

PRÉSENTATION DE BOUML

par Romain Bouleis

TABLES DES MATIÈRES

TABLES DES MATIÈRES.....	2
POURQUOI PRÉSENTER BOUML ?.....	3
Critères de choix.....	3
Fonctionnalités et spécifications.....	3
CONCEPTION DE SYSTÈMES.....	4
Introduction.....	4
Premier projet.....	5
Diagramme de classes.....	5
Diagramme de séquences.....	6
GÉNÉRATION DE CODE.....	8
Introduction.....	8
Les (petits) problèmes.....	8
Personnalisation.....	9
L'export en JAVA.....	10
L'export en C++.....	12
REVERSE ENGINEERING.....	14
AUTRES FONCTIONNALITES.....	17
Génération de documentation HTML.....	17
Java Catalog.....	18
CONCLUSION ET AVIS PERSONNEL.....	19
TABLE DES ILLUSTRATIONS.....	20
TABLE DES SOURCES.....	20

POURQUOI PRÉSENTER BOUML ?

CRITÈRES DE CHOIX

Atelier de génie logiciel (AGL) est un terme très vague : il peut faire référence à n'importe quel programme entrant dans les phases de conception, développement, maintenance d'un logiciel. Dans le cadre de cette présentation, l'AGL présenté devait permettre de réaliser la phase de conception. Ceci fût mon premier critère de choix.

Novice en conception de systèmes, je n'avais pas de critères précis pour choisir un AGL ; j'en ai donc défini deux sans aucun rapport avec la conception en elle même :

- l'AGL devait être gratuit : de nombreux programmes libres ou gratuits sont équivalents, voir meilleurs, que leurs homologues payants. Pourquoi pas dans le monde de la conception ?
- l'AGL devait fonctionner sous Linux. En effet, j'utilise ce système d'exploitation pour la majorité des développements que j'ai à effectuer, il aurait été dommage que la partie conception de ceux ci soit faite sous un autre système.

Ces deux critères furent très judicieux puisque mon choix fût considérablement réduit ; se limitant à trois ou quatre programmes différents. J'ai alors choisi celui dont les spécifications paraissaient le mieux répondre à mes préférences... : BOUML.

BOUML est disponible sur le site suivant : bouml.free.fr.

FONCTIONNALITÉS ET SPÉCIFICATIONS

BOUML est un modelleur UML 2 libre. Il répond aux spécifications d'UML 2. Il est encore en cours de développement ce qui amène quelques remarques :

- Certains bugs mineurs sont présents.
- Ils sont néanmoins vite corrigés : selon l'historique présent sur le site, depuis la version 3.0 sortie le 7 octobre 2007, il y a eu huit nouvelles versions, pour parvenir à celle que j'utilise aujourd'hui : 3.2.2.
- Il évolue au gré de l'évolution de la norme UML 2.

BOUML est multi-plateformes : il fonctionne sous Linux, Unix, Solaris, Mac OS X et Windows.

Il permet de générer du code dans les langages suivants : C++, Java, PHP et Idl. Tout est configurable, de la définition de la class au prototypes des fonctions.

Des « plug-outs » (nommés ainsi car ils sont exécutés hors de BOUML) peuvent être écrits pour ajouter des fonctionnalités.

Rapide et léger : bien que n'ayant pas une grande expérience des AGLs de conception (Poséidon et Eclipse), mes précédents tests m'avaient tous ammené à la conclusion suivante : ces programmes sont vraiment très lents. BOUML ne l'est pas du tout ; son utilisation est très fluide. Des benchmarks sont disponibles sur le site.

Bruno Pagès, le développeur de BOUML est très actif. Il a répondu avec précision à un mail de ma part lui signalant un petit bug en moins de douze heures !

CONCEPTION DE SYSTÈMES

INTRODUCTION

UML 2 définit treize types de diagramme différents, BOUML permet de modéliser neuf d'entre eux :

1. Diagramme de classes [x],
2. Diagramme d'objets [x],
3. Diagramme de composants [x],
4. Diagramme de déploiement [x],
5. Diagramme de package,
6. Diagramme de structures composites,
7. Diagramme de cas d'utilisation [x],
8. Diagramme d'état-transitions [x],
9. Diagramme d'activités [x],
10. Diagramme de séquences [x],
11. Diagramme de communication [x],
12. Diagramme global d'interaction,
13. Diagramme de temps.

Afin de réaliser ce guide, et par la même occasion, modéliser un programme que j'étais actuellement en train de développer, j'ai utilisé deux de ces diagrammes ; le diagramme de classes et le diagramme de séquences. En outre, ce sont les modèles les plus utilisés avec UML.

Je vais donc tout d'abord présenter l'interface du logiciel et la création d'un nouveau projet ; après quoi, je développerai les deux diagrammes dont je me suis servi.

PREMIER PROJET

Le logiciel lancé, la création d'un nouveau projet se fait grâce au menu *Project, New*. La capture d'écran suivante montre BOUML une fois un nouveau projet créé :

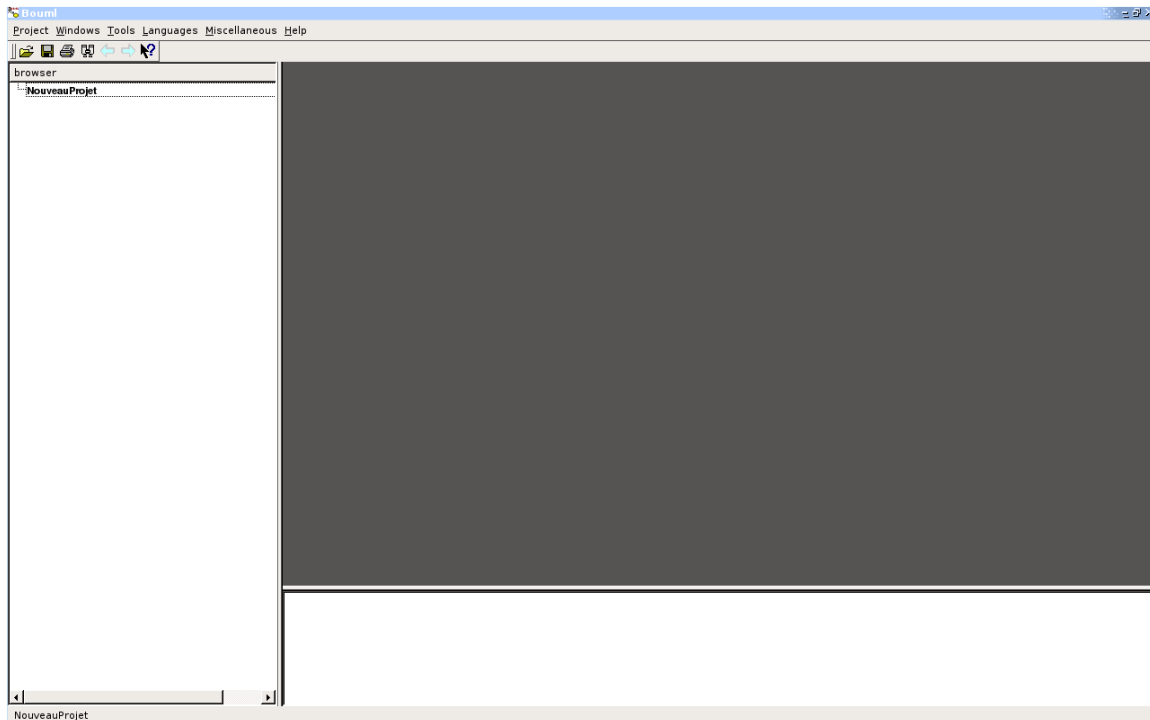


Illustration 1: Nouveau projet

La fenêtre est clairement divisée en trois parties :

- Une partie gauche : le browser. Il permet de naviguer dans les différentes vues du projet. Ces vues permettront d'accéder aux différents diagrammes, objets, acteurs, etc..., du projet.
- La partie centrale : c'est la zone de dessin qui permettra d'éditer les différents diagrammes. Il est possible d'y effectuer des « glisser déposer » depuis le browser.
- La partie basse : associée à une sélection dans la zone de dessin ou dans le browser, elle permet de commenter n'importe quel élément du projet (projet, vue, objet, classe, méthode, etc...). Associé à un élément pouvant donner lieu à une génération de code, le texte édité dans cette zone apparaîtra comme commentaire dans le fichier source.

Remarque : il n'est possible d'ouvrir qu'un projet à la fois par instance de BOUML.

DIAGRAMME DE CLASSES

Pour créer un diagramme de classes, il faut d'abord créer une « class view ». Cette vue contiendra les classes, le ou les diagrammes qui leurs sont associés, ainsi que d'autres types de diagrammes si désiré. C'est une manière de hiérarchiser les modèles de conception, ce qui, au premier abord, est assez confu, mais qui par la suite, s'avère réellement utile.

Pour créer une « class view », cliquer droit sur le projet dans le browser, puis *new class view*. Lui donner un nom, et lui ajouter un diagramme de classes grâce à un clic droit sur na nouvelle vue, puis *new class diagram*.

Un double clic sur le nouvel élément ouvrira alors le nouveau diagramme dans la zone de dessin.

Voici un exemple de diagramme obtenu :

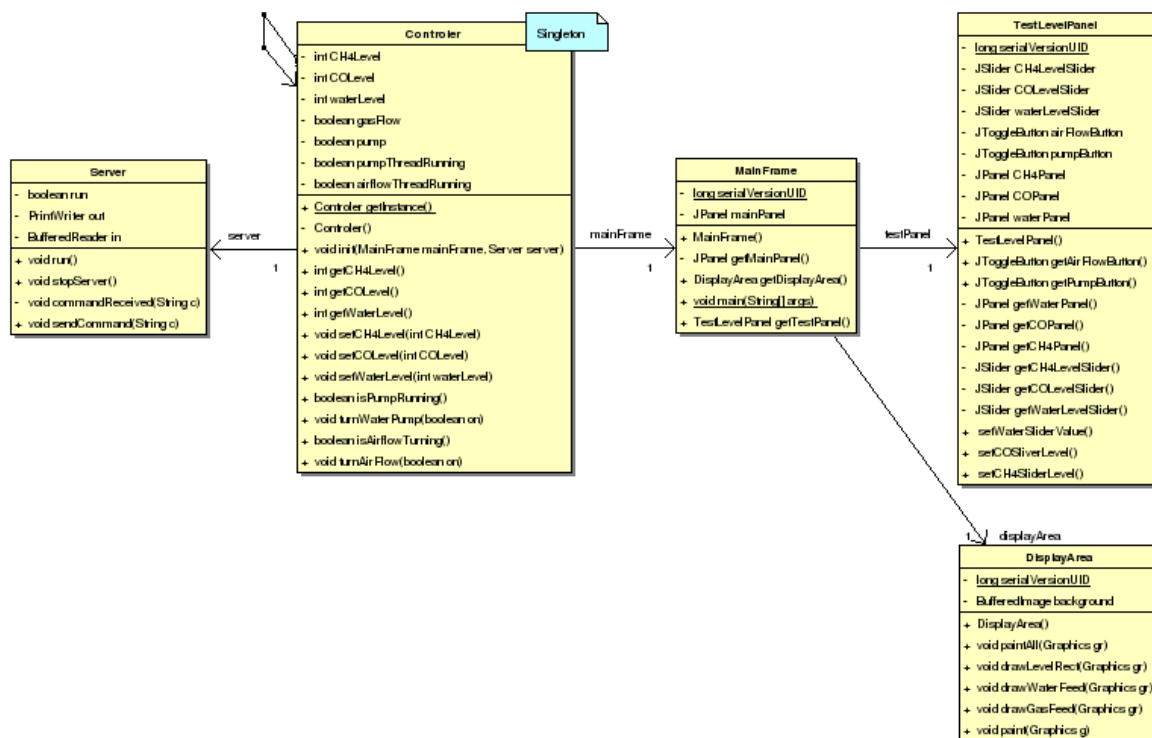


Illustration 2: Diagramme de classes

La prise en main de l'outil de dessin est assez simple et ne requiert pas d'explication particulière.

De multiples raccourcis existent. Par exemple un clic droit sur un attribut nouvellement créé permettra en un clic de générer ses accesseurs.

DIAGRAMME DE SÉQUENCES

Le diagramme de séquence doit être créé dans une « use case view ». Contrairement à la « class view », celle-ci pourra contenir, en plus du diagramme de séquences et des objets associés, des acteurs. Cette vue se crée aussi grâce à un clic droit sur le projet. Le diagramme de séquences et l'acteur se créent eux avec un clic droit sur la vue nouvellement ajoutée.

Ici encore, la mise en forme des composants du diagramme se fait simplement. Attention toutefois à ne pas faire n'importe quoi ; par exemple, l'ajout d'un message récursif à un endroit inapproprié provoque la fermeture du programme. Il s'agit d'un bug, sans conséquence lorsque l'on sait où placer ces flèches, ce qui est supposé être le cas (n'allez pas penser que j'ai découvert ce bug car je ne savais pas où placer mes flèches, ma souris a simplement fourché...).

Je n'ai pas recensé d'autres anomalies du genre.

Voici un diagramme réalisé :

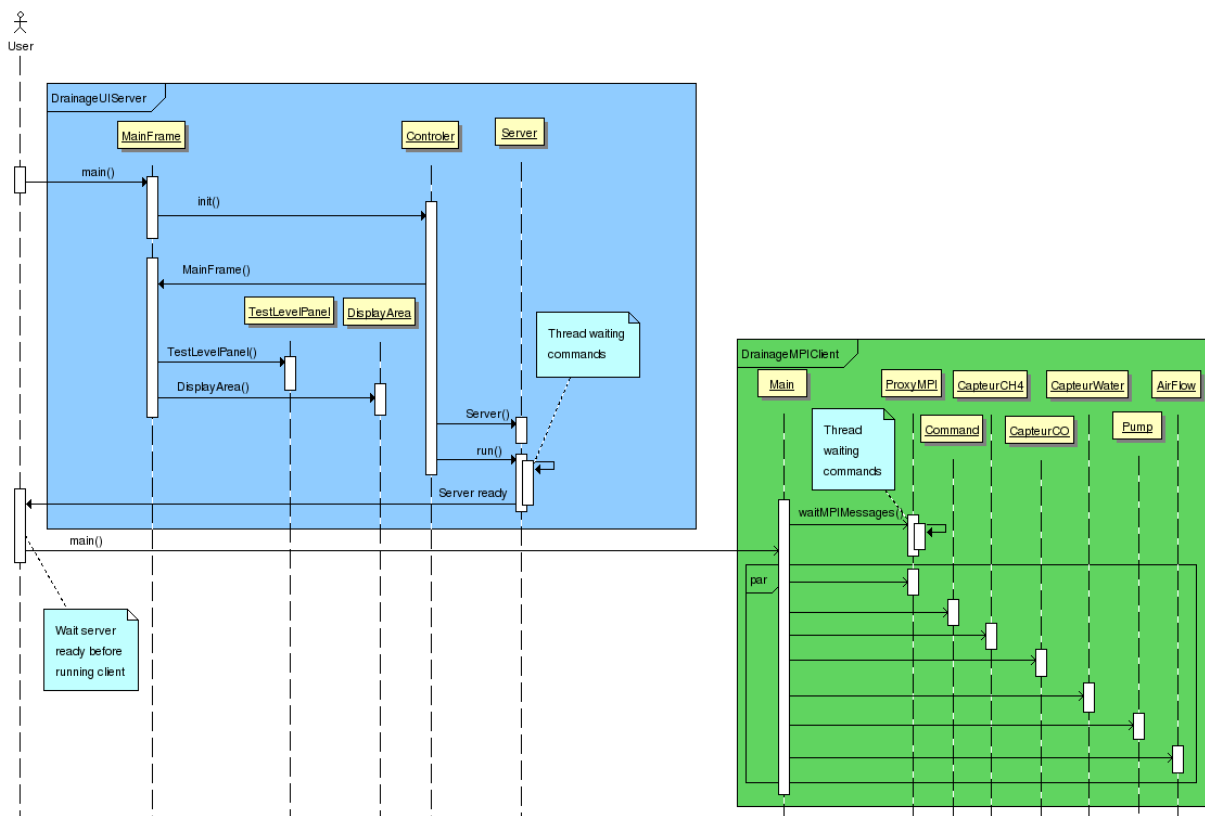


Illustration 3: Diagramme de séquences

BOUML facilite vraiment l'édition de ces diagrammes. Par exemple, lors de la création d'un nouveau message, le logiciel proposera automatiquement les méthodes publiques de l'objet receveur. Ou encore, les nom d'opérateurs tel que « par » seront proposés lors de l'ajout d'opérateurs.

GÉNÉRATION DE CODE

INTRODUCTION

BOUML permet de générer un squelette de code à partir des différents diagrammes dessinés, l'idéal étant un diagramme de classes. Si cette fonctionnalité est aujourd'hui présente dans la majorité des AGLs dite de conception, elle n'en reste pas moins très appréciable.

Néanmoins, si le résultat est au de-là de ce que j'espérais, la difficulté à l'obtenir l'est aussi. En effet, je n'ai pas trouvé particulièrement évident le moyen de générer du code à partir d'un diagramme. Cependant, après avoir lu la documentation, cela s'est éclairci.

Pour générer du code, la première opération consiste à définir le ou les langages dans lesquels seront générés le code. Ceci se fait via le menu *Languages*. Après quoi, il faut créer une nouvelle vue : « deployment view ». Cette vue a pour but de rassembler les entités servant au déploiement de l'application, donc aussi les fichiers sources. On peut aussi y intégrer des diagrammes de déploiement. Là encore, la vue se crée grâce à un clic droit sur le projet dans le browser. Après quoi, il faut associer cette vue à la « class view » où se trouvent vos classes à exporter. Pour se faire, un double clic sur la « class view » permet de l'éditer et de changer la valeur du champ « deployment view ». Il faut maintenant associer à chaque classe un « artifact ». Dans le cas d'une classe, il s'agit du fichier qui la contiendra. Ceci se fait grâce à un clic droit sur la classe puis, « Create source artifact ». La dernière opération consiste à désigner le repertoire où seront créés les fichiers. Clic droit sur le projet, *Edit generating settings, Directory*, puis éditer les champs nécessaires. Enfin, un clic droit sur le projet suivi de *Generate, language* générera le code.

On se rend compte que cette procédure est relativement lourde.

Peu importe ! Maintenant que l'on peut se passer d'écrire des « public static final int String..... », autant en profiter !

LES (PETITS) PROBLÈMES

Prenons un exemple simple :

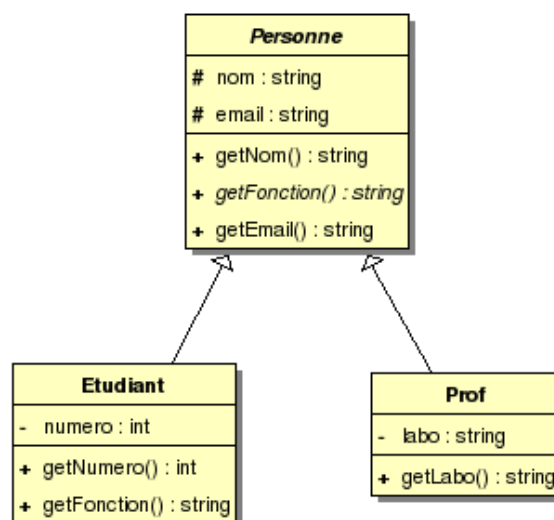


Illustration 4: Diagramme de classes pour génération

Dans cet exemple, *Person* est une classe abstraite et possède une méthode abstraite *getFunction()* : *string* qui, selon la définition même d'une méthode abstraite, devra impérativement être réécrite par les classes filles. *getFunction()* est bien réécrite dans la classe *Etudiant* mais pas dans la classe *Prof*. Il en sera de même dans le code généré. Ceci est, selon moi, un manque. En effet, il serait tout à fait convenable que BOUML crée lui-même ces méthodes dans les classes filles, et par conséquent, les génère. Il existe bien un raccourci ; un menu accessible via un clic droit sur une classe fille permettra d'ajouter automatiquement n'importe quelle méthode de la classe mère. Pour une méthode abstraite, ceci devrait être automatique.

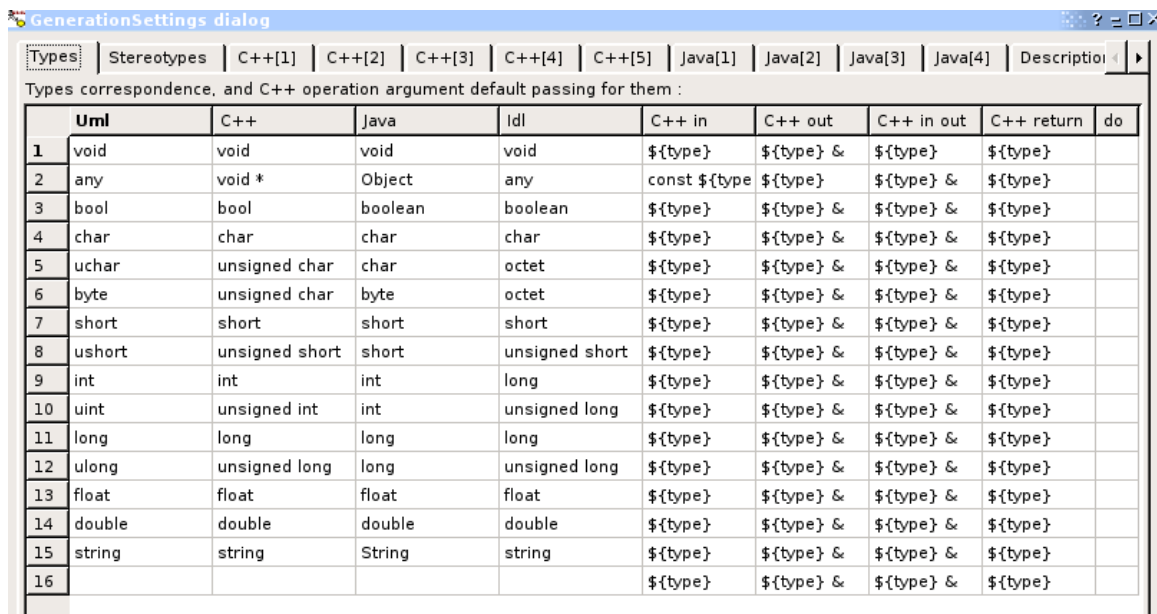
Autre petit manque ; pas de « return » automatique pour les types connus. En effet, aucun corps de fonction n'est généré ce qui aurait pu être le cas pour les fonctions. Par exemple *getNumero()* : *int* pourrait avoir un corps automatique qui serait alors *return 0*. Certes il faudrait le modifier par la suite, mais le code généré par BOUML serait alors directement compilable. Néanmoins, si les accesseurs ont été créés automatiquement, le corps sera généré.

Certes, je « cherche les petites bêtes », mais les défauts de cet AGL sont si rares que pour entretenir mon objectivité, je suis obligé...

PERSONNALISATION

Un atout non négligeable est de pouvoir configurer de manière très précise la façon dont les sources sont générées. Ainsi, si l'on souhaite que le mot clé facultatif en Java précède une classe ne soit pas généré, il est possible de l'enlever. De même, si l'on souhaite que le type « list » UML soit représenté par un « ArrayList » et non un « Vector » (Java) comme c'est le cas par défaut, il suffit de modifier la configuration.

Un clic droit sur le projet nous permet d'accéder au menu *Edit generation settings* et ainsi d'accéder à la fenêtre suivante :



The screenshot shows a window titled "GenerationSettings dialog" with a tabbed interface. The "Types" tab is selected, displaying a table titled "Types correspondence, and C++ operation argument default passing for them :". The table has columns for Uml, C++, Java, Idl, C++ in, C++ out, C++ in out, C++ return, and do. It lists 16 rows of type mappings.

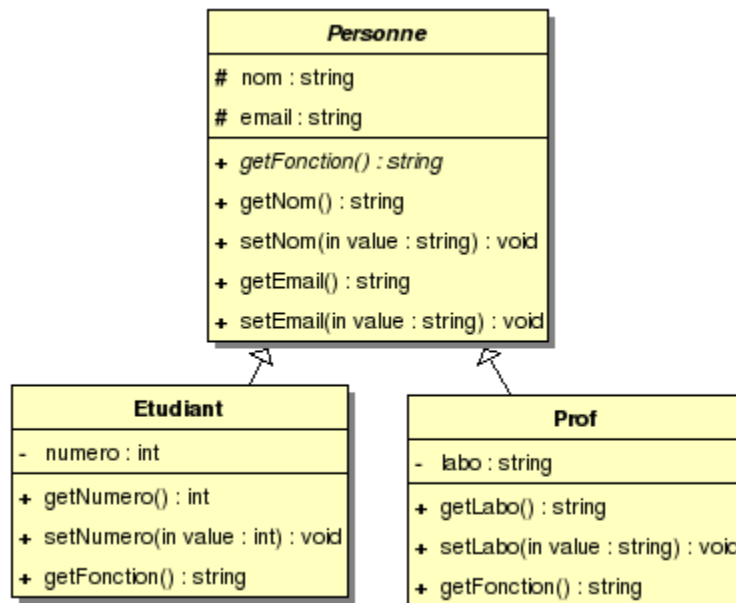
	Uml	C++	Java	Idl	C++ in	C++ out	C++ in out	C++ return	do
1	void	void	void	void	#{type}	#{type} &	#{type}	#{type}	
2	any	void *	Object	any	const #{type}	#{type}	#{type} &	#{type}	
3	bool	bool	boolean	boolean	#{type}	#{type} &	#{type} &	#{type}	
4	char	char	char	char	#{type}	#{type} &	#{type} &	#{type}	
5	uchar	unsigned char	char	octet	#{type}	#{type} &	#{type} &	#{type}	
6	byte	unsigned char	byte	octet	#{type}	#{type} &	#{type} &	#{type}	
7	short	short	short	short	#{type}	#{type} &	#{type} &	#{type}	
8	ushort	unsigned short	short	unsigned short	#{type}	#{type} &	#{type} &	#{type}	
9	int	int	int	long	#{type}	#{type} &	#{type} &	#{type}	
10	uint	unsigned int	int	unsigned long	#{type}	#{type} &	#{type} &	#{type}	
11	long	long	long	long	#{type}	#{type} &	#{type} &	#{type}	
12	ulong	unsigned long	long	unsigned long	#{type}	#{type} &	#{type} &	#{type}	
13	float	float	float	float	#{type}	#{type} &	#{type} &	#{type}	
14	double	double	double	double	#{type}	#{type} &	#{type} &	#{type}	
15	string	string	String	string	#{type}	#{type} &	#{type} &	#{type}	
16					#{type}	#{type} &	#{type} &	#{type}	

Illustration 5: Paramétrage de la génération

Ce tableau est complètement éditable. On peut donc remplacer les types de base par des types personnalisés lors de la génération. Un programmeur Java, préférant les « StringBuffer » aux simples « String » aura ainsi un outil tout à fait adapté.

L'EXPORT EN JAVA

Reprenons l'exemple précédent, en ayant créé tous les accesseurs automatiquement et commenté les éléments grâce à l'éditeur. Voici le nouveau diagramme UML :



Après génération, voici donc le code JAVA des trois classes :

```
/**
 * Classe representant une personne
 */
abstract class Personne {
    protected String nom;

    protected String email;

    /**
     * methode abstraite
     */
    public abstract String getFonction() ;
    /**
     * retourne le nom de la personne
     */
    public String getNom() {
        return nom;
    }

    /**
     * set le nom de la personne
     */
    public void setNom(String value) {
        nom = value;
    }

    /**
     * retourne l'email de la personne
     */
    public String getEmail() {
        return email;
    }

    /**
     * set l'email de la personne
     */
    public void setEmail(String value) {
        email = value;
    }
}

/**
 * Classe representant un etudiant
 */
class Etudiant extends Personne {
    private int numero;

    /**
     * retourne le numero de l'etudiant
     */
    public int getNumero() {
        return numero;
    }

    /**
     * set le numero de l'etudiant
     */
    public void setNumero(int value) {
        numero = value;
    }

    /**
     * methode abstraite
     */
    public String getFonction() {
    }
}
```

Source 1: Person.java et Etudiant.java

```

/**
 * Classe representant un prof
 */
class Prof extends Personne {
    private String labo;

    /**
     * retourne le labo du professeur
     */
    public String getLabo() {
        return labo;
    }

    /**
     * set le labo du professeur
     */
    public void setLabo(String value) {
        labo = value;
    }

    /**
     * methode abstraite
     */
    public String getFonction() {
    }
}

```

Source 2: Prof.java

Il ne reste plus qu'à remplir le contenu de deux méthodes ! les commentaires sont idéalement placés ! Je ne m'attendais pas à un tel résultat. Les corps des fonctions *getFonction()* renseignés, la compilation s'est déroulée sans même un warning.

Ce résultat associé au fait que BOUML permet d'écrire le corps des fonctions, je pense que la phase de conception, qui pouvait être vue comme une perte de temps, peut finalement en faire gagner.

L'EXPORT EN C++

En reprenant exactement le même diagramme de classes que pour la génération en Java, voici le résultat en C++ (class.h | class.cpp) :

<pre> #ifndef _PERSONNE_H #define _PERSONNE_H #include <string> using namespace std; //Classe representant une personne class Personne { protected: string nom; string email; public: //methode abstraite virtual string getFonction() = 0; //retourne le nom de la personne inline const string get_nom() const; //set le nom de la personne void set_nom(string value); //retourne l'email de la personne inline const string get_email() const; //set l'email de la personne void set_email(string value); }; //retourne le nom de la personne inline const string Personne::get_nom() const { return nom; } //retourne l'email de la personne inline const string Personne::get_email() const { return email; } #endif </pre>	<pre> #include "Personne.h" //set le nom de la personne void Personne::set_nom(string value) { nom = value; } //set l'email de la personne void Personne::set_email(string value) { email = value; } </pre>
---	---

Source 3: Person.h et Person.cpp

```
#ifndef _ETUDIANT_H
#define _ETUDIANT_H

#include "Personne.h"
#include <string>
using namespace std;

//Classe representant un etudiant
class Etudiant : public Personne {
private:
    int numero;

public:
    //retourne le numero de l'etudiant
    inline const int get_numero() const;

    //set le numero de l'etudiant
    void set_numero(int value);

    //methode abstraite
    virtual string getFonction();
};

//retourne le numero de l'etudiant
inline const int Etudiant::get_numero()
const {
    return numero;
}

#endif

#include "Etudiant.h"

//set le numero de l'etudiant
void Etudiant::set_numero(int value) {
    numero = value;
}

//methode abstraite
string Etudiant::getFonction() {
}
```

Source 4: Etudiant.h et Etudiant.cpp

Les sources de la classe Prof étant sensiblement les même que celles de la classe Etudiant, je ne les ai pas retranscrites.

Ici encore, le résultat est très satisfaisant. BOUML gère tout ; des « include » aux « namespace ».

Toutes les relations sont bien placées et rien ne manque. Même surchargé, BOUML reste toujours aussi fluide.

Lors d'essais précédents, c'est la première fonctionnalité que j'avais voulu tester ; je l'avais donc essayée sur un programme en cours de développement. Cela m'avait permis de me rendre compte de petites erreurs de conceptions que je n'aurais probablement pas vu sinon. Cela montre que cette utilisation peut être utile.

AUTRES FONCTIONNALITES

Dans cette dernière partie, je vais brièvement faire état de différentes fonctionnalités périphériques à la conception qu'offre BOUML.

GÉNÉRATION DE DOCUMENTATION HTML

Un peu à la manière d'une javadoc, BOUML permet de générer une documentation HTML de ses classes, méthodes et diagrammes. Voici une capture d'écran du résultat obtenu avec l'exemple utilisé précédemment (Etudiant, Prof) :

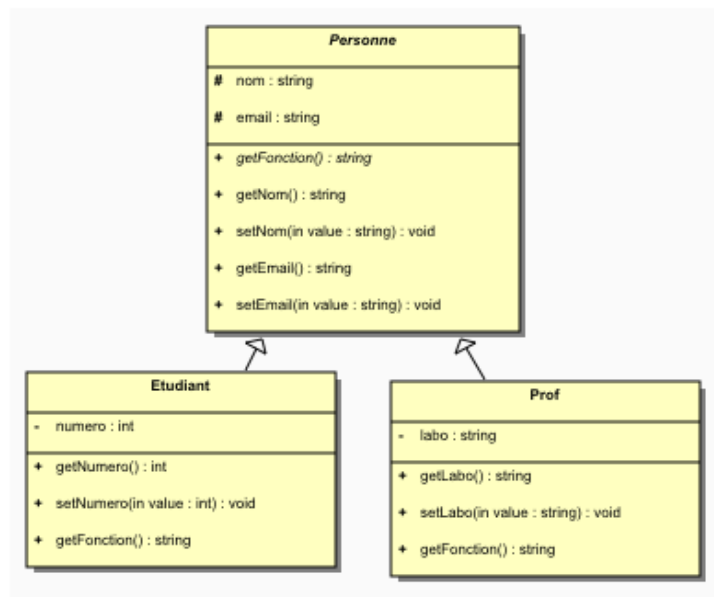
nouveauprojet Documentation

Properties:

- html dir:
/home/uself/AccessoiresLinux/Fac/Master1/AGL/nouveauprojet/

1 Package nouveauPackage

2 Class View classView



classDiagram

Illustration 8: Apperçu 1 documentation

2.1 Class Personne

Classe representant une personne

Declaration :

- C++ : class Personne
- Java : package abstract class Personne

Directly inherited by : [Etudiant Prof](#)

Artifact : [Personne](#)

Attribut **nom**

Declaration :

- Uml : # nom : string
- C++ : protected: string nom
- Java : protected String nom

Attribut **email**

Declaration :

- Uml : # email : string
- C++ : protected: string email
- Java : protected String email

Operation **getFonction**

methode abstraite

Illustration 9: Aperçu 2 documentation

Cette documentation peut être très pratique à diverses étapes de la conception, voir même après.

JAVA CATALOG

Déstiné aux développeurs Java, cette fonctionnalité permet de faire un catalogue de classes Java. L'utilisation la plus commune consiste à intégrer les bibliothèques systèmes de Java de manière à y avoir accès à travers les différents projets personnels. Par exemple, lors de l'utilisation de la classe « Jpanel » de Swing, si celle-ci se trouve dans le catalogue, alors BOUML pourra l'intégrer dans un diagramme de classes ou un objet y aurait une relation. Cela permet aussi de rechercher des classes, d'avoir leurs documentations (commentaires) ou encore d'obtenir la liste de leurs méthodes et attributs.

CONCLUSION ET AVIS PERSONNEL

Au cours de ce test des fonctionnalités de BOUML, je dois dire que je suis vraiment étonné de la qualité de ce logiciel. Il ne permet de modéliser que la méthode UML mais aujourd'hui, il semble que celle-ci soit la plus en vogue. De plus, ses différents diagrammes en font une méthode très complète, je ne peux donc reprocher à BOUML de ne pas être compatible avec Merise par exemple.

Ce logiciel est surprenant, lors du premier lancement, il donne l'impression d'être un modèleur, ni plus, ni moins, et pourtant je pense avoir montré qu'il ne s'y limitait pas. De plus, je n'ai parlé que de deux représentations, mais BOUML en couvre neuf ; il y a donc encore beaucoup à dire. Je n'aborde pas non plus l'aspect des « plug-out » car je n'ai pas le recul nécessaire pour envisager de nouvelles utilités, même si j'en suis sûr, elles existent.

La qualité du générateur de code nous pourrait faire penser qu'il s'agit d'un outil payant. En effet, « avant », je n'envisageais de conception que lors de programmes conséquents. Désormais, la puissance et la qualité des fonctionnalités remettent en cause cette mauvaise habitude. En effet, le générateur de code fait gagner en temps ce que la conception en elle-même pouvait faire perdre en aval du développement.

Si je devais reprocher une chose à ce logiciel, ce serait son manque d'intuitivité. Est réellement possible d'avoir une interface simple et permettant autant de choses complexes à la fois ? Je ne sais pas. La documentation est très bien faite et a répondu à tous mes problèmes ; mais j'ai dû la consulter à plusieurs reprises. De plus, lors d'éventuelles discussions avec des collègues, quand je leur disais « bouml, c'est génial », on m'a répondu à chaque fois (2), « j'ai essayé mais ça ne marche pas ». Certes ils n'avaient pas vraiment persisté dans leurs essais, mais je trouve que cela illustre assez bien la difficulté de prise en main de ce logiciel. Néanmoins, la simplicité n'est clairement pas le but de BOUML qui a pour objectifs sa complétude et sa légèreté (dans le sens occupation mémoire / processeur, rapidité de traitement).

Enfin, comme je l'ai dit, j'ai utilisé la version 3.3.2 et j'ai signalé qu'un bug était présent dans les diagrammes de séquence. Mais aujourd'hui (21/11/2007), une nouvelle version de BOUML est sortie : la 3.3.3. Voici son descriptif (<http://bouml.free.fr>) :

Previous releases crash when you add a reflexive message on a life line in a sequence diagram (this means not on a duration bar), fixed.

Cela rejoint complètement mes propos, où je faisais remarquer que le développeur était très actif et qu'un bug était très vite corrigé ce qui montre aussi la qualité et le sérieux du développement.

Evidemment, je ne saurais trop conseiller de réserver un peu de temps pour prendre en main cet AGL.

TABLE DES ILLUSTRATIONS

Illustration 1: Nouveau projet.....	4
Illustration 2: Diagramme de classes.....	5
Illustration 3: Diagramme de séquences.....	6
Illustration 4: Diagramme de classes pour génération.....	7
Illustration 5: Paramétrage de la génération.....	8
Illustration 6: Reverse Java.....	13
Illustration 7: Reverse : Diagramme de classes.....	13
Illustration 8: Aperçu 1 documentation.....	15
Illustration 9: Aperçu 2 documentation.....	16

TABLE DES SOURCES

Source 1: Person.java et Etudiant.java.....	9
Source 2: Prof.java.....	10
Source 3: Person.h et Person.cpp.....	11
Source 4: Etudiant.h et Etudiant.cpp.....	12