

Rapport de projet

Projet n°4

Génération automatique
de codes sources JAVA et d'un formalisme TUOPA d'un type abstrait.

Version	Auteur	État	Responsable du projet
1.0	Gouchet Hervé Leray Jérôme	En cours	Gouchet Hervé Leray Jérôme

SOMMAIRE

I- Objectif du projet.....	3
II- Rappel	
2.1 - Caractéristiques d'un type abstrait (TUOPA)	4
2.2 - Caractéristiques d'une interface JAVA	4
2.3 - Implémentation d'une classe JAVA	5
III- Répartition des rôles.....	6
IV- Choix techniques	7
4.1 - Choix du formalisme du type abstrait	7
4.2 - Organisation du format XML.....	7
4.2.1 - Les éléments du TUOPA	7
4.2.2 - La DTD	9
4.3 - L'interface graphique	12
V- Organisation du produit.....	13
5.1 - Les classes, objet et méthodes	13
5.2 - L'analyse TUOPA vers JAVA	13
5.2.1 - Analyse du fichier XML	13
5.2.2 - La restitution en langage JAVA.....	14
5.3 - L'analyse JAVA vers TUOPA	16
5.3.1 - Introspection.....	16
5.3.2 - Analyse du fichier JAVA.....	17
5.3.3 - Récupération des éléments pour le TUOPA.....	18
5.3.4 - Construction du fichier XML représentant le type abstrait	18
VI- Conclusion.....	19
VII- Bibliothèque	20
VIII- Annexes	

1. OBJECTIF DU PROJET

L'objectif de ce projet est de réaliser la conversion d'un type abstrait, dit TUOPA¹, en langage JAVA et inversement. Le format de stockage du type abstrait est laissé à notre discrétion. En outre, il est imposé la présence d'une méthode « toString() » dans le fichier JAVA généré à partir du type abstrait.

Dans un premier temps, une recherche de la structure du fichier contenant le TUOPA va être nécessaire afin de pouvoir l'analyser. Le formalisme XML est le format que nous avons choisi, car c'est un langage qui est dit « rigoureux et structuré ». De plus, il est facile à manipuler et il offre une meilleure lisibilité qu'un fichier texte basique. Une fois ce choix effectué, il faut déterminer le code JAVA qui résulte de ce type abstrait, son interface, le squelette de la classe d'implémentation ainsi que l'intégralité du code source de la classe, avec les types de retour, les conditions implémentant les pré-conditions, etc.

Dans le cas suivant, JAVA vers TUOPA, la méthode va consister à faire une analyse syntaxique du code. Cependant, cette analyse devra être plus « adaptable » que la précédente étant donné que la rédaction du code n'est pas aussi rigoureuse qu'un document XML reposant sur une DTD. Une fois cette analyse effectuée et les éléments importants trouvés, la construction du TUOPA est facilitée par le formalisme XML, l'ajout d'élément reposant sur le principe des nœuds (un nœud = un élément).

¹ Type Utilise Opération Pré-condition Axiome

II- RAPPEL

2.1 - Caractéristiques d'un type abstrait (TUOPA)

Il est composé de 5 caractéristiques : (voir annexes p.2)

Type abstrait	Nom du type abstrait
Utilise	Type de variable utilisée
Opération	Ensemble de signatures d'opérations définissant la syntaxe de ces opérations
Pré condition	Opérations logiques qui doivent être vari et qui permettent le fonctionnement de certaines opérations.
Axiome	Précise le comportement attendu des opérations spécifiées, c'est la sémantique.

2.2 – Caractéristiques d'une interface JAVA

Elle contient les signatures des méthodes. C'est une extension de la notion de classe abstraite : pour une interface toutes les méthodes sont abstraites. Une interface ne peut contenir que des attributs abstraits. Il est possible d'implémenter plusieurs interfaces.

Exemple d'interface :

```
public interface monInterface {  
    //Liste des méthodes abstraites  
    public void methodeA();  
    public String methodeB(String maVariable);  
}
```

2.3 – Implémentation d'une classe JAVA

Une classe qui implémente une interface JAVA doit posséder l'ensemble des méthodes et signatures déclarées dans l'interface.

Exemple de classe :

```
public class maClasse implements monInterface {  
    public void methodeA()    {  
        ...  
    }  
  
    public String methodeB(String maVariable)    {  
        ...  
    }  
  
}
```

III - RÉPARTITION DES RÔLES

Ce projet peut aisément se découper en 2 parties. Tout d'abord, une étape qui consiste à récupérer un type abstrait pour l'analyser et le transformer en code JAVA. Ensuite, on traite une classe d'implémentation JAVA pour la transformer en un TUOPA.

Il paraît évident à nos yeux, étant donné que nous formons un binôme, que chacun s'occupe d'une partie du projet. Ainsi chaque personne peut avancer sans se soucier vraiment de l'avancement de l'autre car il n'a pas besoin de l'intervention d'autrui pour progresser, sauf dans le cas où ses travaux reposent sur les accesseurs de son partenaire. Pour être plus précis chacun des membres du binôme peut 'librement' mettre en place sa stratégie d'analyse (une pour le fichier TUOPA et une pour le fichier JAVA).

Nous avons alors décidé que Jérôme LERAY se chargerait de la partie concernant TUOPA vers JAVA et Hervé GOUCHET se chargerait de la procédure JAVA vers TUOPA. En outre, Hervé aura la responsabilité de réaliser une interface graphique conviviale et Jérôme celle de construire des structures de données communes permettant de stocker les résultats des analyses dans les deux sens.

IV - CHOIX TECHNIQUES

4.1 - Choix du formalisme du type abstrait

Pour mieux cibler le format de stockage optimal, nous avons commencé par lister les informations que doit contenir un type abstrait. On a débuté par le plus haut niveau (4 grandes rubriques : Type, Utilise, Opération, pré-conditions, Axiomes), jusqu'au plus petit (une fonction = un nom + des paramètres + un type de retour ou encore une variable = un nom + un type + une valeur).

Dans un premier temps une solution organisée à l'aide de marqueur textuel du type «# » ou encore « , » avait été retenue (voir annexes p.3). Celle-ci avait l'avantage de posséder une analogie très forte avec la représentation du TUOPA étudié en cours, seuls les marqueurs y étaient ajoutés. Mais cette organisation des données n'était pas sans poser de problèmes lors de la lecture par le programme JAVA et même par « l'homme ». En effet une syntaxe aussi proche du langage 'humain' est difficile à interpréter. Malgré ce constat nous avons tout de même conçu et réalisé des classes JAVA capables de comprendre cette syntaxe.

Toutefois, suite à la remarque de M. Lemaire concernant la représentation du TUOPA en fichier texte 'classique', nous avons reconsidéré notre solution et décidé de changer de format de données pour un stockage en XML. Il est vrai qu'un format XML est plus facile à lire et plus aisé à mettre en place car il existe déjà des méthodes de parcours de fichier XML. De plus, un format aussi structuré ne nuit en rien à la compréhension humaine, il permet même d'être plus rigoureux car l'on peut associer au fichier XML une DTD (voir annexes p.5).

Cette solution a donc été définitivement adoptée.

4.2 - Organisation du format XML

4.2.1 - Les éléments du TUOPA

Le type abstrait comporte plusieurs informations qu'il est nécessaire de détailler ici avant d'expliquer l'organisation du fichier XML. Ainsi un type abstrait s'organise en 5 grands éléments :

- Le champ Type
- Le champ Utilise
- Le champ Opération
- Le champ Pré condition
- Le champ Axiome.

Chacun de ces champs est lui-même divisé :

Type : il est le plus simple des champs, il ne possède que le nom du type abstrait.

Utilise : il regroupe l'ensemble des types de variables utilisées, c'est donc une liste de types à stocker.

Opération : il regroupe l'ensemble des opérations, celles-ci sont composées de trois éléments : un nom d'opération, un type de retour et enfin des paramètres. Chacun de ces paramètres est constitué de variables ou d'objets.

Pré-condition : Il regroupe l'ensemble des pré-conditions, chacune d'elle se découpe en deux éléments : l'élément 'gauche' et l'élément 'droite'.

L'élément 'gauche' est constitué d'une opération et des paramètres éventuels de l'opération.

L'élément 'droite' est plus complexe car il est variable. En effet, il peut être composé tout comme l'élément 'gauche' mais il peut aussi être une variable ou bien encore une valeur booléenne. L'élément 'droite' peut en outre contenir un opérateur de comparaison.

Axiome : Il est sans conteste l'élément le plus difficile à représenter et donc à analyser. Il est, à l'instar de pré condition, constitué d'un élément 'gauche', d'un opérateur et d'un élément 'droite'.

L'élément de gauche est constitué d'une opération qui peut dans ses paramètres contenir une autre opération.

L'élément de droite peut prendre la forme d'une valeur booléenne, d'un objet ou d'une variable ou bien encore d'une opération avec des paramètres qui peuvent également renfermer une autre opération.

Pour plus de clarté, voici un tableau récapitulatif des différentes formes recensées:

Champ TUOPA	Formes possibles
Type	Pile
Utilises	int
Opérations	estVide : Pile → Boolean
Pré-conditions	dépiler(p) ⇔ !estVide(p) //P1 nieme(f,n) ⇔ n < nbArbre(f) //P2
Axiomes	estVide(créer) = vrai //A1 sommet(empiler(p,e)) = e //A2 premier(l) = contenu(tete(l)) //A3 !estVide(reste(l)) = succ(tete(l)) = tete(reste(l)) //A4 (non implémenté)

Deux autres éléments plus modulaires sont présents dans ce type abstrait, ils servent à décrire ce dernier. Ainsi il va falloir prendre en compte les opérateurs de comparaison et les variables.

Remarque 1 : Dans le cadre d'une opération comme $\text{dépiler}(p) \Leftrightarrow \text{!estVide}(p)$, l'élément de gauche est « $\text{dépiler}(p)$ » et l'élément de droite est « $\text{!estVide}(p)$ ».

Remarque 2 : Dans le cadre d'un axiome comme $\text{premier}(l) = \text{contenu}(\text{tete}(l))$, l'élément de gauche est « $\text{premier}(l)$ » et l'élément de droite est « $\text{contenu}(\text{tete}(l))$ ».

4.2.2 – La DTD (voir annexes p.4)

Maintenant que l'on connaît les informations à stocker dans le fichier XML, il faut choisir un mode de stockage (attribut ?, valeur ?). Par analogie avec le type abstrait la balise racine s'appellera ' <tuopa> ' et aura 5 fils : ' <ta> ', ' <utilise> ', ' <operations> ', ' <preconditions> ', ' <axiomes> '. Chaque fils correspond nommément à une rubrique du type abstrait.

La balise ' <ta> ' ne possède pas de fils, elle renferme seulement le nom du type abstrait :

Exemple : (exmple complet, voir annexes p.5)

```
<ta>Pile</ta>
```

La balise ' <utilise> ' possède une balise fille appelée ' <types> ', cette balise contient un type et un seul. S'il y a plusieurs types à indiquer il y aura donc plusieurs balise ' <type> '.

Exemple :

```
<utilise>
  <type>Boolean</type>
  <type>Element</type>
</utilise>
```

La balise ' <operations> ' contient un seul type de balise : ' <operation> '. Une balise ' <operation> ' contient une opération et une seule alors que la balise ' <operations> ' contient l'ensemble des opérations. La balise ' <operation> ' possède obligatoirement au moins deux balises filles qui sont ' <nom> ' et ' <retour> '. Le nom de l'opération étant stocké dans ' <nom> ' et le type de retour ' <retour> '. Une seule balise fille est optionnelle : ' <parametres> '. Elle contient l'ensemble des paramètres de l'opération.

Exemple : Opération : empiler(Pile, Element) → Pile

```
<operations>
  <operation>
    <nom>Boolean</nom>
    <parametre>
      ...
    </parametres>
    <retour>Element</retour>
  </operation>
  <operation>
    ...
  </operation>
</operations>
```

La balise '`<preconditions>`' regroupe l'ensemble des pré-conditions, chaque pré-condition est marquée par une balise '`<precondition>`'. Cette balise possède trois balises filles : '`<gauche>`', '`<opérateur>`' et '`<droite>`'. La première balise fille possède les informations sur l'élément de gauche et la balise '`<droite>`' regroupe les informations sur la partie droite de la pré condition. La balise '`<opérateur>`' possède deux rôles :

Le premier, lorsque la pré-condition est de type P1 (voir tableau ci-dessus), elle symbolise la présence du booléen.

Le second, la pré-condition est de type P2 elle symbolise l'opérateur de comparaison, dans notre cas le symbole '<'. Il est à noter que la balise de fin n'a pas de balise fermante et que l'information est stockée dans l'attribut valeur. Les deux balises droite et gauche sont détaillées plus bas.

Exemple : pré-condition : dépiler(p) ⇔ !estVide(p)

```
<preconditions>
  <precondition>
    <gauche>
      ...
    </gauche>
    <opérateur valeur="!" />
    <droite>
      ...
    </droite>
  </precondition>
</preconditions>
```

La balise '<axiomes>' joue le même rôle pour les axiomes que la balise précédente pour les pré-conditions. Elle sert à regrouper l'ensemble des balises filles '<axiome>', cet élément représentant un axiome. La balise 'axiome' est composée des même éléments que la balise pré-condition.

La balise '<gauche>' correspond à l'élément de gauche d'une pré-condition ou d'un axiome. Elle contient obligatoirement une balise '<nomOperation>' qui contient le nom de l'opération concernée (on expliquera ce choix ultérieurement). Deux balises sont optionnelles : la balise '<parametres>' qui recense les paramètres de l'opération et la balise '<boolean>' qui indique si l'élément est soumis à un résultat booléen négatif. Cette dernière ne possède pas de balise fermante et l'information est stockée dans l'attribut valeur.

La balise '<droite>' correspond à l'élément de droite d'une pré-condition ou d'un axiome. Compte tenu de la complexité de l'élément à décrire cette balise peut se décliner en plusieurs combinaisons de balise filles. Trois combinaisons sont possibles.

1. L'élément de droite est un simple booléen (cas A1 : « ...= faux »), la balise '<droite>' aura alors une seule balise fille qui est booléenne (voir paragraphe précédent).
2. L'élément de droite est une variable (cas A2 : « ..=e »), '<droite>' aura également une seule balise fille qui est la balise '<variable>'.
3. L'élément de droite est une fonction ayant une autre fonction en paramètre (cas A3). Dans ce cas la balise '<nomOperation>' est obligatoire et contient le nom de l'opération principale ^{*1}. On ajoutera alors une balise '<parametre>' qui elle-même contiendra une balise '<nomOperation>'. Cette dernière contiendra le nom de l'opération en paramètre ^{*2}.

« = contenu ^{*1} (tete ^{*2} (l)) »

La balise '<variable>' est présente sous deux formes : fermante ou non. Elle sert à stocker les variables indiquées dans les paramètres ou les axiomes. Elle possède obligatoirement l'attribut 'type' qui renseigne sur le type de la variable. L'attribut 'nom' est quant à lui optionnel, il indique le nom de la variable si il existe. Si la balise concerne une variable du type 'Operation', elle sera fermante et aura la balise fille '<nomOperation>', avec tout autre type la balise variable sera fermante.(ex : <variable />)

La balise '<nomOperation>' contient le nom d'une opération indiqué dans la rubrique 'Operations'. Elle sert de pointeur. Cette balise représente avantageusement une balise '<operation>' et évite de répéter son type de retour et ses paramètres.

L'ensemble des structures décrites ci-dessus découle pour partie du choix que l'on a fait de privilégier les opérations vis-à-vis des pré-conditions et des axiomes. Ainsi une pré-condition reprend obligatoirement les opérations concernées. Il est évident que nous perdons l'information sur les variables mais n'ayant pas réussi à exploiter ces dernières, nous n'avons pas jugé utile de les mettre.

Exemple avec un axiome :

```
Information d'origine : premier(cons(l,e)) = e  
Information XML : premier(cons) = e
```

Les type des paramètres sont stockés dans les opérations correspondantes, il est impossible, dans le cadre de nos choix, de savoir si l'élément inséré est le même que l'élément retourné.

4.3 - L'interface graphique (voir annexes p.11)

Afin de présenter au mieux et de permettre une utilisation assez simple de ce convertisseur JAVA vers TUOPA et inversement, on a mis en place une interface de gestion. L'accent au cours de sa réalisation a été mis sur la convivialité. Ainsi en un clic sur le menu générer ou grâce à un raccourci clavier, un fichier JAVA devient un TUOPA en XML.

En outre, pour faciliter la lecture des différents codes, des styles d'écritures ont été mis en place. Ainsi, les balise XML apparaissent en bleu, leur contenu en noir. De même, au niveau du code JAVA, les éléments caractéristiques de ce code sont mis en évidence.

Enfin, il est à noter que dans un souci d'esthétique, l'interface graphique repose sur la charte établie par la plate-forme sur lequel elle se lance. Ainsi si on démarre JavaToTuopa² sur un ordinateur qui tourne sur Linux, ce dernier possèdera l'apparence Linux. (voir annexes p.12)

² le nom que l'on a donné à ce mini logiciel

V - ORGANISATION DU PRODUIT

5.1 - Les classes, objet et méthodes (voir annexes p.7)

L'ensemble des classes peut-être distingué en 4 fonctionnalités :

1. Analyse de fichiers XML ou JAVA
2. Stockage des données résultant de l'analyse
3. Restitution des données en JAVA ou XML
4. Interface graphique

Ainsi on retrouvera l'objet 'TA' et 'AnalyserJava' respectivement responsable de l'analyse d'un fichier Xml et d'un fichier JAVA dans le groupe 1.

Le stockage des données est organisé de la même façon que le type abstrait est construit, ainsi on retrouvera les classes 'Operation', 'PreCondition', 'Axiome'. Il est à noter que chaque structure de stockage possède des méthodes facilitant leur interaction avec les objets d'analyse et de restitution. En effet, si on prend la classe 'Operation', cette dernière possède une méthode permettant de générer le XML intégrant les opérations.

Les classes 'JavaFichierTxt' et 'TxtFichierJava' permettent respectivement de restituer le code XML et JAVA demandé au cours de la conversion.(Cf. : documentation javadoc).

Comme son nom l'indique le 4^{ème} groupe concerne toutes les classes relatives à la gestion de l'IHM.

5.2 - L'analyse TUOPA vers JAVA

La conversion d'un fichier XML contenant le type abstrait vers le code JAVA correspondant s'effectue en 2 étapes majeures : l'analyse du fichier XML et la restitution en langage JAVA.

5.2.1 - Analyse du fichier XML

Comme pour tout fichier XML la première étape consiste à parser le fichier. Le type de parseur qui a été retenu est DOM car il permet de parcourir l'arbre des éléments. L'API utilisée pour mettre en place DOM est JAXP. Elle est implémentée dans la classe TA grâce aux imports **org.w3c.dom.*** et **javax.xml.parsers.***. L'objet utilisé pour stocker l'arbre des éléments est DocumentBuilder.

L'analyse est découpée en 5 étapes dont chacune correspond à une rubrique du TUOPA. Pour repérer chaque rubrique au sein du fichier on utilise une méthode de l'objet DocumentBuilder qui est

getElementsByTagName('String maBalise'). Cette méthode nous retourne ensuite une interface de type `NodeList` qui permet d'accéder à chaque `Node` portant ce nom.

L'interface `Node` est primordiale dans notre analyse. En effet, elle permet de savoir si un nœud possède des fils et si oui, d'en obtenir la liste. Un nœud n'est pas nécessairement un élément, c'est-à-dire qu'il ne correspond pas obligatoirement à une balise. Ainsi un nœud peut être de type `ATTRIBUTE_NODE` (c'est un attribut contenu dans un nœud) ou encore de type `CDATA_SECTION_NODE` (texte contenu entre une balise ouvrante et sa balise fermante). Pour vérifier le type d'un nœud l'interface `Node` met à notre disposition la méthode **getNodeType()**. Grâce à ces méthodes, le parcours des fils d'un nœud se résume à demander la liste des nœuds fils, puis pour chacun d'en tester le type et enfin de comparer son nom avec le nom recherché.

L'analyse débute par la recherche du nom du type abstrait. Pour cela on recherche la balise '<ta>' et on en récupère sa valeur. On poursuit avec la rubrique Utilise, on pointe sur la balise qui rassemble l'ensemble les types soit '<utilises>'. On effectue alors la procédure expliquée ci-dessus pour retirer la valeur des nœuds fils (: '<type>').

Les trois rubriques restantes sont plus complexes à analyser car elles contiennent plusieurs niveaux dans leur structure hiérarchique et les informations présentes ne sont pas figées. Ainsi une opération ne possède pas obligatoirement de paramètre (ex : créer()). Cependant, la même méthode de parcours s'applique.

5.2.2 - La restitution en langage JAVA

La restitution du type abstrait en JAVA est pris en charge par l'objet 'TxtFichierJava'. Il a la charge de récupérer les informations stockées dans les objets 'Operation', 'PreCondition' et 'Axiome', tous regroupés dans l'objet 'TA'.

Comme l'objectif du produit le stipule, la restitution est de 3 ordres :

- interface
- squelette d'implémentation de la classe
- code de la classe (partiel)

- L'interface :

L'interface java est seulement constituée du nom du type abstrait et de la signature des méthodes. On peut faire une analogie entre le nom de la classe d'interface et le nom du type abstrait. En ce qui concerne les méthodes, celles-ci ont les mêmes caractéristiques que les opérations déclarées dans

le type abstrait. Toutefois, par convention, dans un TUOPA toutes les opérations prennent en paramètre un objet de même nature que le type en court de définition. Pour récupérer les signatures des méthodes, il faut alors ne pas tenir compte de la première occurrence de même type que le type abstrait en cours, on récupère ensuite le type de retour de l'opération, ainsi que son nom pour construire les méthodes.

Exemple avec une opération du type abstrait Pile :

```
estVide : Pile → boolean donne public boolean estVide () ;
```

- Le squelette JAVA :

Il n'existe pas de différence notable entre l'interface et le squelette java. On notera tout de même que la mise en place d'une méthode est ponctuée par '{' et '}' alors que seul un ';' suit la signature de la méthode dans l'interface. De plus, lors de la spécification de la classe, on ajoute le mot clé 'implements' suivi du nom du type abstrait. Le nom de la classe d'implémentation est alors écrit avec en préfixe : 'Impl' suivi du nom du type abstrait. Ceci est un choix délibéré de notre part pour ne pas confondre l'interface et sa classe d'implémentation.

- Le code JAVA :

La difficulté au cours de la génération du code java est l'interprétation des pré-conditions et des axiomes. En effet, on peut faire l'analogie avec une boîte noire dont on ne connaît que ce qui en sort et ce qui y entre. Il faut alors interpoler le code java correspondant à la boîte noire en se basant sur les paramètres et les types de retour. Il paraît donc évident qu'il est impossible de générer la totalité du code.

Néanmoins, certaines parties des pré-conditions peuvent être interprétées. En particulier, les pré-conditions dans lesquelles l'élément de droite est constitué d'une unique valeur booléenne ou d'une opération simple³.

Exemple :

```
depiler(p) ⇔ !estvide
```

Il est alors facile d'interpréter que si 'estVide' est 'vrai' il ne faut pas exécuter cette méthode. Le code correspondant sera donc inséré dans le code de la méthode depiler().

³ Simple : une opération qui ne prend pas en paramètre une autre opération

Exemple :

```
if (estVide()) {System.exit();}
```

En ce qui concerne les axiomes, nous n'avons malheureusement pas réussi à les exploiter. Nous nous sommes heurté sur interprétation automatique de ces derniers. Nous avons essayé de chercher les éléments communs à tous les TUOPA. La méthode créer() nous est apparut alors la plus récurrente. On a donc essayé d'établir des liens de cause à effet depuis le postulat selon lequel tout objet nouvellement créé est vide. Cependant, à chaque fois nous sommes venus buter sur l'action qu'effectue une opération sur l'objet de départ. Ainsi une méthode 'empiler' (pour une Pile) qui prend en paramètre un objet et retourne une Pile a pour fonction d'ajouter un élément à la pile. Or, seul le fait de modifier la pile est traduit dans les axiomes, on ne peut donc pas savoir s'il s'agit d'un ajout ou d'une suppression.

Conscient de nos lacunes et du temps qui nous était donné nous avons choisi de ne pas pousser notre recherche plus en avant sur cette partie du projet. Toutefois, nous avons décidé que tout objet d'un TUOPA serait implémenté sous la forme d'un vecteur, réduisant notre champ de compétence qu'aux classes de type linéaire. Le constructeur initialise donc un vecteur « à vide ». La méthode toString() affichera le nombre d'objets contenus dans la structure.

5.3 - L'analyse JAVA vers TUOPA

5.3.1 – Introspection

Nous sommes partis dès le début du projet sur le système d'analyse syntaxique d'un document JAVA, ne connaissant pas l'existence de la classe Class. Cette classe permet, après compilation, de récupérer en outre, le nom des constructeurs, leurs signatures et les méthodes. Même si cette classe répond en partie à notre besoin, nous avons préféré conserver le système d'analyse syntaxique pour deux raisons :

- Premièrement, l'utilisation de la classe Class aurait été viable que pour une partie de l'analyse, la partie pré-condition aurait nécessité l'utilisation de l'analyse syntaxique.

- Enfin, la compilation de fichier JAVA faisant référence à des méthodes ou à des classes se trouvant dans d'autres fichiers aurait posé des problèmes de compilation. Dans ces conditions, la recherche des méthodes aurait pu s'avérer non fiable.

5.3.2 - Analyse du fichier JAVA

L'inconvénient majeur d'un fichier JAVA est que contrairement à des langages comme XML, il n'est en aucun cas rigoureux au sens stricte du terme. Certes des éléments sont obligatoirement présents pour qu'un fichier java soit valide, mais leur positionnement au sein du fichier, leur syntaxe n'est pas définie précisément. Alors lorsque que l'on fait une analyse syntaxique de ce dernier, toute la difficulté est de repérer les éléments qui seront utiles à la création du type abstrait.

Dans l'objectif d'analyser au mieux un fichier java, plusieurs routines sont mises en place :

Premièrement, il faut nettoyer le code java afin d'en garder seulement les éléments utiles. Ainsi par exemple, on enlève tous les commentaires du fichier.

Ensuite, on repère la position de chaque élément, afin de les récupérer par la suite. Le principe qui a été mis en place pour analyser un fichier java, est la lecture ligne par ligne dans un premier temps et par la suite la lecture mot par mot de cette ligne. Cette analyse permet de faire une analyse syntaxique précise mais plus complexe. Car le langage java est assez libre au niveau de la syntaxe, ainsi on peut écrire :

```
public class maClasse{
```

Mais aussi :

```
public class maClasse {
```

La nuance est subtile, mais ce genre de problème se pose aussi dans le cas où l'on déclare une méthode de la sorte :

```
public String maMethode (String monParam,  
                        int monAutreParam)    {
```

ou :

```
public static int maMethode(int monParam)    {
```

ou encore :

```
public int maMethode(int monParam){
```

En bref, la liste des cas particuliers à gérer assez longue dans ce genre d'analyse. L'utilisation de méthodes et de fonctions propres à la classe String sont nécessaires pour ce genre d'analyse, en voici une liste exhaustive :

- Regex : Pattern et Matcher, qui permettent de repérer des caractères précis dans une chaîne, dans donner sa position ou même de les remplacer par d'autres caractères.
- indexOf : cette méthode possède la même fonction que matcher, cependant il ne repère que la position de la première occurrence.

- split : qui permet de découper une chaîne en fonction d'un caractère précis ou d'un regex.
- substring : permet de découper une chaîne de caractère en fonction de 2 positions.

Une fois le nom de la classe, les types de variables utilisés, les constructeurs et les méthodes de repérés, il reste à déterminer les pré-conditions. Grâce à l'analyse effectuée précédemment, on sait le nombre et le nom des méthodes existantes dans le fichier, il ne reste plus qu'à déterminer la position de début de cette méthode et celle de fin afin de récupérer son contenu dans le but de l'analyser. C'est l'analyse des conditions de la méthode qui permettra de déterminer si la méthode possède ou nom des pré-conditions. On va se référer tout d'abord au contenu de la condition, si une méthode est détectée, alors on détermine la condition qui lui est associé, pour en déduire par la suite, la formulation de la pré – condition.

Exemple :

```
if (estVide()) {  
    System.err.println("Erreur");  
    System.exit(1);  
}
```

5.3.3 - Récupération des éléments pour le TUOPA

L'analyse du fichier JAVA (cf. : AnalyserJava.java) à fait apparaître tout les éléments indispensables à la construction d'un type abstrait. Désormais, grâce à leurs positions, on va pouvoir récupérer ces éléments et construire un objet TA (cf. : TA.java). C'est cet objet qui sera l'objet de stockage du TUOPA.

5.3.4 - Construction du fichier XML représentant le type abstrait

La restitution du type abstrait en XML est pris en charge par l'objet 'JavaFichierTxt'. Il a la charge de récupérer les informations stockées dans les objets 'Operation', 'PreCondition' et 'Axiome', eux même tous regroupé dans l'objet 'TA'. Grâce aux accesseurs de ces différents objets, 'JavaFichierTxt' va pouvoir construire nœud par nœud le fichier XML représentant le type abstrait généré à partir du fichier java, en ajoutant dans l'entête le prologue XML ainsi que le lien vers la DTD.

VI - CONCLUSION

Ce projet était un projet très ambitieux du point de vue algorithmique, il contenait une analyse syntaxique et une analyse logique de données. Ces deux procédés ont nécessité quelques recherches afin de bien cerner toutes les difficultés de ce type de procédure. En outre, le sujet en lui-même a révélé quelques problèmes, notamment au niveau de la gestion des axiomes. L'interprétation et le repérage des axiomes au sien d'un fichier JAVA nous sont apparus trop complexes à cause de la détermination des liens logiques entre les différentes méthodes. Ce souci est apparu dans le cadre de l'analyse d'un TUOPA. En revanche, la détermination de tous les autres éléments du type abstrait à été effectuée avec succès. Ce logiciel afin d'être totalement opérationnel mériterait donc d'être amené à évoluer pour traiter l'élément Axiome et répondre ainsi aux recherches que font actuellement diverses universités dans ce domaine.

VII - BIBLIOTHEQUE

Aide XML :

<http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap031.htm>

Aide SWING et REGEX :

<http://java.sun.com/docs/books/tutorial/search.html>

Aide JAVADOC :

<http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap044.htm>

VIII - ANNEXES

8.1 - Le TA Pile (avec formalisme vue en cours)	2
8.2 - Exemple : le TUOPA pile en simple fichier texte.....	3
8.3 - La DTD	4
8.4 - Exemple : le TUOPA pile en XML	5
8.5 – L'UML de notre projet	7
8.6 - L'interface graphique	11
8.7 – Manuel utilisateur.....	12

8.1 – Le TA Pile (avec formalisme vue en cours)

```
TA
  Pile

Utilise
  boolean, element

Opérations
  créer      : -> Pile
  estVide    : Pile -> boolean
  empiler    : Pile * element -> Pile
  dépiler    : Pile -> Pile
  sommet     : Pile -> element

Pré-conditions
  dépiler(p) <=> ! estVide(p)
  sommet(p)  <=> ! estVide(p)

Axiomes
  estVide(créer) = vrai
  estVide(empiler(p,e)) = vrai
  sommet(empiler(p,e)) = e
  dépiler(empiler(p,e)) = p
```

8.2 - Exemple : le TUOPA pile en simple fichier texte

```
Pile
#boolean&
Object
#Pile() none&
Creer() Pile&
estVide(Pile:maPile) boolean&
empiler(Pile:maPile2,Object:monObject) Pile&
depiler(Pile:maPile3) Pile&
somet(Pile:maPile4) Object
#depiler(Pile:p)<=>!estVide(Pile:p)&
somet(Pile:p)<=>!estVide(Pile:p)
#none
```

8.3 – La DTD

```
<?xml version="1.0" encoding="iso-8859-1"?>

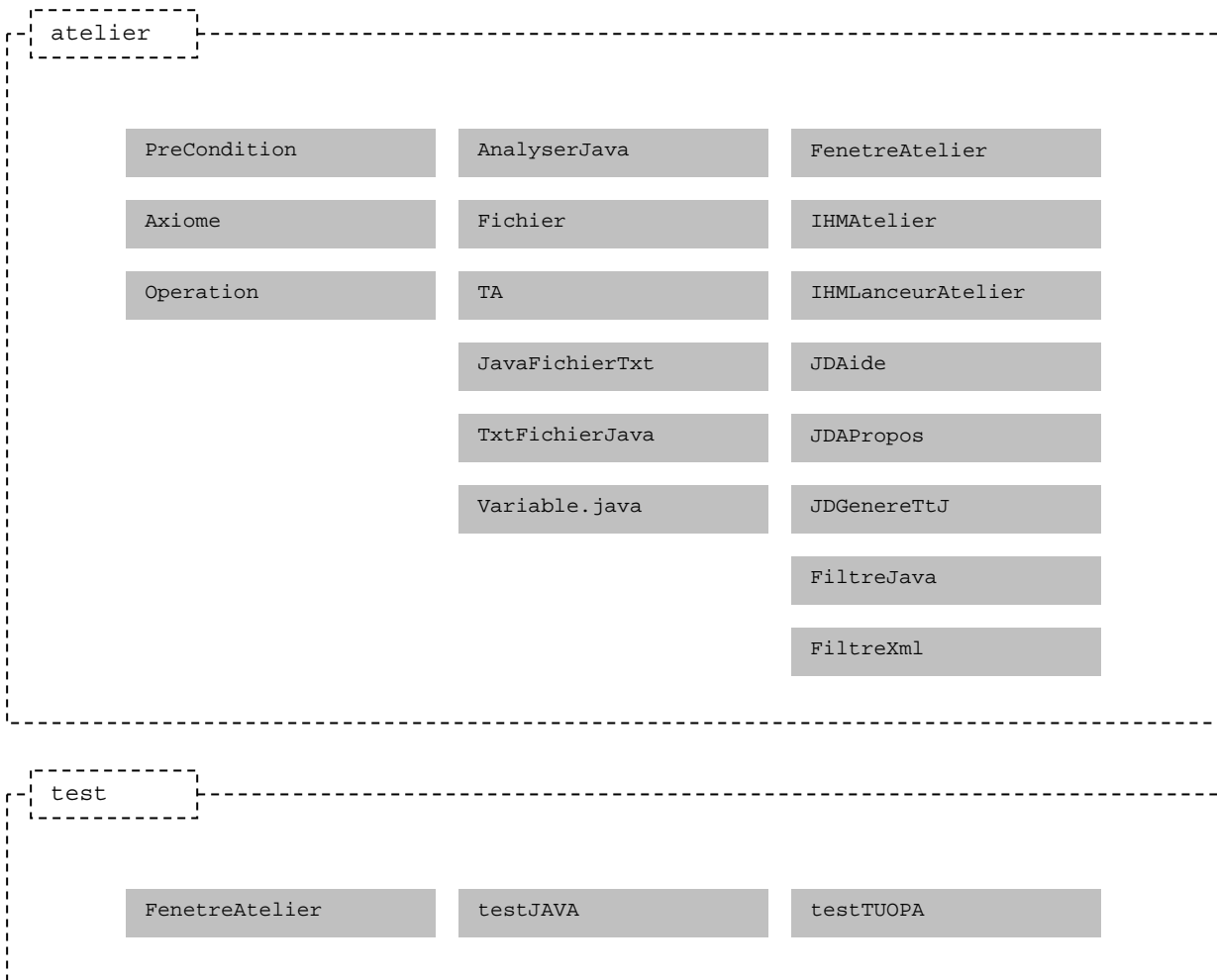
  <!ELEMENT tuopa (ta,utilises,operations,preconditions,axiomes) >
  <!ELEMENT ta (#PCDATA)>
    <!ELEMENT utilise (type*)>
    <!ELEMENT type (#PCDATA)>
  <!ELEMENT operations (operation*)>
    <!ELEMENT operation (nom,parametres?,retour)>
    <!ELEMENT nom (#PCDATA)>
    <!ELEMENT parametres (variable*)>
    <!ELEMENT retour (#PCDATA)>
  <!ELEMENT preconditions (precondition*)>
    <!ELEMENT precondition (gauche,opérateur,droite)>
    <!ELEMENT gauche (nomOperation,parametre?,boolean?)>
    <!ELEMENT nomOperation (#PCDATA)>
    <!ELEMENT opérateur EMPTY>
    <!ELEMENT droite ((nomOperation,parametre?, (opérateur,variable)?
|boolean|variable)>
    <!ELEMENT parametre (nomOperation?,variable?)>
    <!ELEMENT boolean EMPTY>
    <!ELEMENT variable (nomOperation?)>
  <!ELEMENT axiomes (axiome*)>
    <!ELEMENT axiome (gauche,opérateur,droite)>
    <!ATTLIST opérateur valeur CDATA #IMPLIED>
    <!ATTLIST boolean valeur (vrai|faux) #REQUIRED>
    <!ATTLIST variable type CDATA #REQUIRED nom CDATA #IMPLIED>
```

8.4 - Exemple : le TUOPA Pile en XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE tuopa SYSTEM "tuopa.dtd">
<tuopa>
  <ta>PileTab</ta>
  <utilise>
    <type>boolean</type>
    <type>Pile</type>
    <type>Object</type>
  </utilise>
  <operations>
    <operation>
      <nom>creer</nom>
      <parametres>
      </parametres>
      <retour>none</retour>
    </operation>
    <operation>
      <nom>estVide</nom>
      <parametres>
        <variable nom="p" type="PileTab" />
      </parametres>
      <retour>boolean</retour>
    </operation>
    <operation>
      <nom>empiler</nom>
      <parametres>
        <variable nom="p" type="PileTab" />
        <variable nom="o" type="Object" />
      </parametres>
      <retour>Pile</retour>
    </operation>
    <operation>
      <nom>sommet</nom>
      <parametres>
        <variable nom="p" type="PileTab" />
      </parametres>
      <retour>Object</retour>
    </operation>
    <operation>
      <nom>depiler</nom>
      <parametres>
        <variable nom="p" type="PileTab" />
      </parametres>
      <retour>Pile</retour>
    </operation>
  </operations>
  <preconditions>
    <precondition>
      <gauche>
        <nomOperation>sommet</nomOperation>
      </gauche>
      <opérateur valeur="!" />
      <droite>
        <nomOperation>estVide</nomOperation>
      </droite>
    </precondition>
  </preconditions>
</tuopa>
```

```
<precondition>
  <gauche>
    <nomOperation>depiler</nomOperation>
  </gauche>
  <operateur valeur="!" />
  <droite>
    <nomOperation>estVide</nomOperation>
  </droite>
</precondition>
</preconditions>
<axiomes>
  <axiome>
    <gauche>
      <nomOperation>estVide</nomOperation>
      <parametre>
        <nomOperation>creer</nomOperation>
      </parametre>
    </gauche>
    <operateur valeur="=" />
    <droite>
      <boolean valeur="vrai" />
    </droite>
  </axiome>
  <axiome>
    <gauche>
      <nomOperation>estVide</nomOperation>
      <parametre>
        <nomOperation>empiler</nomOperation>
      </parametre>
    </gauche>
    <operateur valeur="=" />
    <droite>
      <boolean valeur="faux" />
    </droite>
  </axiome>
  <axiome>
    <gauche>
      <nomOperation>sommet</nomOperation>
      <parametre>
        <nomOperation>empiler</nomOperation>
      </parametre>
    </gauche>
    <operateur valeur="=" />
    <droite>
      <variable type="Object" nom="o"/>
    </droite>
  </axiome>
  <axiome>
    <gauche>
      <nomOperation>depiler</nomOperation>
      <parametre>
        <variable type="PileTab" nom="p" />
      </parametre>
    </gauche>
    <operateur valeur="=" />
    <droite>
      <variable type="PileTab" nom="p" />
    </droite>
  </axiome>
</axiomes>
</tuopa>
```

8.5 – L'UML de notre projet



Objets de stockage du TUOPA

PreCondition	Operation	Axiome
<ul style="list-style-type: none"> - Operation opGauche - Operation opDroite - String valComp, strCompare, variableComp - String[] tabSigne 	<ul style="list-style-type: none"> - String opNom - String opRetour - Vector nomVar - Vector typVar 	<ul style="list-style-type: none"> - String opGauche - String opDroite - String opParamGauche - String operateur - String operateurDroite - String ssOpDroite - Variable varParamGauche - Variable varDroite - Boolean bGauche - Boolean bDroite
<pre> + PreCondition (TA monTa) + getPreConditionXML() : String + setCompare (String comparateur) : void + setElmntGauche(String uneOperation) :void + setElmntDroite(String uneOperation) :void + setVariable(String var, String type) : void + setValeur(String val) : void + getElmntGauche() : String + getElmntDroite() : String + isSimplePrec() : boolean + isOperation() boolean + toString() : String </pre>	<pre> + Operation () + Operation(String txt) + Operation (String nom, String retour) + donneNom(String txt) : String + setParam(String txt) : void + donneRetour(String txt) : void + isConstructeur() : boolean + setNom(String nom) : void + setRetour(String retour) : void + addParam(String nom,String type) : void + getNom() : String + getParamTuopa() : String + getRetour() : String + toString() : String </pre>	<pre> + Axiome () + setopGauche (String nomOperation) :void + setBoolGauche (String uneBGauche) : void + setVarDroite(Variable uneVariable) : void + setOpDroite(String uneOpDroite) : void + setOpParamGauche(String uneOperation) : void + setVarParamGauche(Variable une var) : void + setOperateur(String operateur) :void + setOperateurDroite(String unOperateur) : void + setOperationParamDroite (String uneOperation) : void + getOperateur () : String + getOperateurDroite (): String + getBoolGauche () : String + getBoolDroite () : String + toString() : String </pre>

Objets d'analyse et de restitution

<p>TA</p> <ul style="list-style-type: none"> - String TANom - String[] txtCoup - Vector TAUtilise - Vector TAOOperation - Vector TAPreCondition - Vector TAAxiome - Document document; - DocumentBuilderFactory factory <hr/> <ul style="list-style-type: none"> + TA() + TA(File txt) - createType() : void - createUtilise() : void - createParamOp(Node monNoeudTemp) : void - createOperation() : void - createPreOperation() : void - getOperationXML(Node operation) : Operation - createAxiome() : void + addUtilise(String txt) : void + addOperation(Vector mesOperations) : void + getNom() String + afficheUtilise() : void + afficheOperation() : void + getOperation() Vector + getPrecondition() : Vector + getUniqueUtilise () : String + debugage() : void + toString() : String 	<p>AnalyserJava</p> <ul style="list-style-type: none"> - String type - String[] javadecoupe,utilise - Vector vecOperation,vecPrecondition,vecAxiome <hr/> <ul style="list-style-type: none"> + AnalyserJava () + AnalyserJava (String java) + TrType() : String + TrUtilise(String java) : String[] + TrOperation(String java) : Vector + TrPreCondition(String java) : Vector + TrAxiome(String java) : Vector + getType() : String + getUtilise() : String[] + getOperation() : Vector + getPreCondition() : Vector + getAxiome() : Vector + toString() : String 	<p>JavaFichierTxt</p> <ul style="list-style-type: none"> - String javaTxt - String[] utilise - Vectot vOp <hr/> <ul style="list-style-type: none"> + JavaFichierTxt () + JavaFichierTxt (AnalyserJava monAnalyse) + getText() : String + toString() : String 	<p>TxtFichierJava</p> <ul style="list-style-type: none"> - String entete, txtJava - Vectot vOp <hr/> <ul style="list-style-type: none"> + TxtFichierJava () + TxtFichierJava (TA monTA) + getText() : String + toString() : String 	
		<p>Fichier</p> <ul style="list-style-type: none"> - File path - String contenu, textAnalyse <hr/> <ul style="list-style-type: none"> + Fichier() + Fichier (File path) + enregistrer(File path,String monFichier) : boolean + lecture(File path) + getContenu() : String + getTextAnalyse() : String 		

Interface graphique

```
IHMLanceurAtelier

# JComponent pan1
# JLabel monImage

+ IHMLanceurAtelier()
# static CreerImage(String
path) : ImageIcon
+ show() : void
```

```
JDAide

# JLabel lnoms,linfo
# JButton bFermer
# JPanel pan1, pan2, pan3
# JFrame proprietaire

+ JDAide(String
titreFenetreOuvrir,JFrame owner)
+ show() : void

//innerclass
ActionFermer
GestionFenetreOuvrir
```

```
JDAPropos

# JLabel lnoms,linfo
# JButton bFermer
# JPanel pan1, pan2, pan3
# JFrame proprietaire

+ JDAPropos (String
titreFenetreOuvrir,JFrame owner)
+ show() : void

//innerclass
ActionFermer
GestionFenetreOuvrir
```

```
JDGenererTtJ

# JLabel intro,blanc
# Button bChoix
final int nbBoutons
# JPanel pan1, pan2, pan3, pan4,
pan5
# JFrame proprietaire
# JRadioButton[] radioBoutons
static ButtonGroup groupBoutons

+ JDGenererTtJ (String
titreFenetreOuvrir,JFrame owner)
+ show() : void

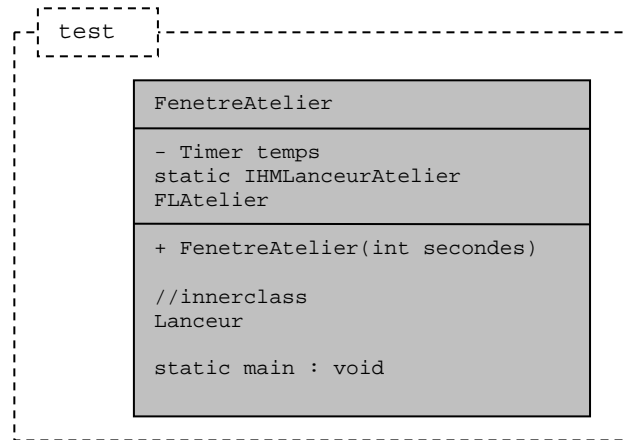
//innerclass
ActionGenerer
GestionFenetreOuvrir
```

```
FiltreJava

+ accept(File fichier) boolean
+ getDescription() String
```

```
FiltreXml

+ accept(File fichier) boolean
+ getDescription() String
```



```
IHMAtelier

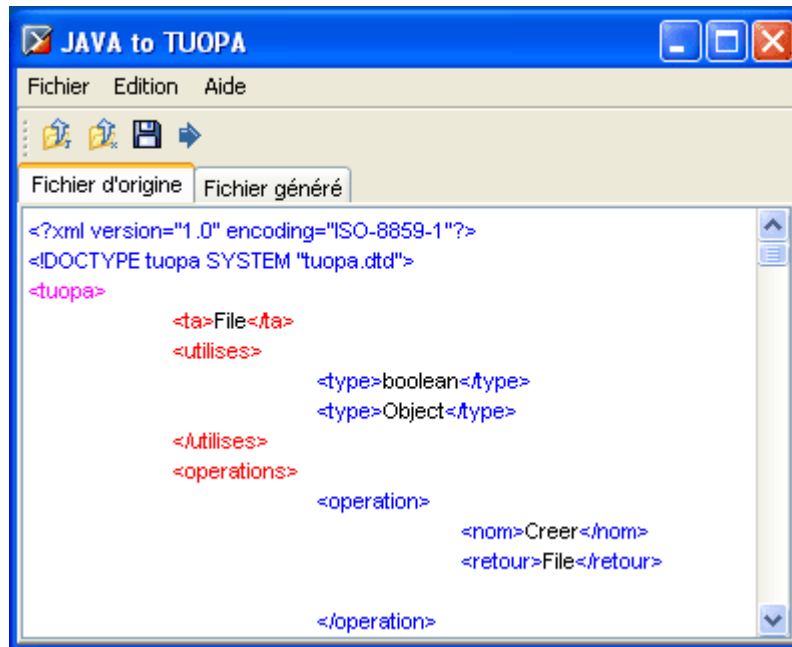
# JMenuBar menuBar1
# JMenu menu1, menu2, menu3
# JMenuItem sousmenu1,
sousmenu2, sousmenu3,
soussousmenu2, soussousmenu3,
sousmenu4, sousmenu5, sousmenu6,
soussousmenu4, soussousmenu5
# JComponent pan1, pan2
# IHMAtelier maJFrame
# JTextPane tavant,tapres
# Style styleTemporaire,
parDefaut, styleDuTexte
# StyleContext leStyle
# DefaultStyledDocument
format,format2
# JScrollPane scrollPane,
scrollPanel
# JToolBar toolBar
# JtabbedPane tabbedPane
# String choix, textAnalyse
# File nomFicTtJ
# Dimension tailleEcran
# int maxX, maxY
- String OUVRIRJAVA, OUVRIRXML,
ENREGISTRER, GENERER

+ IHMAtelier(String
titreFenetre)
# makePanel(String text,String
contenu) : JComponent
# getExtension(FileFichier) :
String
# gestionTtJ(String choix) :
void
#setCharacterAttribute(DefaultSt
yledDocument monFormat, String
monStyle, String[] mesRegex,
String contenu, String
typeRecherche) : void
# ajouterIcône(JToolBar
toolBar) : void
# ajouterBoutonMenu(String
monImage,String
actionCommand,String
toolTipText,String altTexte) :
JButton

+ show() : void

//innerclass
ActionQuitter
ActionAPropos
ActionAide
ActionEnregistrer
ActionOuvrir
ActionGenerer
GestionFenetre
```

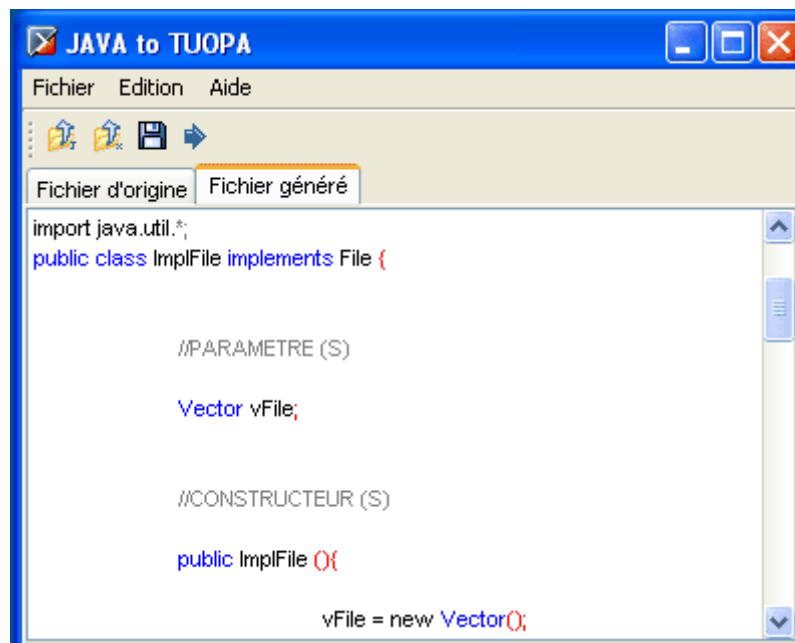
8.6 - Interface graphique



The screenshot shows the 'JAVA to TUOPA' application window. The menu bar includes 'Fichier', 'Edition', and 'Aide'. Below the menu bar is a toolbar with icons for file operations. Two tabs are visible: 'Fichier d'origine' and 'Fichier généré'. The 'Fichier d'origine' tab is active, displaying XML code. The code defines an XML structure for a file creation operation.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE tuopa SYSTEM "tuopa.dtd">
<tuopa>
  <ta>File</ta>
  <utilises>
    <type>boolean</type>
    <type>Object</type>
  </utilises>
  <operations>
    <operation>
      <nom>Creer</nom>
      <retour>File</retour>
    </operation>
  </operations>
</tuopa>
```

IHM JavaToTuopa – Fichier source en XML



The screenshot shows the 'JAVA to TUOPA' application window. The menu bar includes 'Fichier', 'Edition', and 'Aide'. Below the menu bar is a toolbar with icons for file operations. Two tabs are visible: 'Fichier d'origine' and 'Fichier généré'. The 'Fichier généré' tab is active, displaying Java code that implements the File class.

```
import java.util.*;
public class ImplFile implements File {

    //PARAMETRE (S)

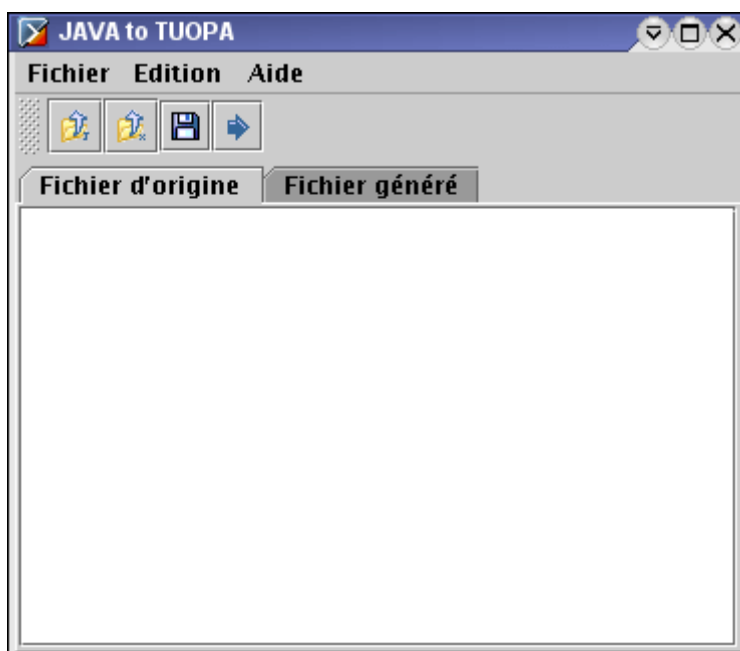
    Vector vFile;

    //CONSTRUCTEUR (S)

    public ImplFile ()

        vFile = new Vector();
}
```

IHM JavaToTuopa – Fichier généré en JAVA à partir du XML



IHM JavaToTuopa – vue sous linux

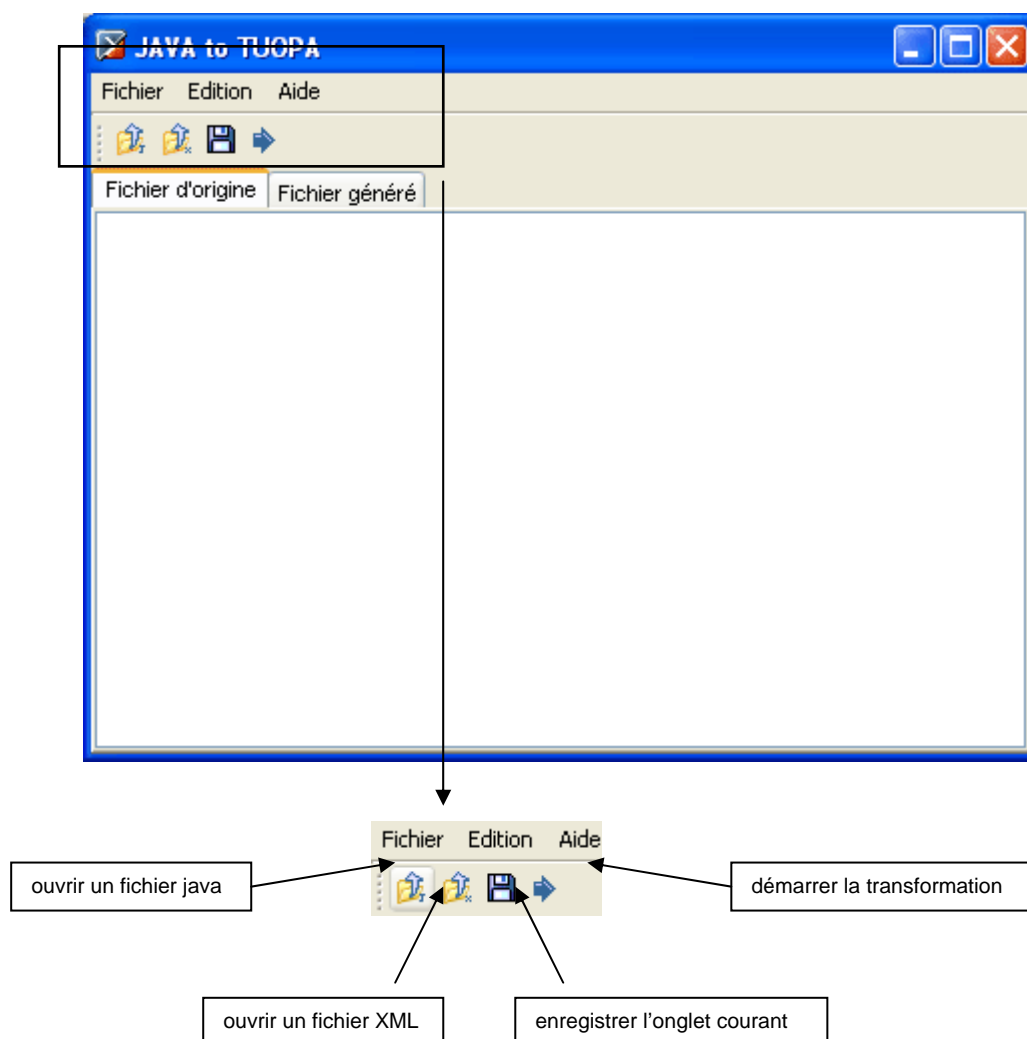
8.7 – Manuel utilisateur

Le logiciel se présente sous la forme du fichier jar que l'on lance à l'aide de la commande

```
java -jar TUOPA2JAVA.jar
```

ou par l'intermédiaire de fichier TUOPA2JAVA.jar.

Le logiciel se présente ainsi :

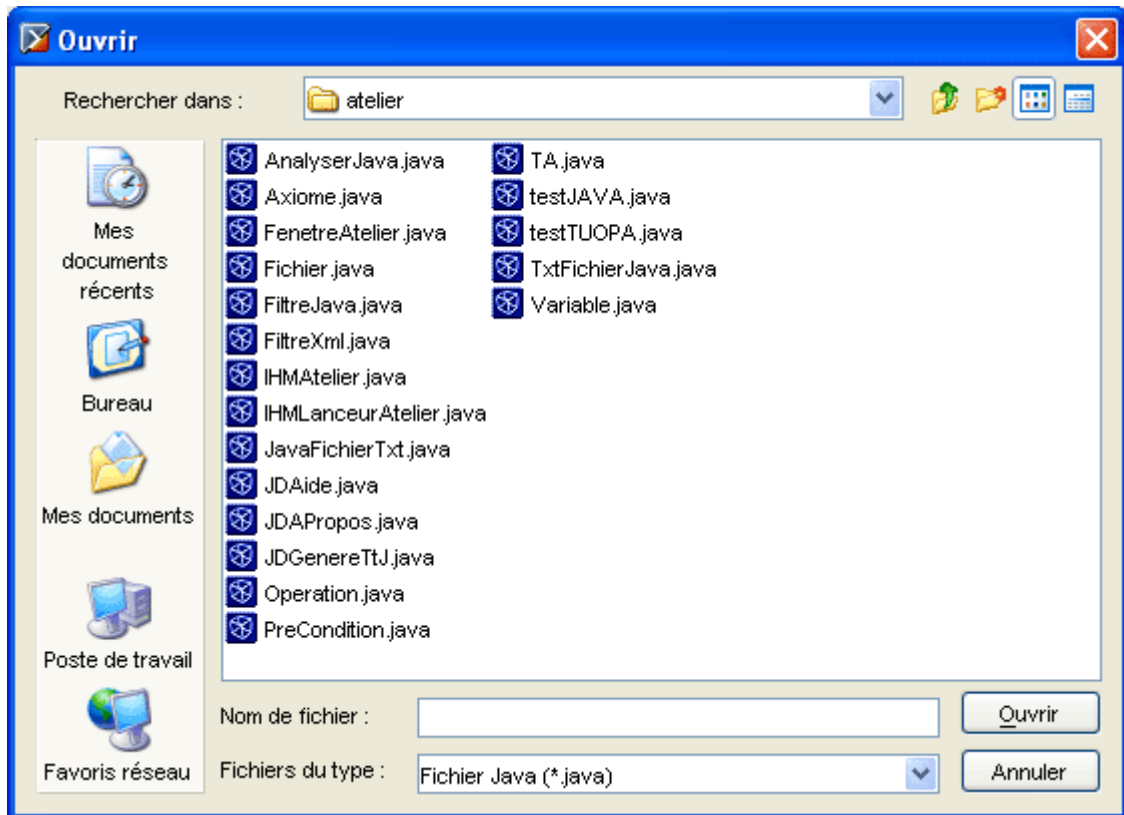


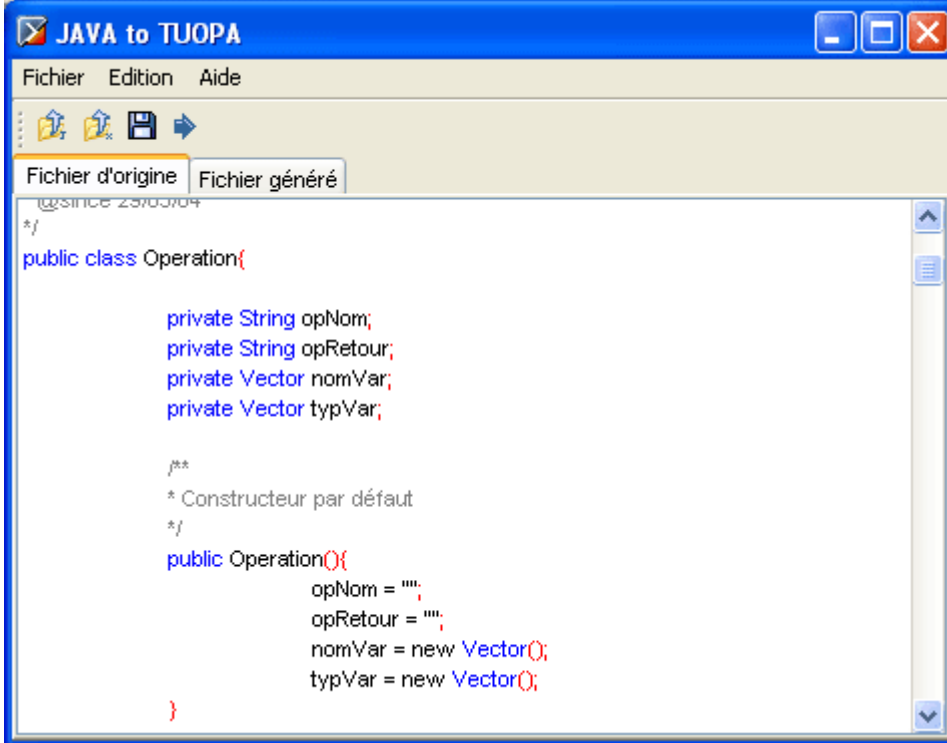
Le menu Fichier permet d'ouvrir

- un fichier java
- un fichier Xml contenant le TUOPA

Les raccourcis reprennent les fonctionnalités du menu, ainsi le premier icone en partant de la gauche permet d'ouvrir un fichier JAVA et le deuxième un fichier XML. Le troisième permet d'enregistrer le texte contenu dans l'onglet sélectionné et le dernier permet de lancer la transformation.

Ouvrir un fichier java et l'analyser:





```

@since 2.5/0.0/0.0
*/
public class Operation{

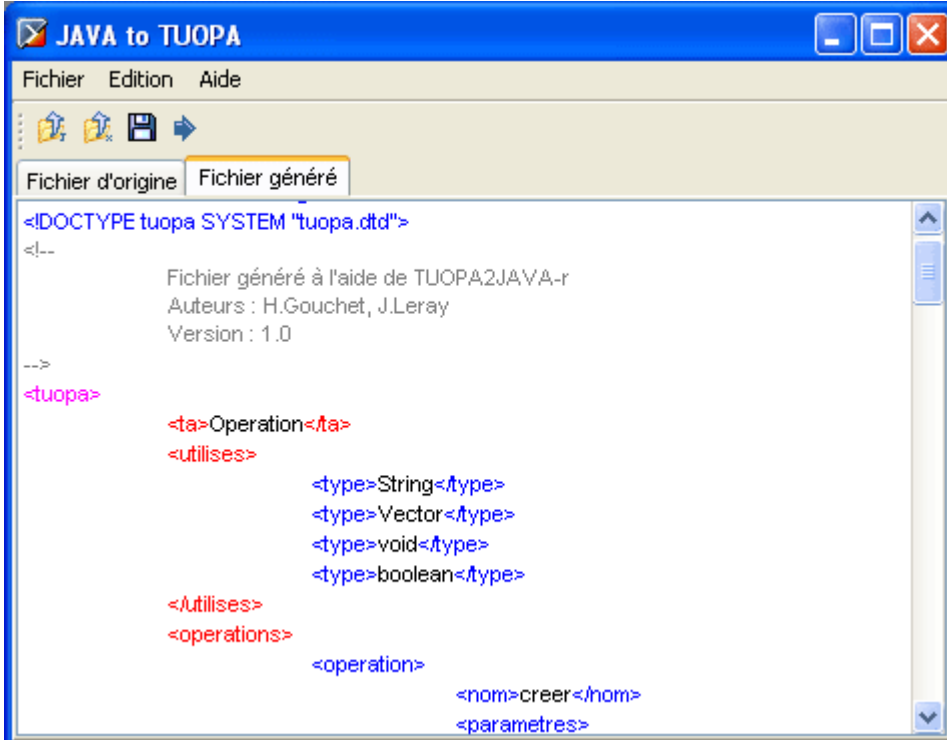
    private String opNom;
    private String opRetour;
    private Vector nomVar;
    private Vector typVar;

    /**
     * Constructeur par défaut
     */
    public Operation(){
        opNom = "";
        opRetour = "";
        nomVar = new Vector();
        typVar = new Vector();
    }
}

```

Dès que l'on a sélectionné un fichier java, l'onglet 'Fichier d'origine' est sélectionné et le contenu du fichier est placé à l'intérieur. Il ne reste plus qu'à lancer la transformation.

Résultat de la transformation : JAVA vers TUOPA



```

<!DOCTYPE tuopa SYSTEM "tuopa.dtd">
<!--
    Fichier généré à l'aide de TUOPA2JAVA-r
    Auteurs : H.Gouchet, J.Leray
    Version : 1.0
-->
<tuopa>

    <ta>Operation</ta>
    <utilises>

        <type>String</type>
        <type>Vector</type>
        <type>void</type>
        <type>boolean</type>

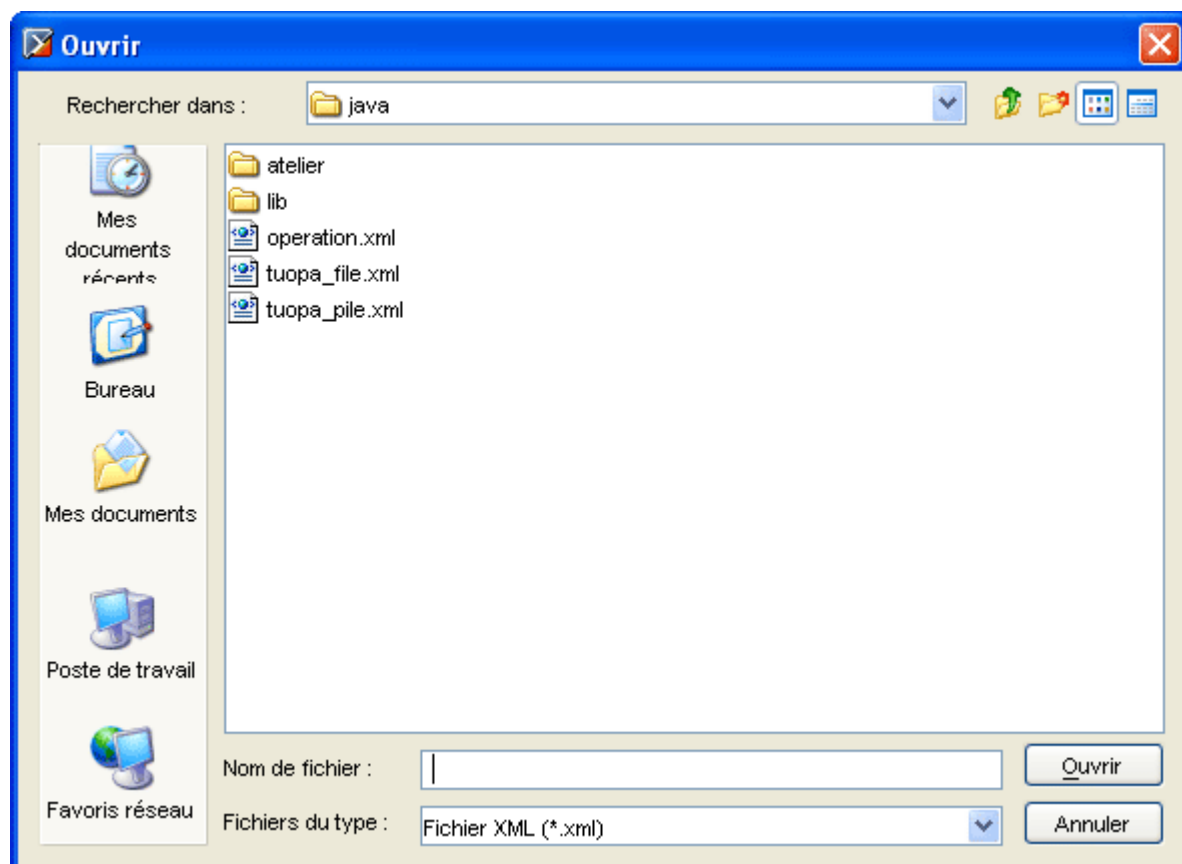
    </utilises>
    <operations>

        <operation>

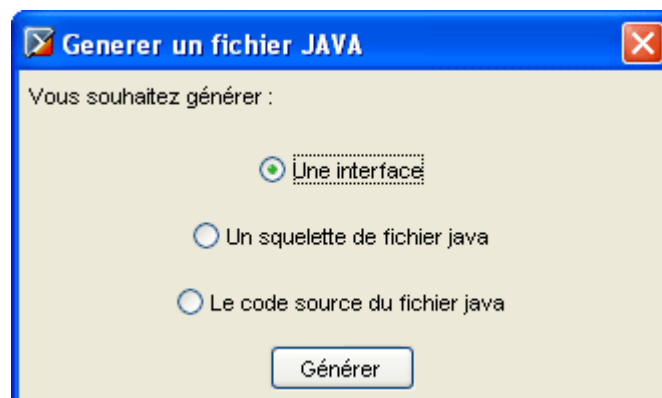
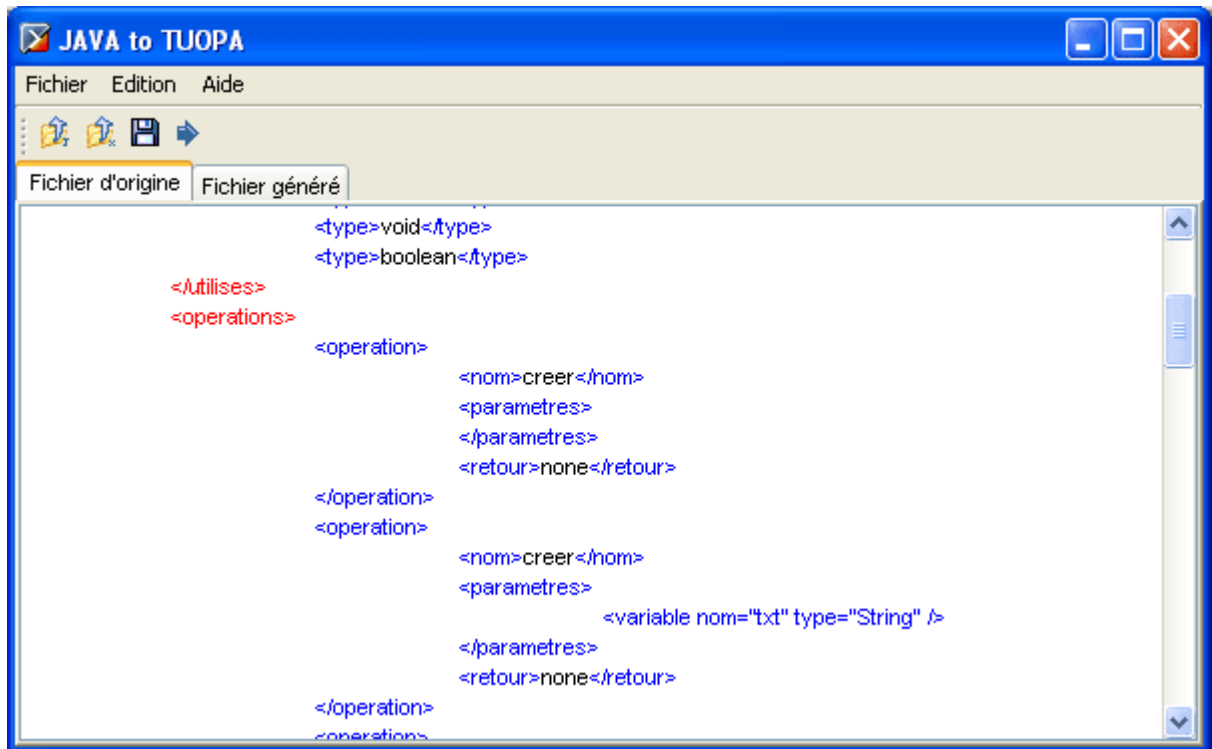
            <nom>creer</nom>
            <parametres>

```


Ouvrir un fichier XML et l'analyser:



Dès que l'on a sélectionné un fichier java, l'onglet 'Fichier d'origine' est sélectionné et le contenu du fichier est placé à l'intérieur. Il ne reste plus qu'à lancer la transformation. A la différence de la manipulation précédente une fenêtre demandant quel type de transformation est à effectuer s'ouvre.



Une fois le type de transformation sélectionnée il vous suffit de cliquer sur générer.



```

//PARAMETRE (S)

Vector vOperation;

//CONSTRUCTEUR (S)

public ImplOperation (){

    vOperation = new Vector();

}

public ImplOperation (String txt){

    vOperation = new Vector();
}

```

Résultat de la génération

Enregistrer un fichier :

Pour en enregistrer un fichier vous pouvez ouvrir le menu '*fichier*' et choisir '*enregistrer*' ou cliquer sur l'icône représentant une disquette. Le texte de l'onglet courant sera enregistré.

