Utilisation de Bouml en projet avec un dépôt SVN

Bouml

BOUML est un logiciel de création de diagrammes UML (norme 2.0). Il est multilingue, supporte la génération de code et la rétro-ingénierie.

UML (*Unified Modeling Language*) est un langage de modélisation graphique à base de pictogrammes. Il est utilisé pour le développement logiciel en orientée objet. Wikipedia

L'ergonomie et la richesse graphique de BOUML sont parfois limitées, tout au moins en comparaison de modeleurs UML plus conventionnels comme StarUML ou ArgoUML. Par contre, il est jugé extrêmement efficace pour la « rétro-modélisation » (créer un modèle UML à partir de codes sources) et pour le développement *roundtrip* (faire des aller-retour entre modèle UML et code source).

BOUML permet ainsi de générer automatiquement du code à partir des diagrammes de classes UML, ainsi que de reconstruire ces derniers à partir de sources existantes, pour les langages C++, Java, PHP et MYSQL. Pour les langages Python et IDL, il peut générer du code mais ne permet pas la rétro-conception UML.

Lien : www.bouml.fr

BOUML est utilisé pour spécifier, visualiser, modifier et construire les documents nécessaires au bon développement d'un logiciel orienté objet.

Avec BOUML, il est également possible de générer automatiquement une partie du code (génération de squelette de code par exemple en langage Java, C++, PHP, …) à partir des divers documents réalisés.

Il est possible de l'utiliser avec un système de gestion de version tel que SVN. Nativement, il supporte la synchronisation de plusieurs utilisateurs sur le même projet.

À ce jour, BOUML n'existe pas en tant qu'extension à un EDI (Environnement de développement intégré).

BOUML se décompose, comme UML, en plusieurs sous-ensembles :

- **Les vues** : elles décrivent le système d'un point de vue donné, qui peut être organisationnel, dynamique, temporel, architectural, logique, etc. En combinant toutes ces vues, il est possible de définir (ou retrouver) le système complet.
- **Les diagrammes** : ils décrivent le contenu des vues. Le diagramme de classes est généralement considéré comme l'élément central d'UML.
- **Les modèles d'élément** : ils sont les briques des diagrammes UML.

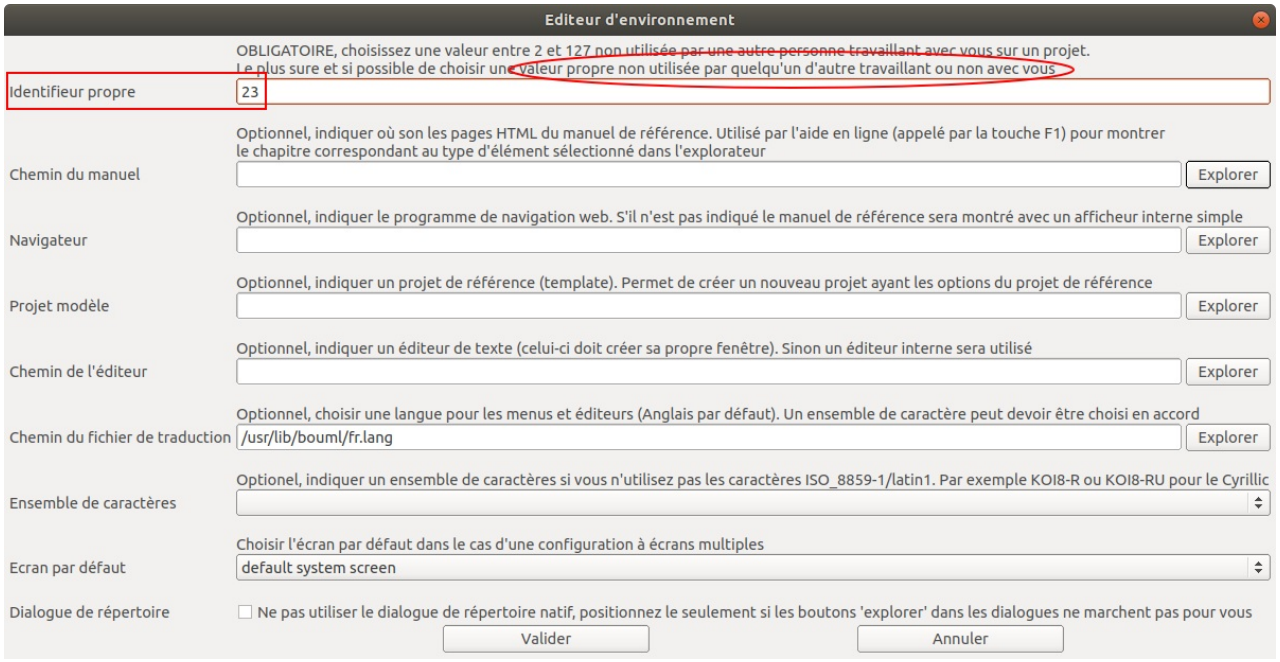
À cela s'ajoute la notion de **paquetage** (*package*). Les paquetages sont des éléments d'organisation : ils

permettent de regrouper des éléments de modélisation UML. Ils sont parfois associés à des éléments concret comme les répertoires.

Aller plus loin : UML-PresentationBOUML.pdf et tp-poo-bouml.pdf

Description des fichiers d'un projet Bouml

BOUML attribue un identifiant aux objets principaux. Cet identifiant est établi lors de la création de l'objet et ne changera pas plus tard, pour permettre à plusieurs utilisateurs de travailler simultanément sur le même projet, cet identifiant d'objet contient l'identifiant de l'utilisateur créant l'objet . Cela explique pourquoi BOUML vous demande d'avoir un propre identifiant entre 2 et 127.



Remarque : La **X** précise si le fichier doit être conservé dans le dépôt SVN.

- **X** `<nom du projet>.prj` : contient le *package* de niveau supérieur et les données de niveau projet, par exemple les répertoires de génération.
- **X** `<numéro>` : contient les données d'un *package* (un *package* uml) et ses sous-éléments. Le numéro est l'identifiant du *package*.
- **X** `<numéro>.diagram` : contient une définition de diagramme. Le numéro est l'identifiant du diagramme.
- **X** `<numéro>.bodies` : contient les corps C ++ de toutes les opérations d'une classe. Le numéro est l'identifiant de la classe.
- **X** `generation_settings` : contient les paramètres de génération.
- **X** `tools` : contient la déclaration des *plug-ins*.
- **X** `cpp_includes` : contient les spécifications d'inclusion et d'utilisation C ++ pour les types non définis, utilisées par le générateur C ++ et définies via les paramètres de génération C ++.
- **X** `java_imports` : contient les importations Java associées aux types non définis, pas déjà utilisées par le générateur Java et définies via les paramètres de génération Java.

- `idl_includes` : contient les importations d'importation Idl vers les types non définis, non déjà utilisées par le générateur Idl et définies via les paramètres de génération Idl.
- * `<id utilisateur>.session` : contient l'état de la session (diagrammes ouverts, vue navigateur …), défini à la fermeture du projet.
- `<numéro>_<id utilisateur>.d` : contient la nouvelle définition d'un diagramme jusqu'à une sauvegarde.
- `<numéro>_<id utilisateur>.b` : contient les nouvelles définitions des opérations d'une classe jusqu'à une sauvegarde.
- `<user_id>.lock` : un répertoire créé lorsque vous chargez le projet et supprimé lorsque vous fermez le projet.

*** : Le fichier `<id utilisateur>.session` peut être placé dans le dépôt (notamment si vous travailler sur plusieurs postes) MAIS il doit être propre à chaque utilisateur.**

Remarque : Vous pouvez indiquer à SVN d'ignorer des fichiers et/ou des répertoires. Pour cela, vous pouvez créer un fichier `.svnignore` dans lequel vous indiquer chaque fichier et/ou répertoire (un par ligne) que SVN doit ignorer. Ensuite, vous appliquer les commandes suivantes dans le répertoire concerné (`.` désigne le répertoire courant) :

```
$ vim .svnignore

$ svn propset svn:ignore -F .svnignore .

$ rm .svnignore

$ svn commit --message "Fichiers/répertoires à ignorer"

$ svn proplist -v
```

Conseils

1 . **Utiliser des numéros de session différents pour chaque utilisateur**.

2 . **Il est toujours plus prudent de commiter lorsque le projet a été fermé dans Bouml** (cf. les fichiers `.d` et `.b`).

3 . Lorsque vous créez des éléments dans Bouml (par exemple des diagrammes), il faudra **les ajouter au dépôt SVN** :

```
$ svn status
?       134551.diagram

$ svn add 134551.diagram
A       134551.diagram

$ svn commit ...
```

Sinon le diagramme sera bien associé au *package* de votre projet Bouml MAIS il sera vide :

```
$ cat 128002
```

```
...
     classdiagram 134551 "diagramme_classes"
      classdiagramsettings member_max_width 0 end
      size A2
    end
...
```

4 . Même si les fichiers d'un projet Bouml sont au format ASCII, il sera peut-être pénible de résoudre des conflits détectés par subversion (lors d'un `commit` / `update`). Dans cas, il est plus prudent de **verrouiller** (`svn lock`) le dossier du projet Bouml et le **déverrouiller** (`svn unlock`) une fois son travail terminé.

5 . **Mettre les chemins vers le code source en relatif** (voi ci-dessous).

Reverse/Roundtrip

L'utilisation du *Reverse/Roundtrip* a été vue dans le tp-poo-bouml.pdf.

Rappel : En développement orienté objet avec UML, le *Reverse* consiste à générer des diagrammes UML à partir d'un code source.

En projet, on réalisera l'opération *Reverse* une seule fois. Ensuite, on utilise *Roundtrip*.

Rappel : Le développement *Roundtrip* consiste à faire des aller-retour entre le modèle UML et le code source. L'objectif est de conserver une cohérence entre la modélisation UML et le code source et donc de propager les modifications de l'un vers l'autre.

Dans Bouml, le *Roundtrip* est l'opération qui synchronise les éléments UML à partir du code source.

Attention : En projet, il ne faut jamais utiliser l'opération `Générer` (ou éventuellement au tout début de projet) car celle-ci écrasera votre code source.

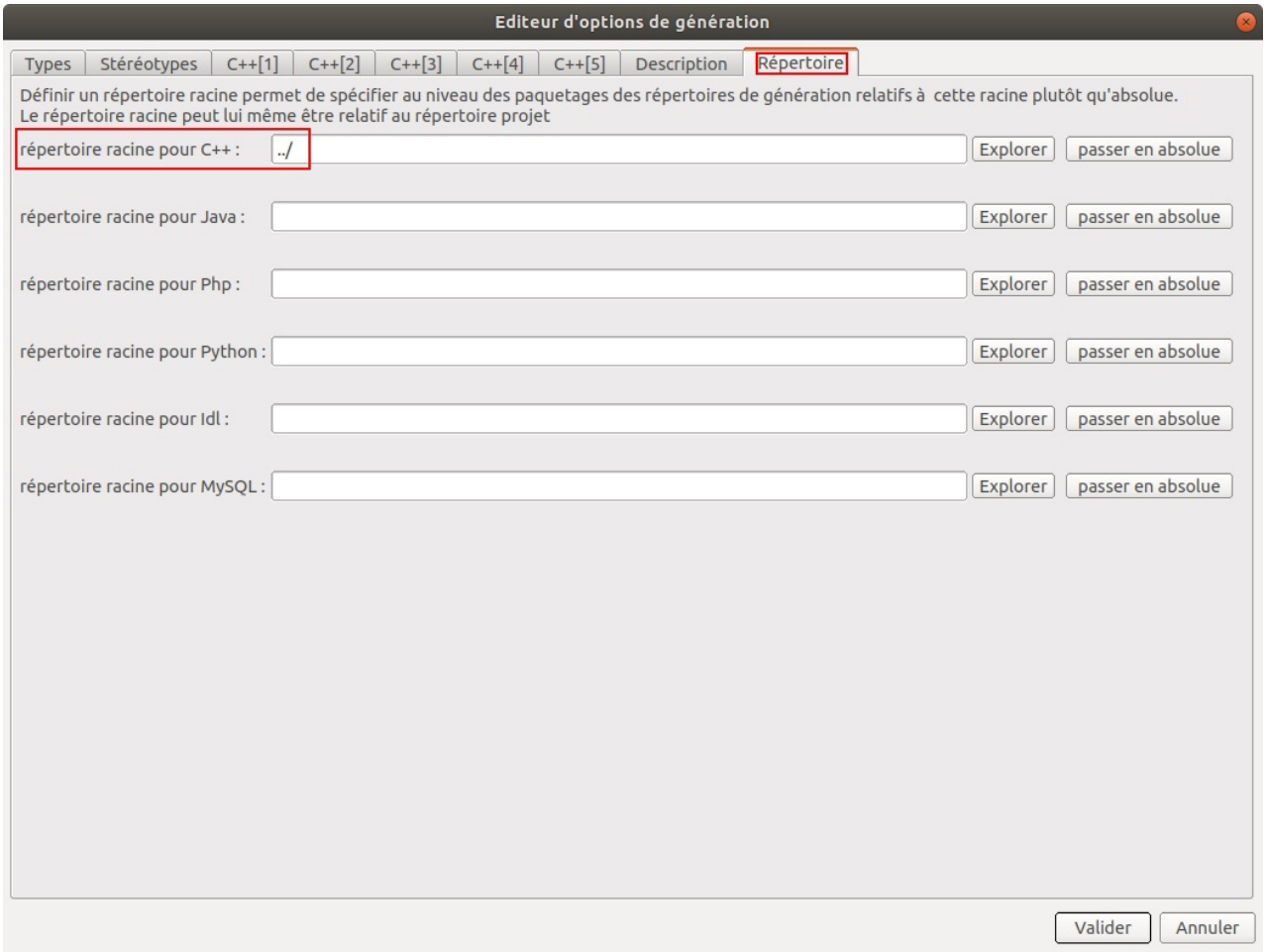
Vous allez réaliser très souvent des opérations de *Roundtrip* pour mettre à jour vos éléments UML. Pour cela, Bouml doit connaître l'emplacement de votre code source. Vous devez le configurer pour une utilisation à partir de **chemin en relatif** et jamais en absolu (qui est le choix par défaut de Bouml) !

La **racine** de votre code source est une information qui est stockée dans le fichier `<nom du projet>.prj` :

```
cpp_root_dir "../"
```

Rappel : Le répertoire `.` désigne le répertoire courant (ici celui qui contient le fichier `<nom du projet>.prj` et le répertoire `..` désigne le répertoire parent (par exemple votre dossier `trunk`).

Vous pouvez modifier la **racine** directement dans **Bouml** en allant dans `Projet` ⟶ `Editer` ⟶ `Editer les options de génération` :

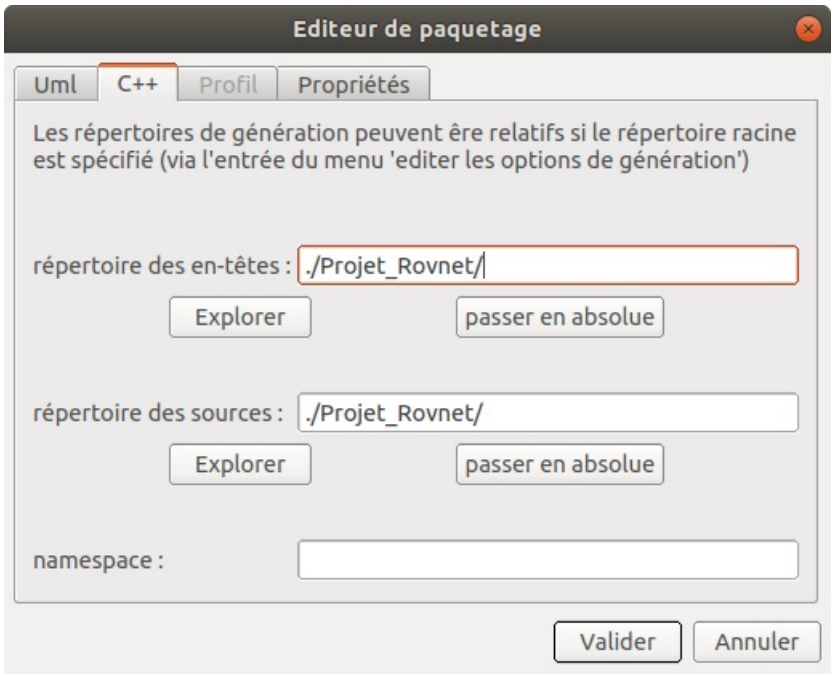


Chaque *package* renseigne ensuite ses répertoires pour les fichiers *headers* (`.h`) et sources (`.cpp`) dans les fichiers `<numéro>` :

```
cpp_h_dir "./Projet_Rovnet/"
cpp_src_dir "./Projet_Rovnet/"
```

Remarque : Bouml permet donc de séparer dans des dossiers différents les fichiers *headers* (`.h`) et sources (`.cpp`).

Vous pouvez modifier ces chemins directement dans **Bouml** en faisant un clic droit sur le *package* concerné :



Remarque : Bouml permet de regrouper dans le même projet Bouml plusieurs “applications” en les affectant dans des *packages* séparés.

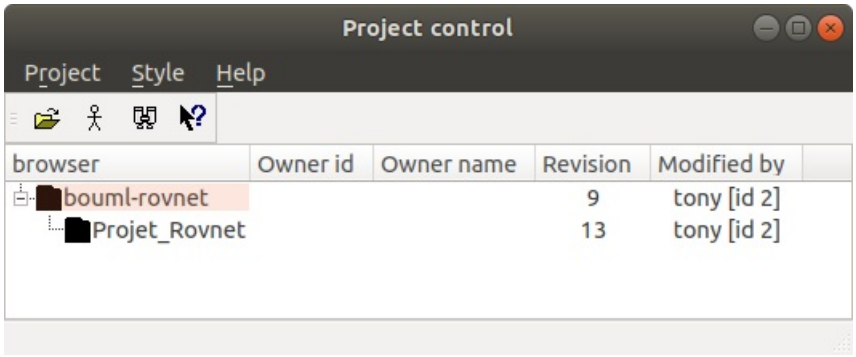
Au final, les deux options s’ajoutent de cette façon : `cpp_root_dir` + `cpp_src_dir` et `cpp_root_dir` + `cpp_h_dir` .

Remarque : Evidemment, même chose pour Java.

Outils Bouml

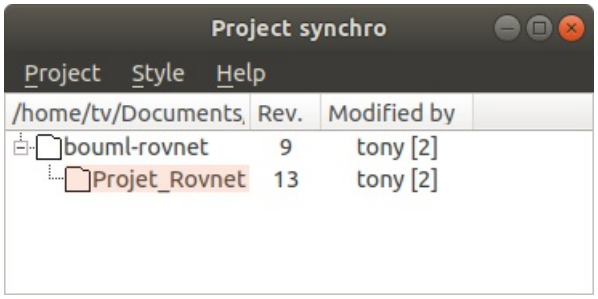
Bouml fournit des outils (que l’on utilise pas) pour une utilisation multi-utilisateurs :

- `File control` est un *plug-in* principalement dédié à être une interface entre BOUML et un contrôle de fichiers comme Clearcase, CVS ou **Subversion** etc …, cependant, l’utilisation d’un contrôle de fichiers peut affecter l’autorisation d’écriture des fichiers. Documentation : File control
- `Project control` est un outil gérant une autorisation d’écriture applicative éventuellement mémorisée dans les fichiers de projet associés aux packages. Cela permet de définir l’autorisation d’écriture indépendamment du système d’exploitation que vous utilisez. Documentation : Project control



- `Project Synchro` est un outil qui synchronise plusieurs images d’un projet développé en parallèle par plusieurs utilisateurs. La synchronisation se fait en remplaçant les anciens fichiers par de nouveaux. Notez

que les fichiers sont copiés et non fusionnés. Documentation : Project Synchro



Retour au sommaire