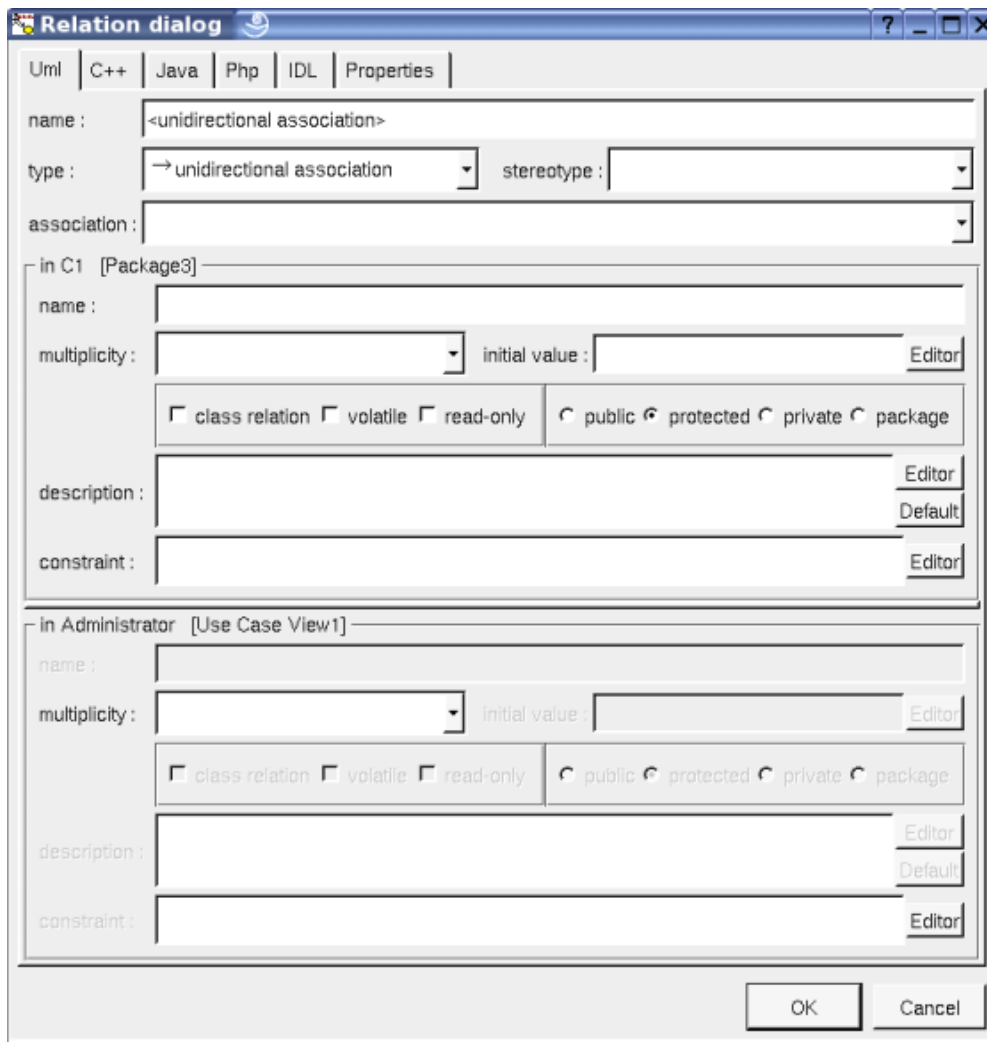
II-E - Relations

Nous voulons ajouter une association unidirectionnelle de *C1* vers *Administrator*, appuyez sur le bouton correspondant et dessinez la relation comme vous l'aviez fait pour celles entre les *use cases* :





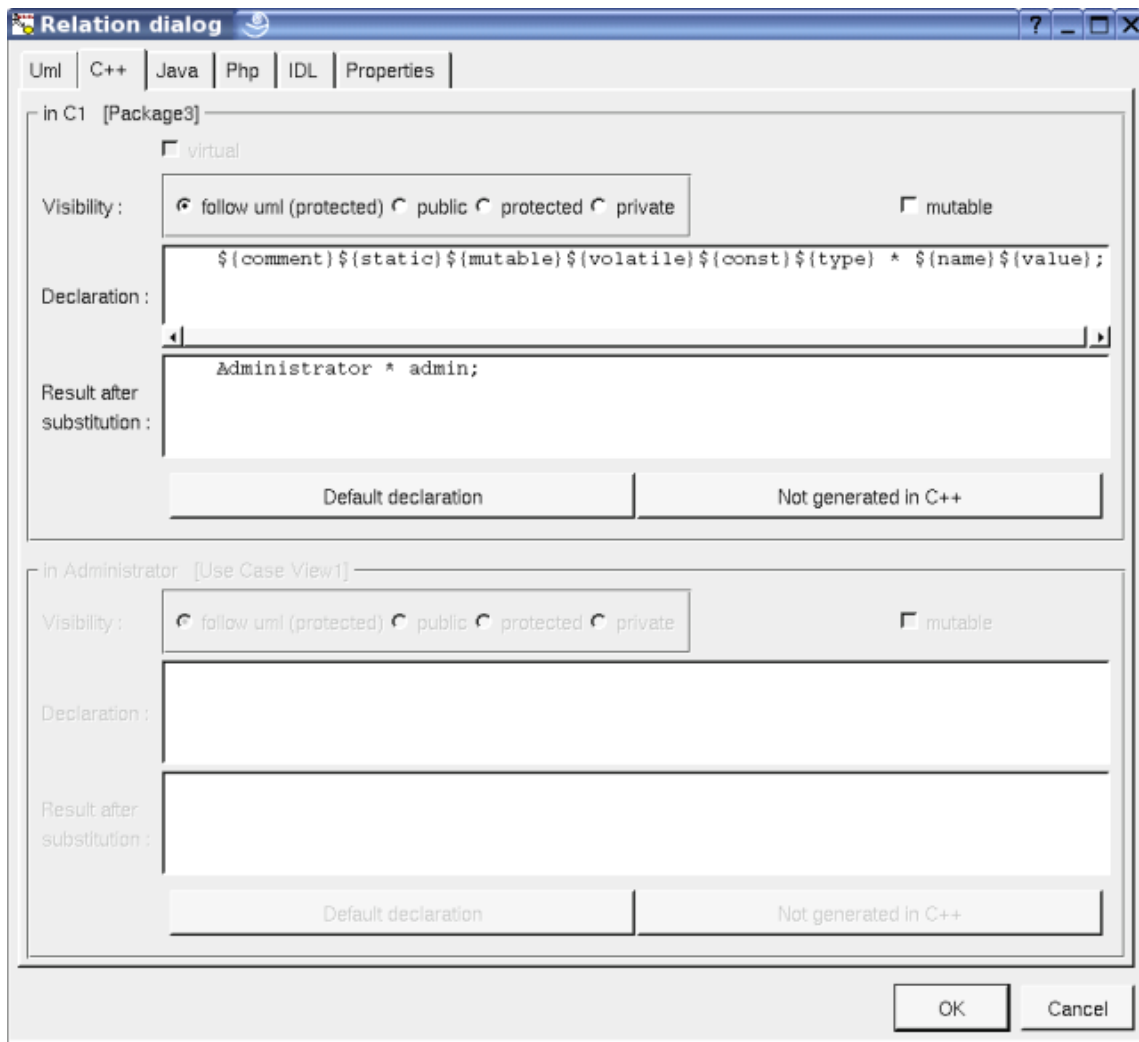
La relation est visible dans le *browser*, par défaut son nom est *<unidirectional association>*, éditez la relation :



Comme pour les attributs vous avez un intercalaire pour UML et chaque langage.

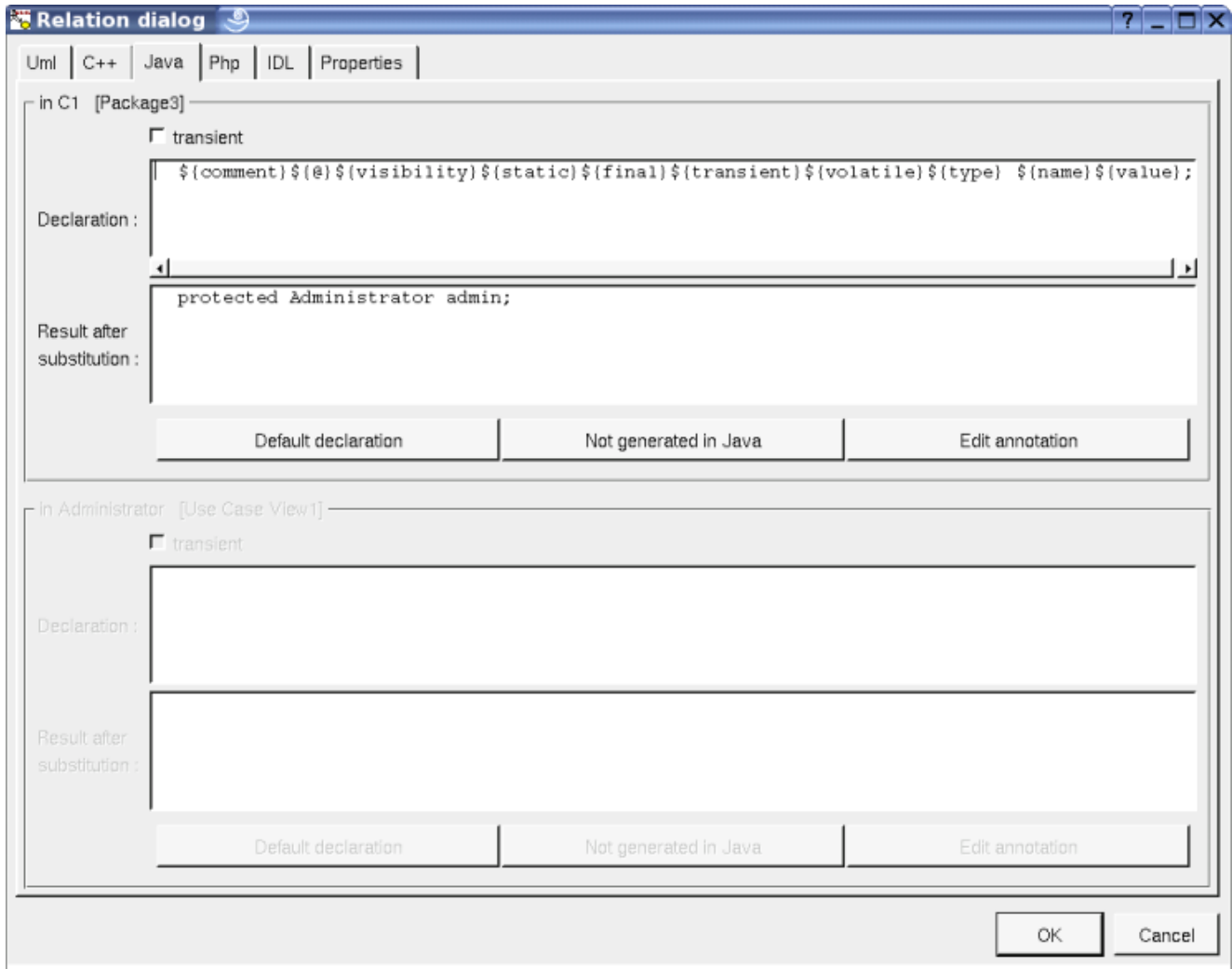
Parce que la relation est unidirectionnelle seul un des deux rôles est éditable. De même que pour les attributs et opérations, la visibilité par défaut est spécifiée *via* les *Class settings*. Le nom du rôle produira le nom du membre lors de la génération de code, et nous retrouvons les même propriétés que pour les attributs : du point de vue génération de code il n'y a pas de différences entre une relation et un attribut.

Changeons le nom en *admin* et allons dans l'intercalaire pour C++ :



Par défaut une association et une agrégation produisent un pointeur en C++, une agrégation par valeur ne produit pas de pointeur. Bien sûr ces définitions par défaut peuvent être modifiées *via* les *generation settings* ou juste au niveau de chaque relation en changeant le texte placé devant *Declaration*.

Allons dans l'intercalaire *Java*, évidemment pas de pointeur :



Relation dialog

Uml | C++ | Java | Php | IDL | Properties

in C1 [Package3]

☐ transient

Declaration : `${comment} ${@} ${visibility} ${static} ${final} ${transient} ${volatile} ${type} ${name} ${value};`

Result after substitution : `protected Administrator admin;`

Default declaration | Not generated in Java | Edit annotation

in Administrator [Use Case View1]

☐ transient

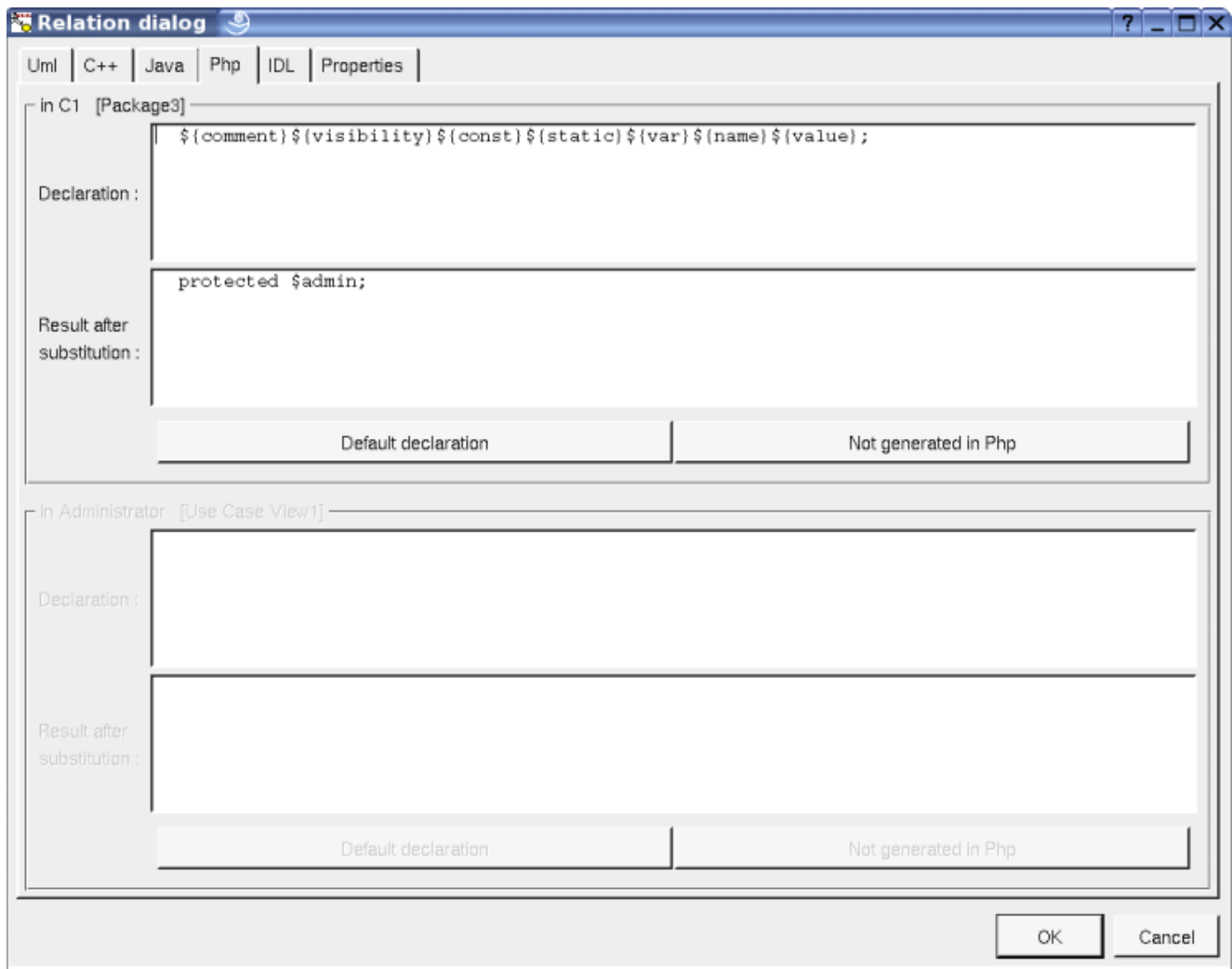
Declaration :

Result after substitution :

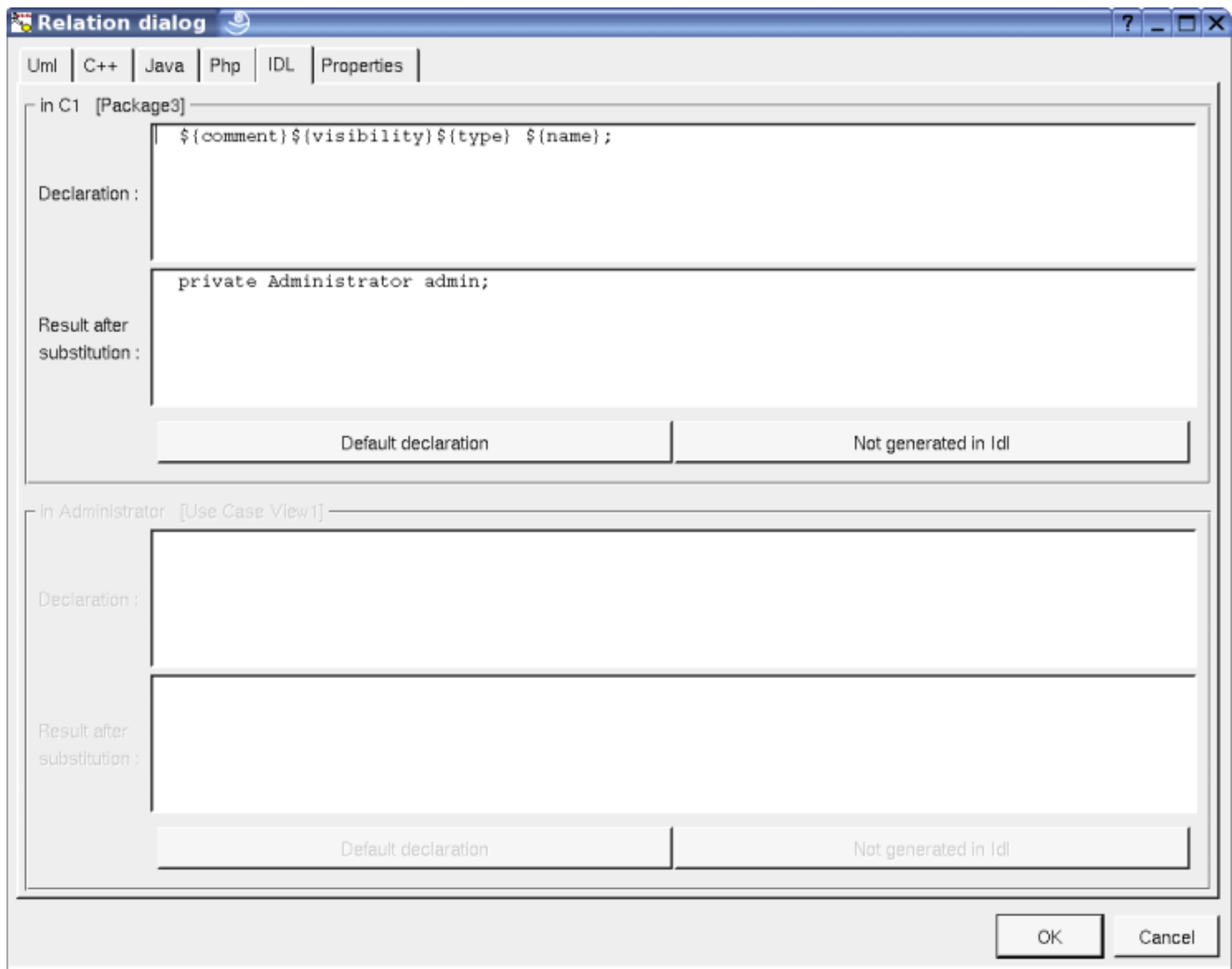
Default declaration | Not generated in Java | Edit annotation

OK Cancel

Allonsz dans l'intercalaire *Php* :



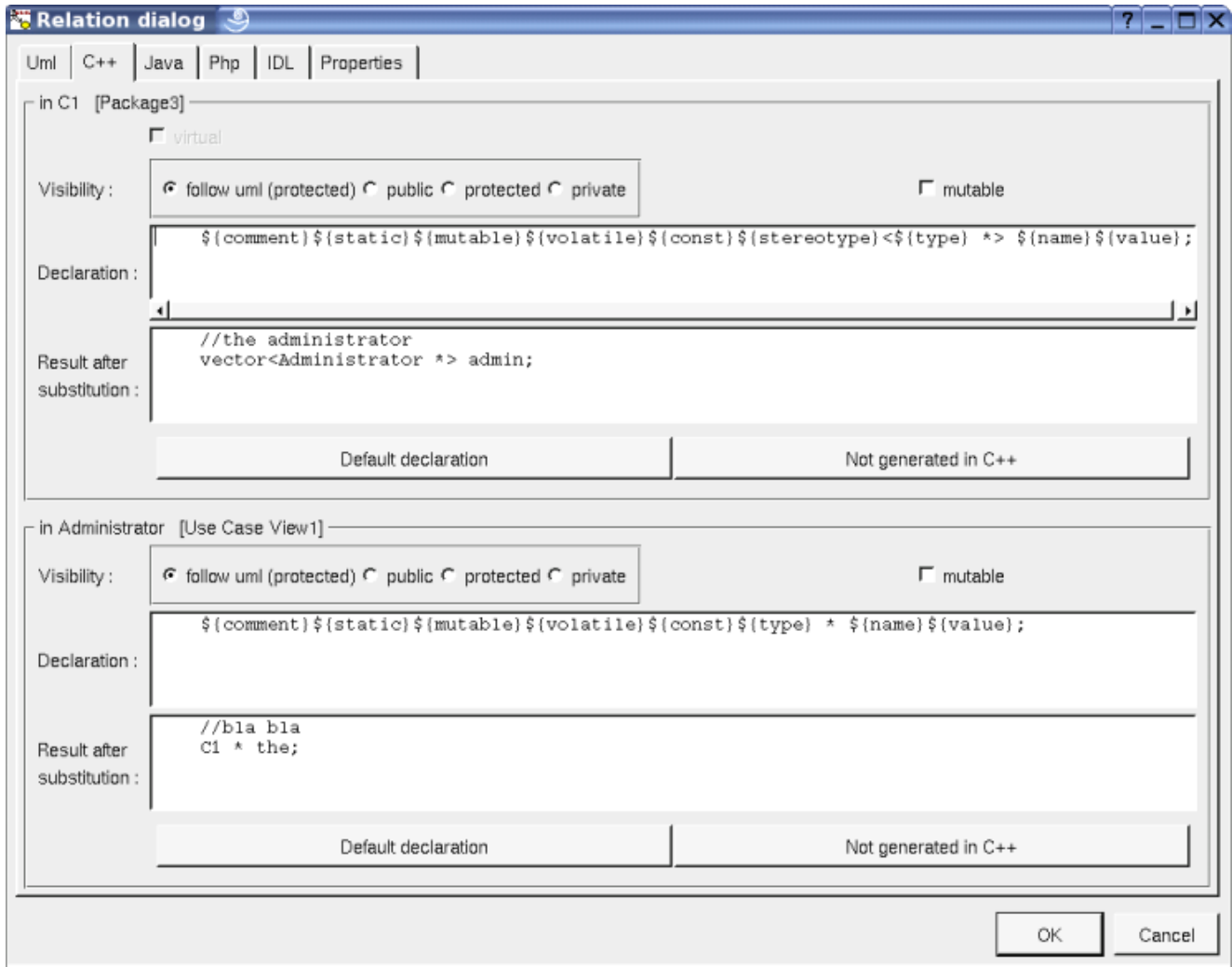
Allez dans l'intercalaire *Php* :



Retournons dans l'intercalaire UML, changeons d'abord le type de la relation pour avoir une association (bidirectionnelle) puis faisons les autres modifications pour avoir :

The screenshot shows the 'Relation dialog' window in Bouml. The 'Uml' tab is selected. The 'name' field contains '<association>'. The 'type' dropdown is set to 'association' and the 'stereotype' dropdown is set to 'vector'. The 'association' dropdown is empty. The dialog is divided into two sections: 'in C1 [Package3]' and 'in Administrator [Use Case View1]'. In the 'in C1' section, the 'name' is 'admin', 'multiplicity' is '1..*', and 'description' is 'the administrator'. In the 'in Administrator' section, the 'name' is 'the', 'multiplicity' is empty, and 'description' is 'bla bla'. Both sections have checkboxes for 'class relation', 'volatile', and 'read-only', and radio buttons for 'public', 'protected', 'private', and 'package'. The 'Default' button is visible next to the 'description' field in both sections. The 'OK' and 'Cancel' buttons are at the bottom right.

Allons dans l'intercalaire pour C++ et appuyons sur *Default declaration* pour les deux rôles, on obtient :



Relation dialog

Uml | C++ | Java | Php | IDL | Properties

in C1 [Package3]

☐ virtual

Visibility: ☒ follow uml (protected) ☐ public ☐ protected ☐ private ☐ mutable

Declaration: `${comment} ${static} ${mutable} ${volatile} ${const} ${stereotype} <${type} * > ${name} ${value};`

Result after substitution: `//the administrator
vector<Administrator * > admin;`

Default declaration Not generated in C++

in Administrator [Use Case View1]

Visibility: ☒ follow uml (protected) ☐ public ☐ protected ☐ private ☐ mutable

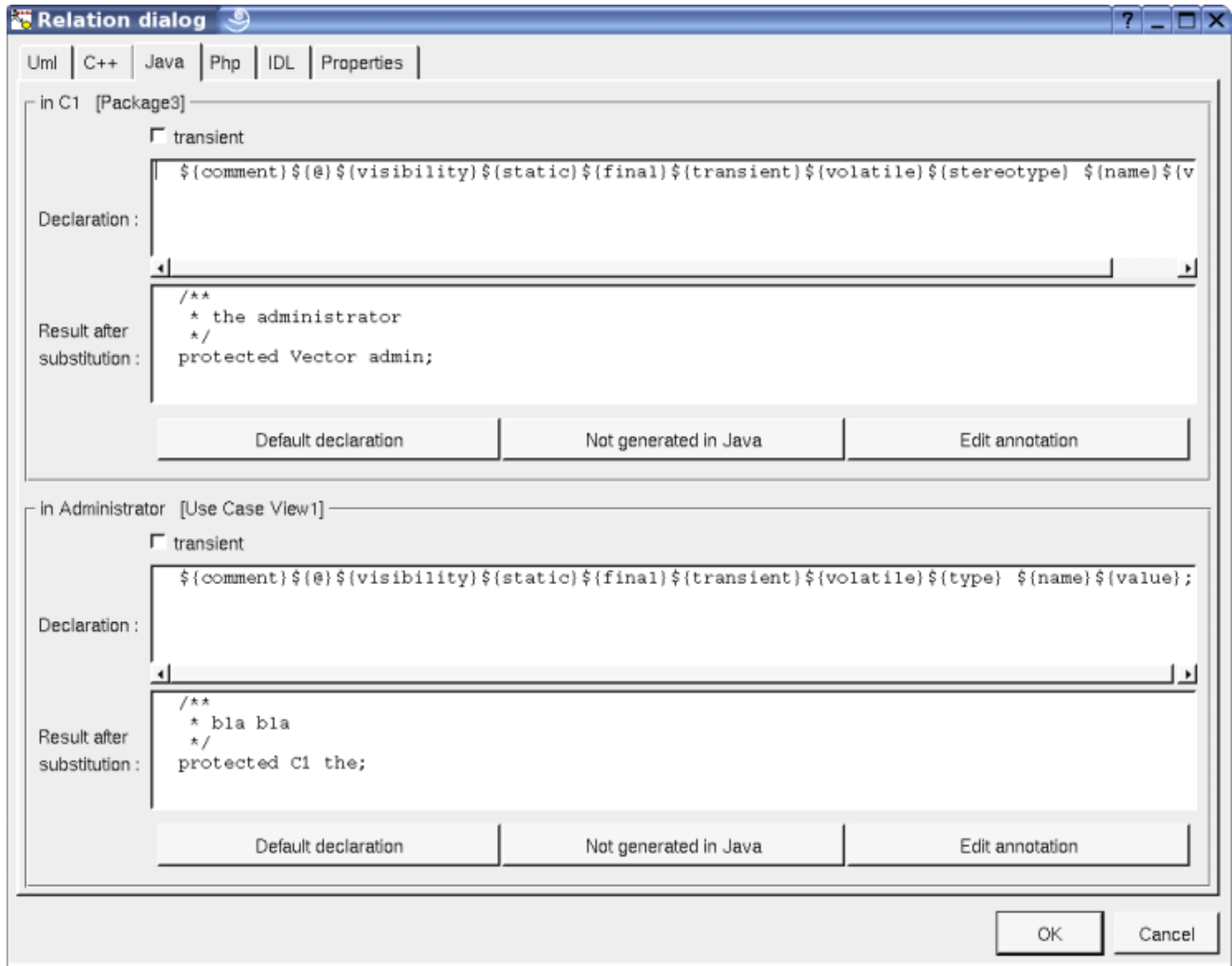
Declaration: `${comment} ${static} ${mutable} ${volatile} ${const} ${type} * ${name} ${value};`

Result after substitution: `//bla bla
C1 * the;`

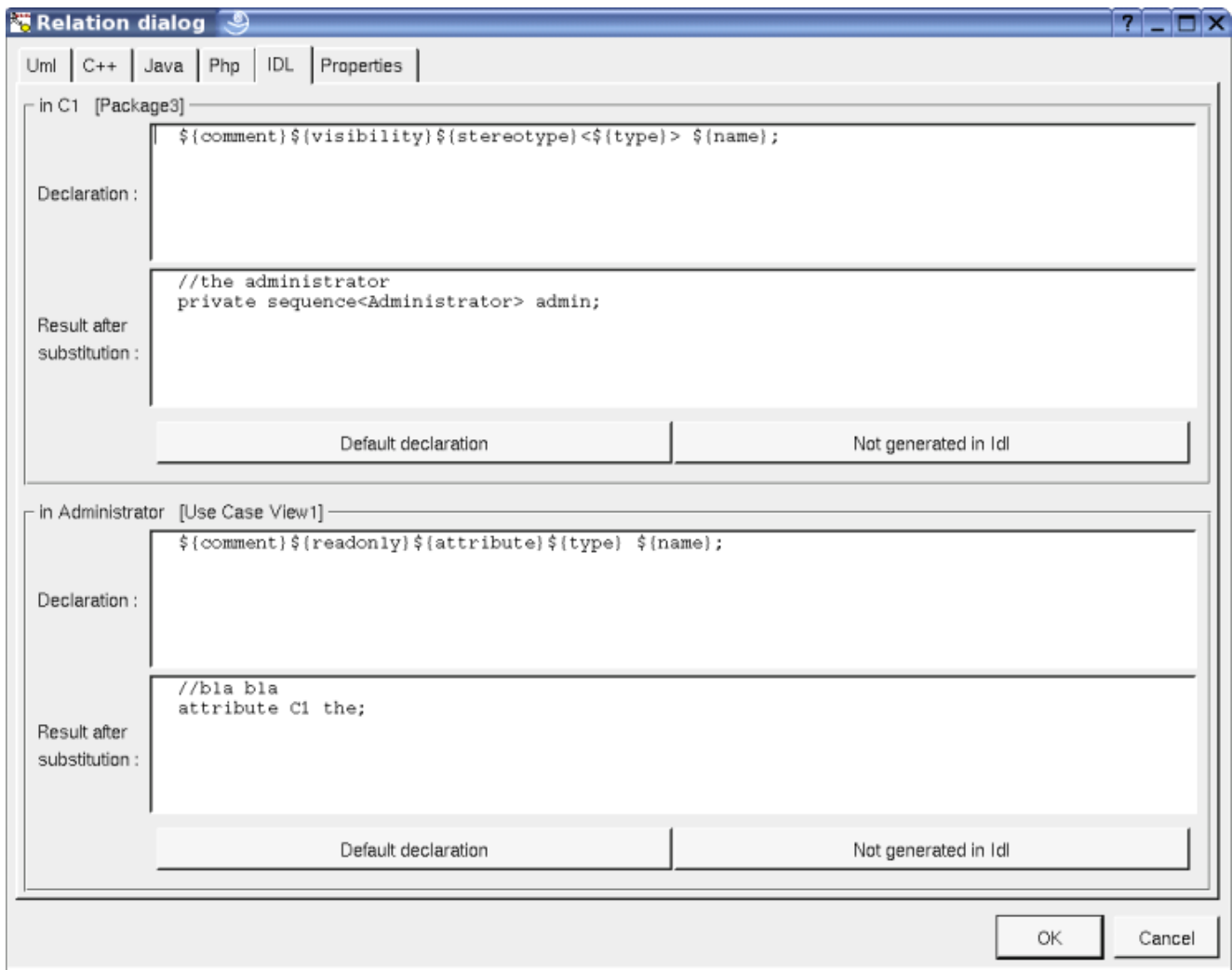
Default declaration Not generated in C++

OK Cancel

Allons dans l'intercalaire *Java* et appuyons sur *Default declaration* pour les deux rôles, on obtient :

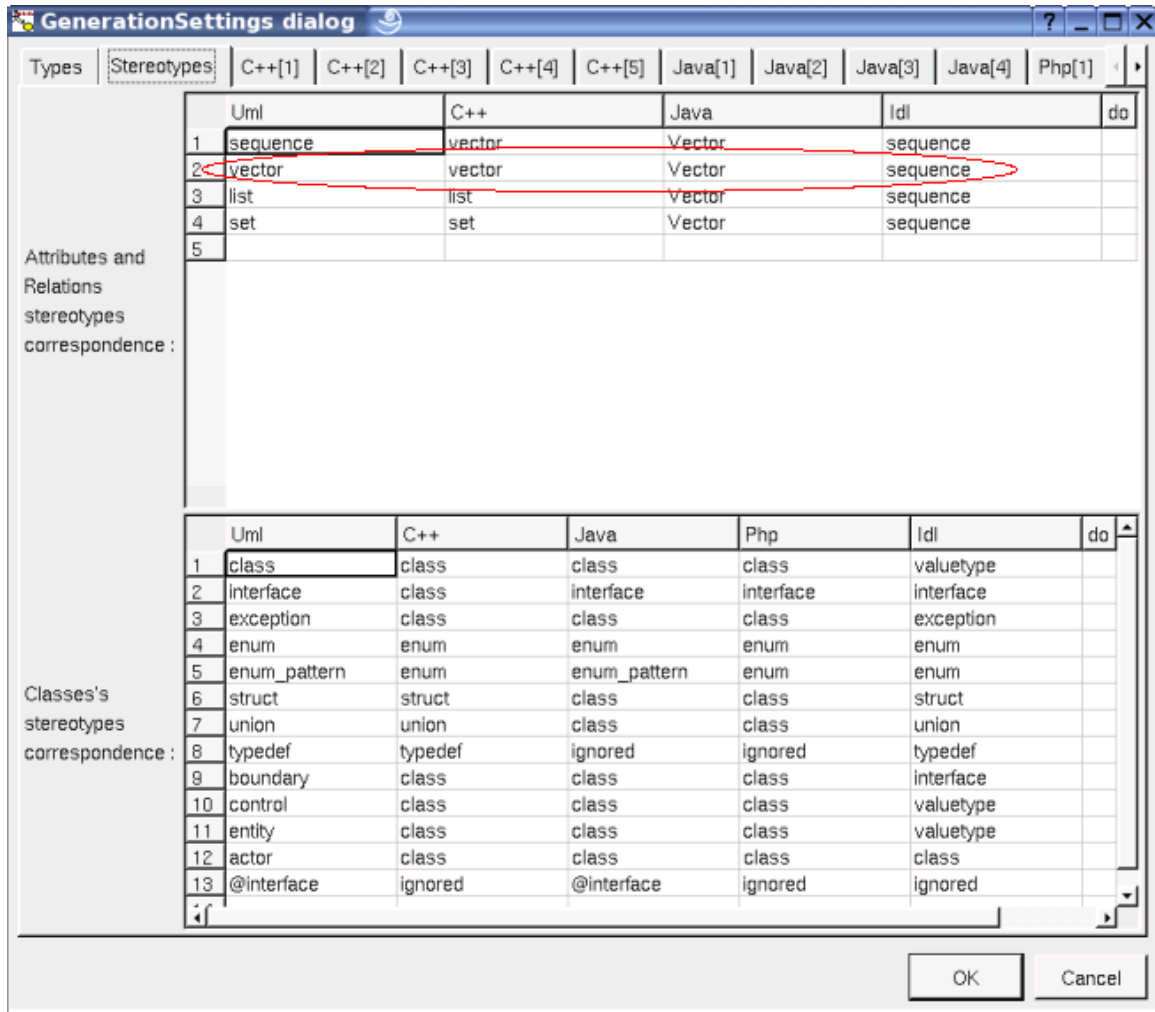


Faisons de même pour *IDL*, on obtient :

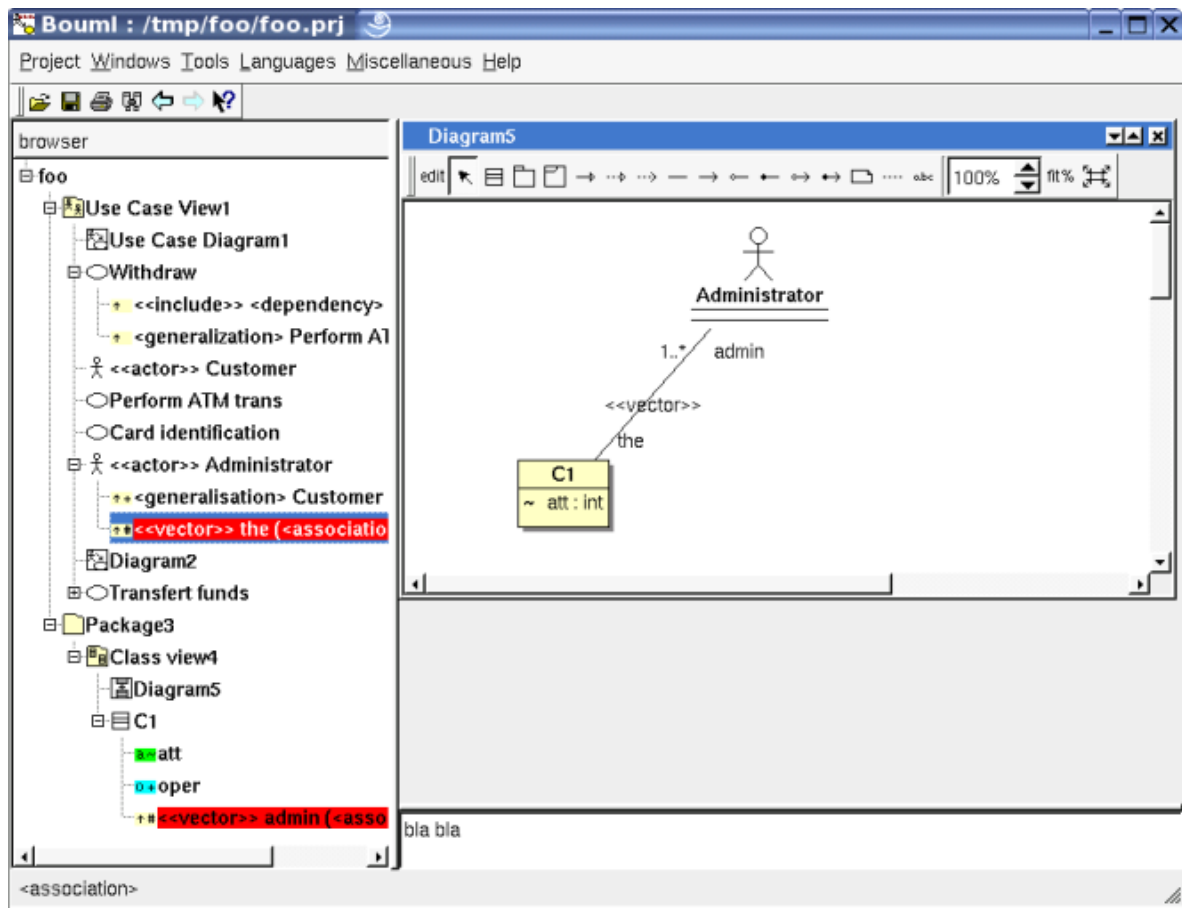


Comme vous l'avez vu, les définitions par défaut tiennent naturellement compte du type de la relation et de la multiplicité, bien sûr tout est modifiable *via* les *generation settings* : vous décidez de ce qui doit être produit lors de la génération de code.

La façon dont les stéréotypes sont projetés pour chaque langage est également définie *via* les *generation settings*, ainsi *vector* produit *vector* en C++, *Vector* en Java, et *sequence* en IDL :



Validons les modifications de la relation et regardons le diagramme :



Parce que la relation est maintenant bidirectionnelle celle-ci apparaît dans le *browser* sous *C1* et *Administrator* (je les ai marqué dans le *browser*, c'est pourquoi elles sont rouges, les marques permettent de faire des sélections multiples dans le *browser* de façon plus résistante sans craindre un clic de travers les annulant).

III - Classes spéciales

III-A - Définir des structures

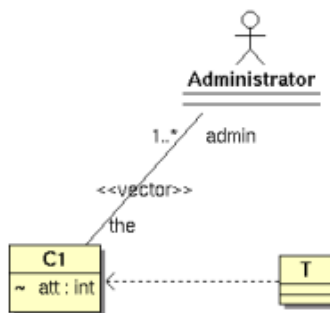
Pour définir une structure en *C++* ou *IDL* il faut changer le stéréotype d'une classe en *struct*, et utiliser le bouton *Default declaration* dans le(s) langage(s) désiré(s). En *Java* et *Php* une structure est implémentée *via* une classe standard, bien sûr la projection d'un stéréotype d'UML vers un langage donné est définie *via* les *generation settings*, regardez la boîte de dialogue précédente et reportez-vous au **manuel de référence**.

III-B - Définir des unions

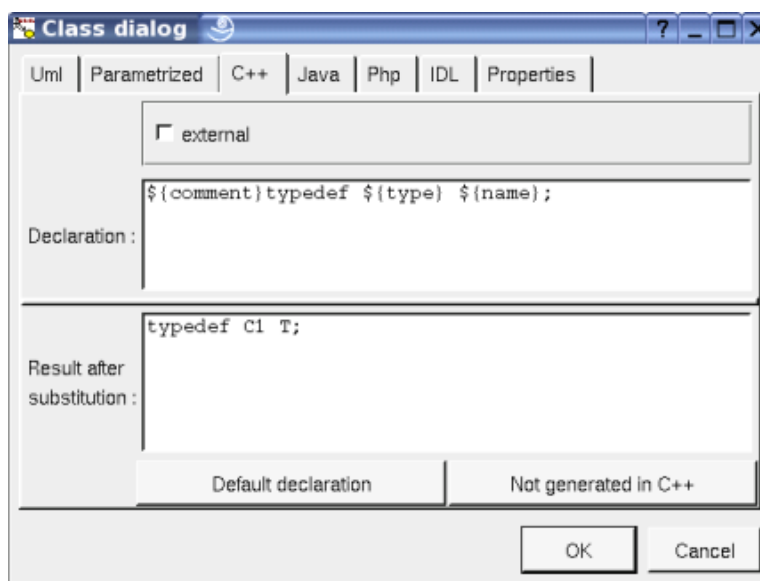
Pour définir une *union* en *C++* ou *IDL* utilisez le stéréotype de classe *union*, puis utiliser le bouton *Default declaration* dans le(s) langage(s) désiré(s).

III-C - Définir des typedefs

Pour définir en *C++* le *typedef* *T* étant un pointeur sur *C1* : créez la classe *T*, et dessinez une dépendance de *T* vers *C1* :

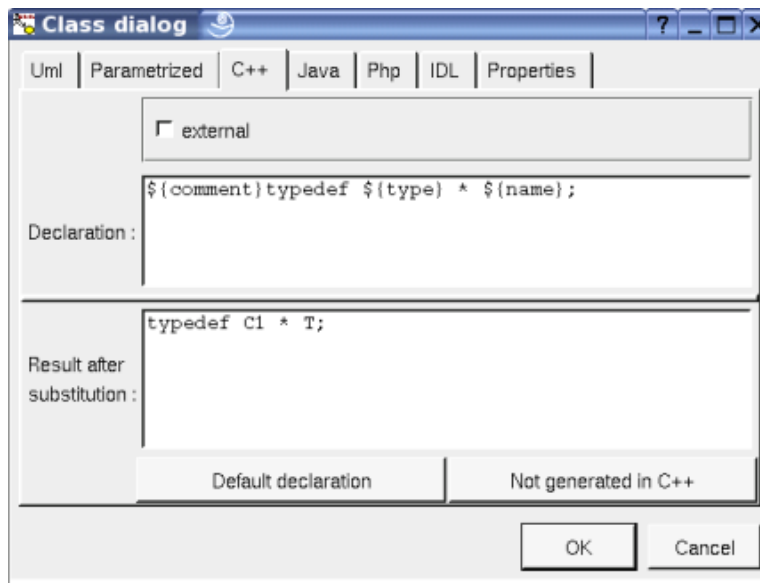


Éditez la classe et mettez son stéréotype à *typedef* : le dialogue change et indique un type de base (*base type*), grâce à la dépendance ce type de base et mis à *C1*, mais vous pouvez le changer.



Bien évidemment la dépendance n'est pas obligatoire, par exemple pour avoir *typedef int turlututu*.

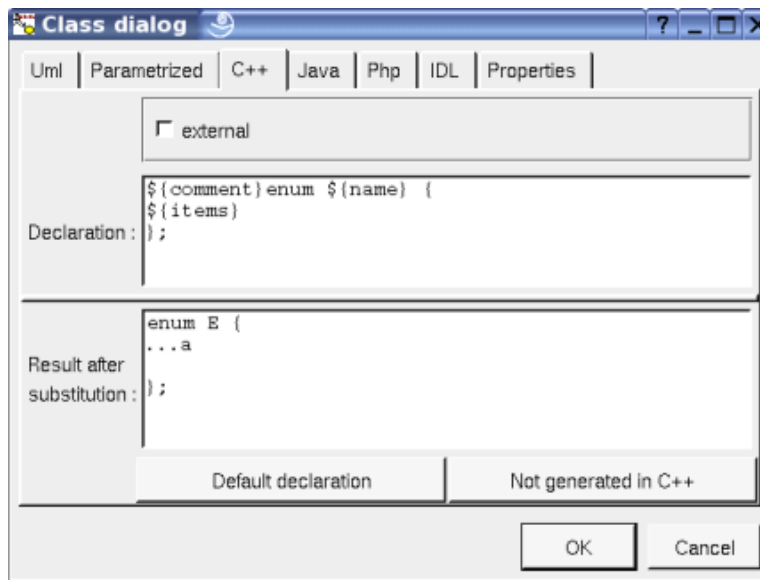
Allez dans l'intercalaire C++ et ajoutez une "*" entre $\{type\}$ et $\{name\}$:



III-D - Définir des énumérations

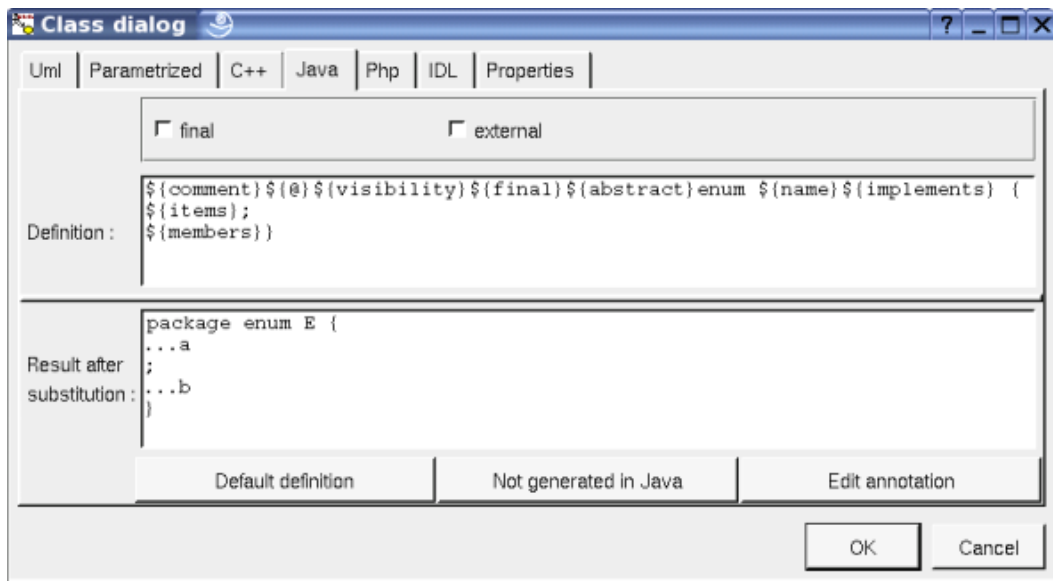
Les énumérations sont supportées *via* les stéréotypes de classe *enum* et *enum_pattem*.

Créez la classe *E*, éditez *E* et changez son stéréotype en *enum* puis validez par *Ok*. Appelez le menu de la classe *E* dans le *browser* et choisissez *add item* et nommez le *a*. Maintenant rappelez le menu et choisissez *add attribute* et nommez le *b*. Éditez la classe *E* et allez dans l'intercalaire C++, demandez la définition par défaut :



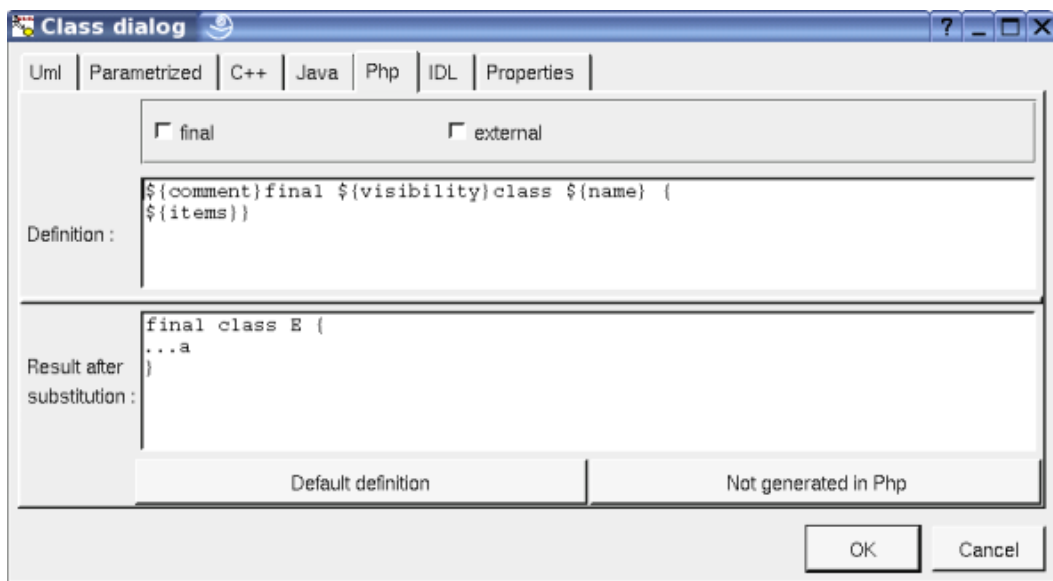
Comme vous le voyez, $\{items\}$ produit 'a' (les '...' indiquant que seul le nom est affiché, pas la définition complète) mais pas 'b', ceci parce qu'en C++ une énumération peut seulement avoir des items, les attributs et opérations étant illégaux.

Allons dans l'intercalaire *Java*, c'est une énumération pour au moins la *JDK 5* :

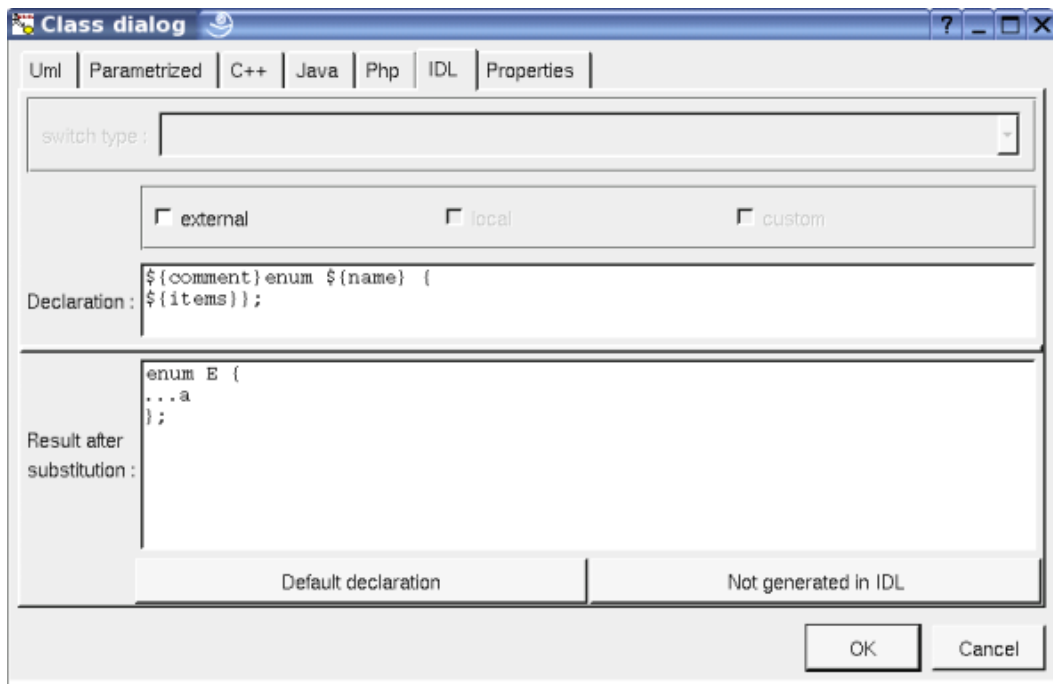


'b' est produit par `${member}`, le stéréotype *attribute* permet de distinguer items et attributs.

Allons dans l'intercalaire pour *Php*, comme en *C++* 'b' n'apparaît pas :

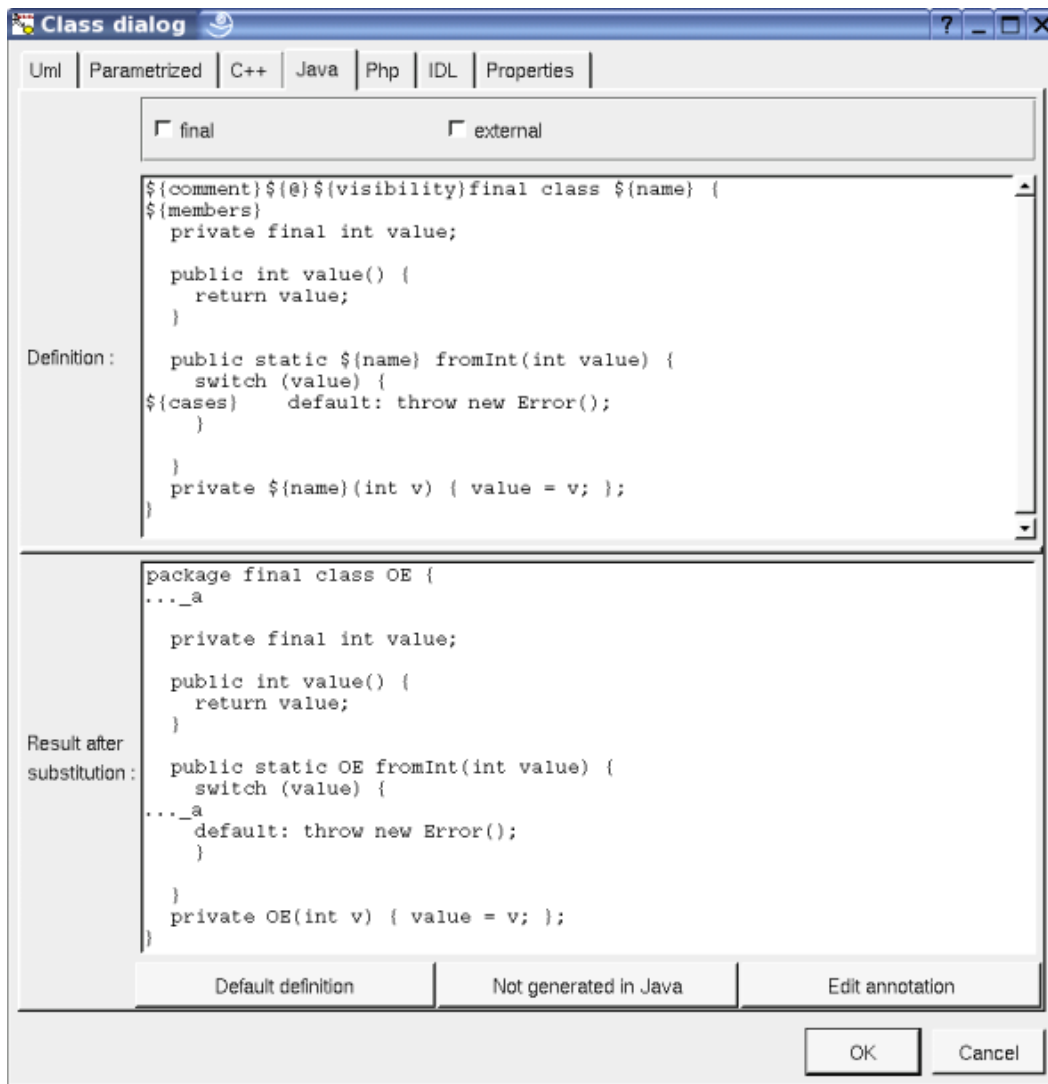


Allons dans l'intercalaire pour *IDL*, comme en *C++* et *Php* 'b' n'apparaît pas :



De la même façon, si vous ajoutez une opération à une énumération, celle-ci ne sera pas définie en C++ ni en *IDL*.

Maintenant créez la classe *OE*, éditez la pour mettre son stéréotype à *enum_pattern* puis validez ma modification via *Ok*. Regardez le menu de *OE* dans le *browser* : il est seulement possible d'ajouter des items. Il n'y a aucune différence entre *E* et *OE* en C++ et en *IDL* mais tout change en *Java* où ce type d'énumération est supportée par une classe standard , ceci permet de définir des Énumérations pour des versions de *Java* précédant la *JDK 5*:



Comme d'habitude cette forme est définie *via* les *generation settings* vous permettant de la modifier.

Je vous propose d'éditer la définition de 'a' dans les classes *E* et *OE* et de regarder leur définition pour tous les langages.

III-E - Définir des interfaces

Pour définir une *interface* en *Java* ou *IDL*, utilisez le stéréotype de classe *interface*, puis utiliser le bouton *Default declaration* dans le(s) langage(s) désirés.

Le stéréotype *@interface* est également pris en compte pour la gestion des annotations en Java.

III-F - Définir des exceptions

Pour définir une *exception* en *IDL*, utilisez le stéréotype de classe *exception*, puis utiliser le bouton *Default declaration* dans le(s) langage(s) désiré(s).

III-G - Templates et Génériques

Une classe *template* en C++ ou *générique* en Java est une classe ayant des *formals* définis via l'intercalaire *parametrized* de l'éditeur de classe.

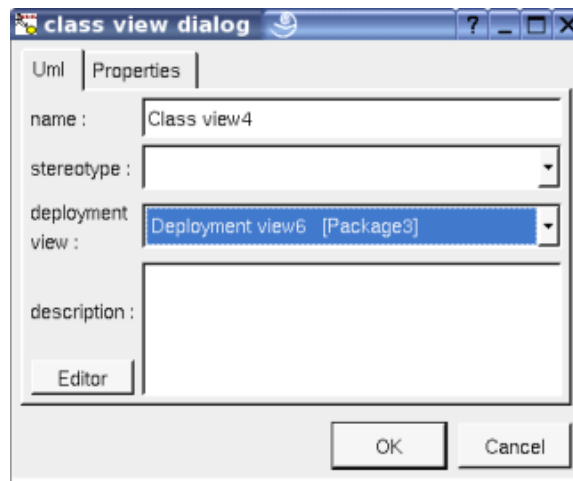
Lorsqu'une classe généralise ou réalise une classe paramétrée, l'intercalaire *instantiate* de l'éditeur de classe vous permet d'indiquer les *actuals*. Bien évidemment une classe paramétrée peut hériter d'une autre, dans ce cas celle-ci possède à la fois des *formals* et des *actuals*.

IV - Génération de code, deployment view, artifact

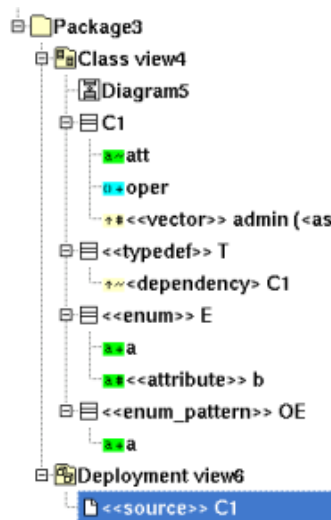
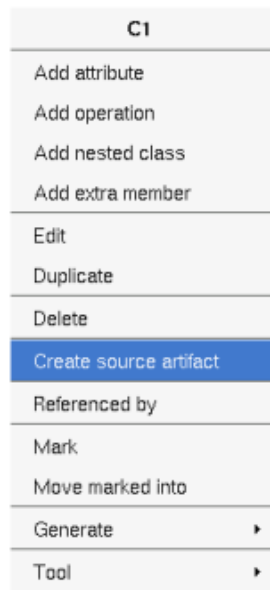
Peut être aviez-vous vu l'entrée *Generate* dans le menu des classes, essayez de générer le code pour *C1* : quelque soit le langage cible vous avez l'erreur *C1 : does not have associated artifact*. En UML 2 les **artifacts** représentent les fichiers sources, objet, librairie, exécutable, *jar* etc. La génération produit des fichiers source, et les *artifacts* sont utilisés pour dire quels sont ces fichiers sources, . Dans les premières versions de BOUML la génération de code était associée aux **components** mais cela n'est pas compatible avec UML 2 (UML 1 était très évasif sur le sujet), j'ai donc préféré faire la modification, contrairement à plusieurs autres modeleurs UML ...

Un *artifact* ne peut être placé que dans une **deployment view**. Créez une *deployment view* dans le package *Package3* (cela n'est pas obligatoire, vous pouvez la placer dans n'importe quel *package*), et supposons qu'elle s'appelle *Deployment view6*.

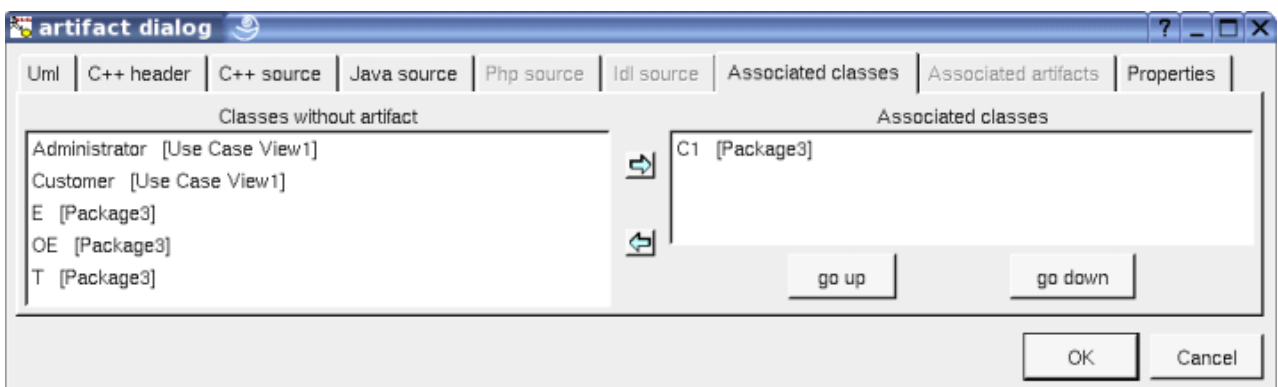
Maintenant vous avez deux façons d'associer chaque classe à un *artifact* de type source. La façon la plus longue est de faire pour chaque classe : créer un *artifact* dans *Deployment view6*, éditez le pour l'appeler comme la classe, changez son stéréotype en *source* et associez le à la classe. La seconde méthode est plus rapide pour créer des *artifacts* pour plusieurs classes : associez *Deployment view6* à *Class view4* (parce qu'il contient *C1* etc), pour cela éditez *Class view4* :



Grâce à cette association, quand vous appelez le menu de *C1* ou des autres classes de *Class view4* dans le *browser*, vous pouvez choisir l'entrée *create source artifact* et l'*artifact* correspondant sera créée. Faites cela pour *C1* et vous obtiendrez:



Éditez l'*artifact* (vous pouvez le sélectionner rapidement en appelant le menu de C1 et en choisissant *select associated artifact*) et allez dans l'intercalaire *associated classes* :



C1 est associée à l'*artifact* auquel vous pouvez associer plusieurs classes, dans ce cas ces classes seront produites dans le même fichier source. Vous pouvez changer l'ordre de génération de ces classes en utilisant les boutons *go up/down*. Les classes seront générées dans l'ordre d'affichage.

Trois intercalaires ne sont pas disponibles :

- *Php source* car la classe associée (ici *C1*) n'a pas de définition en *Php* ;
- *Idl source* car la classe associée (ici *C1*) n'a pas de définition en *IDL* ;
- *Associated artifact* parce que le stéréotype de l'*artifact* est *source*. Lorsque le stéréotype n'est pas *source* vous pouvez associer plusieurs autres *artifacts* à l'*artifact* par exemple pour indiquer que ces *artifacts* forme un exécutable (cette convention est suivit pas le *plug-out genpro*).

Regardez les autres intercalaires et reportez-vous au [manuel de référence](#) pour plus de détails.

Créez un *artifact* pour les classes *T*, *O* et *OE*.

Essayez de nouveau de générer le code, cela est possible en appelant le menu sur chaque classe, ou au niveau des *class view* les contenant, ou au niveau du *package* contenant les vues etc jusqu'au niveau du projet. En le faisant au niveau du projet vous demandez la génération de tous les *artifact* et des classes qui leur sont associées. Provoquer la génération *via* le menu *Tools* est une autre façon de demander la génération au niveau projet.

Cette nouvelle tentative de génération de code produira un autre message d'erreur, car BOUML ne sait pas où les fichiers sources doivent être produits : les artifacts donnent le nom des fichiers, pas leur emplacement !

Il est possible d'indiquer un répertoire de génération pour chaque *package* (vous pouvez également spécifier un *namespace*, *package* ou *module*), il est également possible de spécifier un répertoire de génération (ou un répertoire de base) pour tout le projet *via* les *generation settings*. Appelez le menu du projet dans le *browser* (clic droit sur *foo*) et choisissez *generation settings*, allez dans le dernier intercalaire appelé *Directory* et spécifiez un répertoire pour *C++* et *Java*, par exemple */tmp* pour les deux (utilisez d'autres et meilleurs répertoires dans le futur !), validez le changement (bouton *Ok*).

Maintenant si vous demandez la génération de code, celle-ci se fera. Demandez à générer le code une deuxième fois pour le même langage : la trace indique que rien n'est produit car c'est inutile. Notez que le générateur de code vérifie véritablement que chaque fichier présent a le bon contenu, si vous changez le contenu d'un de ces fichiers à la main et redemandez la génération de code, le fichier en cause sera réécrit. Cela signifie que même si ce n'est pas toujours approprié au niveau vitesse (surtout si seul un fichier est modifié sur 1000), vous pouvez toujours demander la génération au niveau du projet et aucune modification inutile ne sera faite, au plus grand plaisir de vos *Makefile* et gestionnaires de configuration.

V - Conclusion

V-A - Epilogue

Maintenant vous êtes prêts à utiliser BOUML ... et à lire le manuel de référence car seule une faible partie des possibilités offertes par BOUML est indiquée dans les tutoriels.

Pour finir, n'oubliez pas que :

- BOUML est extensible *via* les *plug-outs*. Un *plug-out* est défini *via* BOUML et peut être implémenté en C++ ou Java, les *plug-outs* existants peuvent vous servir d'exemples (en attendant la diffusion de tutoriels sur les *plug-outs*).
- Quand vous ne savez pas comment produire un code donné, le mieux est probablement d'écrire ce que vous voulez générer (ou du moins une partie) à la main dans un fichier, de faire un **reverse** de ce code manuel dans un nouveau projet (pour ne pas polluer votre projet courant), et de regarder attentivement ce que le *reverse* a produit dans le modèle. Bien évidemment un *reverse* peut être également utilisé pour constituer un projet à partir d'une application déjà écrite.
- Programmeurs Java : utilisez le *plug-out Java Catalog* !

V-B - Liens

- Le site de Bouml : <http://bouml.free.fr>
- Le tutoriel *premiers pas avec Bouml* est [ici](#)
- Le tutoriel *Réalisation d'un plug-out de tri pour BOUML* est [ici](#)

V-C - Remerciements

Merci à **Nip** et à **Miles** pour la relecture de ce tutoriel.

Bonnes modélisations !