UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS
Département d'informatique et d'ingénierie

Rapport final
Développement d'un éditeur pour la méthode de traces

Travail présenté à
Monsieur Michael Iglewski

Par
Mathieu Massé
François Barthe

Dans le cadre du cours
INF4173 – Projet Synthèse

Remis le
18 décembre 2006

Table des matière

Introduction....4
Développement 5
    Besoins de l'éditeur....5
    Outils Utilisés....6
    Interface Utilisateur....7
    Validation des Prédicats....9
        Analyse Lexicale 9
        Analyse Sémantique 10
Conclusion 14
    Travail future....14
Bibliographies....15
Annexe 16
    Diagrammes de classes....16

Table des Figures

Figure 1: Diagramme de haut niveau des packages 16
Figure 2 : Package inf4173.trace 17
Figure 3 : Package inf4173.utilitaire 17
Figure 4 : Package inf4173.trace.expression 18
Figure 5 : Package inf4173.trace.fonction 18
Figure 6 : Package inf4173.trace.ui 19
Figure 7 : Package inf4173.trace.ui (Menu) 20
Figure 8 : Package inf4173.trace.ui (Toolbars) 20
Figure 9 : Package inf4173.trace.ui.tableSyntaxique 21
Figure 10 : Package inf4173.trace.ui.traceCanonique 21
Figure 11 : Package inf4173.trace.ui.fonctions 22

Introduction

Le but de notre projet a été de développer un éditeur pour la méthode des traces. La méthode des traces est une méthode de spécification de module pour un logiciel. Elle consiste principalement à représenter l'historique de l'exécution d'un module dès sa création et de tous les événements qui influencent le module (habituellement des invocations de programmes d'accès).

Cette méthode pour spécifier des modules est très puissante mais relativement difficile à apprendre pour les étudiants. Les deux principales difficultés dans la méthode des traces sont: la théorie du comportement observable et la syntaxe des prédicats. Notre éditeur a donc pour but de simplifier l'utilisation et l'apprentissage de la méthode des traces, en particulier la préparation des signatures de programme et l'utilisation des prédicats.

Notre éditeur consistera de trois sections principales:

1. La table syntaxique

   Consiste en la liste des programmes avec la définition de leurs paramètres et type de retour
2. Les traces canoniques

   Consiste en la définition des traces canoniques
3. Les spécifications des programmes

   Consiste en la définition de la légalité de chaque programme avec la spécification des valeurs des paramètres modifiés et du type de retour

Développement

Besoins de l'éditeur

Les principaux besoins de l'éditeur sont de fournir à l'utilisateur l'espace pour remplir la table syntaxique, les traces canoniques et les spécifications des programmes.

Pour la table syntaxique ceci ce résume à:

- Ajouter/Modifier/Enlever des programmes
- Pour un programme:
  - Ajouter/Enlever des paramètres
  - Modifier les attributs d'un paramètre
  - Ajouter/Enlever le paramètre de retour

Pour les traces canoniques le besoins est de pouvoir éditer la valeur de la trace, avec accès à un toolbar/menu pour les symboles.

Pour les spécifications des programmes:

- Génération automatique des signatures des programmes
- Toolbar/Menu pour les symboles et opérateurs
- Spécification des fonctions auxiliaire (*non-disponible dans cette version*)

Autre besoins:

- Validation des prédicats et autres entrées.
- Ouverture/Sauvegarde dans un fichier
- Sauvegarde automatique (*non-disponible dans cette version*)
- Impression de la spécification

Pour une description détaillée des besoins consulter le document des besoins *(besoins.doc).*

Outils Utilisés

L'éditeur de Trace a été développé avec le langage de programmation Java de Sun Microsystems (version 1.5). Un des principaux avantages d'utiliser Java est la portabilité du logiciel : une fois compilé le programme peut être exécuté sur plusieurs plateformes. De plus Java comporte de nombreuses classes pour le développement d'interface usagers (SWING).

Nous avons utilisé le “Integrated Development Environment” (IDE) Eclipse (version 3) pour faire le développement même de l'éditeur. Cet IDE permet de faire tout le codage et la compilation. De plus plusieurs “plugins” ont été utilisés dans Eclipse pour simplifier le développement:

- ANT: outils pour faire les “build” de l'application, similaire à “make”
- Junit: “Framework” pour effectuer des tests unitaires
- Subclipse: “plugins” du serveur de code Subversion pour Eclipse

Nous avons donc définis un projet Java dans Eclipse contenant la structure suivante:

- trace
  - ant: contient les scripts ant
  - bin: contient le code compilé et le fichier .jar
  - cup: contient les fichiers de configuration pour la validation des prédicats
  - doc: répertoire pour la documentation (Javadoc)
  - lib: répertoire contenant diverse librairie
  - src: contient le code source de l'éditeur (et les classes de testes)

Pour la compilation de l'éditeur une librairie externe est nécessaire: java-cup-11a-runtime.jar, elle doit être incluse dans le “classpath” lors de la compilation.

Il faut noter que trois classes dans le package inf4173.trace.expression sont générées automatiquement par CUP et JFlex (voir section validation des prédicats): Parser.java, Scanner.java et SymbolConstants.java. Le script Ant build.xml, qui doit être utilisé pour bâtir l’application, contient des “targets” pour les nécessaires : génération des trois classes, compilation, exécution des tests, génération de Javadoc, exportation de l'application dans un fichier .jar

Interface Utilisateur

L'interface utilisateur est divisée en trois grandes sections :

- Le menu
- Les barres d'outils
- La fenêtre principale

Le menu contient les sous-menus standards retrouvés dans la majorité des applications comportant une interface graphique. Voici les fonctions des différents sous-menus :

- Fichier
  - Nouveau : Permet de créer un nouveau document pour spécifier un module. Cette fonction demandera à l'utilisateur de sauvegarder un document existant si ce n'était pas déjà fait.
  - Ouvrir : Permet à l'utilisateur d'ouvrir une spécification de module qui a été sauvegardée sur le disque dur.
  - Sauvegarder : Permet à l'utilisateur de sauvegarder la spécification de module courante. Si elle n'a pas déjà été sauvegardée, la fonction "Sauvegarder sous" sera appelée.
  - Sauvegarder sous : Permet a l'utilisateur de sauvegarder la spécification de module courante en spécifiant l'emplacement sur le disque dur et le nom du fichier.
  - Imprimer : Permet à l'utilisateur d'imprimer la spécification de module courante.
  - Quitter : Permet à l'utilisateur de quitter l'application.

- Insérer
  - Symboles arithmétiques : Permet à l'utilisateur d'insérer un symbole arithmétique lors de l'édition d'une expression quelconque.
  - Symboles booléens : Permet à l'utilisateur d'insérer un symbole booléen lors de l'édition d'une expression quelconque.
  - Fonctions : Permet à l'utilisateur d'insérer une fonction, qui a été spécifiée dans la table syntaxique, lors de l'édition d'ne expression quelconque.

- Affichage
  - Barres d'outils : Permet à l'utilisateur de spécifier l'affichage (visible ou non) des différentes barres d'outils de l'application.

- ?

- A propos : Permet à l'utilisateur de visualiser la fenêtre “A propos”.

Les barres d'outils, qui peuvent être utilisées pour insérer des symboles ou des fonctions lors de l'édition d'expressions, sont les suivantes :

- Symboles arithmétiques : Permet à l'utilisateur d'insérer un symbole arithmétique lors de l'édition d'une expression quelconque.
- Symboles booléens : Permet à l'utilisateur d'insérer un symbole booléen lors de l'édition d'une expression quelconque.
- Fonctions : Permet à l'utilisateur d'insérer une fonction, qui a été spécifiée dans la table syntaxique, lors de l'édition d'ne expression quelconque.

La fenêtre principale est divisée en trois onglets qui reflètent essentiellement les différentes parties de la spécification de modules en utilisant la méthode de traces canoniques :

- Table syntaxique : Cet onglet permet à l'utilisateur, via une table, de spécifier les différents programmes d'accès utilisé par la spécification de module. Pour chaque programme, un utilisateur doit spécifier :
  - Le nom de fonction : Le nom est spécifié à l'aide d'un champ de texte. Il sera validé à l'aide d'une expression régulière.
  - Les paramètres : L'utilisateur doit spécifier le type de donnée du paramètre (String, Integer, etc.) et le type de paramètre (O, V, VO) à l'aide de listes déroulantes. L'utilisateur peut aussi spécifier un nouveau type de donnée qui sera validé avec une expression régulière.
  - Le type de retour : L'utilisateur doit spécifier le type de donnée du paramètre (String, Integer, etc.). Il peut aussi spécifier un nouveau type de donnée qui sera validé avec une expression régulière.

- Traces canoniques : Cet onglet permet à l'utilisateur, via un champ d'entrée de texte, de spécifier les traces canoniques du programme d'accès. Il pourra utiliser le menu Insérer et les différentes barres d'outils pour faciliter la spécification. Une fois terminé, l'utilisateur pourra cliquer sur le bouton valider pour savoir si la syntaxe de son expression est correcte ou non.

- Fonctions d'équivalence : Cet onglet permet à l'utilisateur de spécifier la valeur des différents programmes d'accès de la table syntaxique. Une liste déroulante permet à l'utilisateur de choisir le programme qu'il désire spécifier. L'application se chargera alors la tache de générer automatiquement les signatures pour lesquelles l'utilisateur doit spécifier la nouvelle valeur (essentiellement, tous les paramètres de type O et le type de retour). De plus, l'utilisateur peut spécifier à l'aide d'une table, les valeurs de légalité des programmes d'accès.

Validation des Prédicats

Comme mentionné dans l'introduction un des buts principal de l'éditeur est de simplifier l'entrée des prédicats et aussi d'effectuer la validation de ces prédicats. Nous avons dut coder un validateur pour les prédicats, plus particulièrement un validateur pour les expressions arithmétiques et booléienne. La validation des prédicats revient donc tout simplement à faire l'analyse syntaxique du prédicat. L'analyse syntaxique du prédicat a été divisée en deux sections: l'analyse lexicale et l'analyse sémantique.

Analyse Lexicale

L'analyse lexicale consiste à transformer le prédicat en tokens. Pour générer la classe qui effectue cette opération nous avons utilisé l'outil "open source" JFlex qui génère un scanneur en code Java à partir d'un simple fichier de configuration. Le principe derrière le scanneur est la reconnaissance des tokens dans les prédicats à l'aide d'expression régulière. Les tokens sont spécifiés dans une interface lui-même généré (par Cup, voir plus loin): SymbolConstants.java. Voici les tokens acceptés dans l'éditeur:

PLUS	+
MINUS	-
TIMES	*
DIVIDE	/
MODULO	%
AND	∧
OR	∨
NOT	!
EQUAL	=
LESS	<
GREATER	>
LESS_EQUAL	<= ou ≤
GREATER_EQUAL	>= ou ≥
NOT_EQUAL	!= ou ≠
LEFT_PARENTHESIS	(
RIGHT_PARENTHESIS	)
LEFT_BRACKET	[
RIGHT_BRACKET	]
LEFT_CURLY	{
RIGHT_CURLY	}
ELEMENT	∈, ∉, ⊂, ⊃, ⊄, ⊅, ⊆, ⊇, ⊈ ou ⊉
FOR_ALL	∀
EXISTS	∃
PERIOD	.
COMMA	,
SEMICOLUM	;
COLUM	:

WHERE	where (insensible à la casse)
LITERAL	n'importe quel texte
NUMBER	tous les nombres
BOOLEAN	true, false, vrai ou faux (insensible a la casse)
FUNCTION	_ ou une fonction ex: f(x,y)

Tout les espaces, tabulation, retour de chariot sont complètement ignorés par le scanneur. Le scanneur essai toujours de faire match avec l'expression régulières la plus longue sinon il suit l'ordre dans le fichier de configuration. Pour ce qui est des quatre derniers, ceux lorsqu'ils sont convertie en token reçoivent aussi un type: arithmétique, booléienne ou inconnue. Le token LITERAL reçoit automatiquement le type inconnu car il correspond à une variable. Le token NUMBER reçoit le type arithmétique, le token BOOLEAN reçoit le type booléienne. Pour le token FUNCTION un traitement spécial est effectué:

- le nom de la fonction est recherché dans la table syntaxique, si elle n'est pas trouvée le token est refusé;
- le nombre de paramètre est comparé avec celui dans la table syntaxique, s'il est différent le token est refusé;
- ensuite le type est déterminer en analysant le type de retour définit dans la table syntaxique.

Une fois la conversion terminée, le scanneur retourne une pile contenant tout les tokens lus. Si la conversion est impossible pour une série de caractères le scanneur lance une exception donnant la position dans la chaine de caractères ou se trouve cette erreur; dans ce cas le traitement arrête à point.

Analyse Sémantique

L'analyse sémantique constitue en l'analyse du sens ou de la compréhension des tokens. L'analyse sémantique se fait à partir d'un parseur de token et d'une grammaire. Comme pour le scanneur nous avons utilisé un générateur de parseur pour créer le code Java nécessaire, cet outil s'appelle CUP. Le type de parseur généré par CUP est un parseur “Look Ahead LR” utilisant une grammaire formelle avec des règles de production du type: $V \rightarrow w$, ou V est un symbole non-terminal et w est une série de soit un ou plusieurs symboles terminaux ou non-terminaux. Une grammaire formelle veut dire que pour chaque V peuvent être remplacé par w. De plus dans le cas de CUP chaque règle de production peut comporter d'actions qui sont exécuté lorsque la règle est activée.

Pour notre éditeur nous avons dut développer une grammaire pour les traces canoniques. Cette grammaire à donc la possibilité d'accepter les opérations arithmétiques, booléennes et la notation pour les traces canoniques. Voici les différents symboles terminaux de la grammaire:

```
PLUS, MINUS, TIMES, DIVIDE, MODULO
UNARY_MINUS
AND, OR
NOT
EQUAL, LESS, GREATER, LESS_EQUAL, GREATER_EQUAL, NOT_EQUAL

LEFT_PARENTHESIS, RIGHT_PARENTHESIS
LEFT_BRACKET, RIGHT_BRACKET
LEFT_CURLY, RIGHT_CURLY
ELEMENT
FOR_ALL, EXISTS, WHERE

LITERAL, NUMBER, BOOLEAN, FUNCTION

PERIOD, COMMA, SEMICOLUM, COLUM
```

Noter que ce sont les mêmes symboles que pour le scanneur. Voici maintenant la liste des symboles non-terminaux:

```
expr;
arit_expr;
bool_expr;
trace, sousTrace;
dec_list, declaration, var_list, definition, iterator;
```

Voici la grammaire définit pour accepter les traces canoniques (les actions de chaque règle ne sont par présentés):

```
expr        ::= NUMBER
            | BOOLEAN
            | LITERAL
            | LEFT_PARENTHESIS expr RIGHT_PARENTHESIS
            | arit_expr
            | bool_expr
            | trace
            | expr WHERE dec_list
            | EXISTS dec_list SEMICOLUM expr
            | FOR_ALL dec_list SEMICOLUM expr

trace       ::= LEFT_BRACKET trace PERIOD sousTrace RIGHT_BRACKET LEFT_CURLY
            iterator RIGHT_CURLY
            | trace PERIOD sousTrace
            | sousTrace;

sousTrace   ::= LEFT_BRACKET FUNCTION RIGHT_BRACKET LEFT_CURLY iterator RIGHT_CURLY
            | FUNCTION;

iterator ::= LITERAL EQUAL NUMBER PERIOD PERIOD LITERAL
```

```
                | LITERAL EQUAL LITERAL PERIOD PERIOD LITERAL
                | LITERAL EQUAL NUMBER PERIOD PERIOD NUMBER
                | LITERAL EQUAL LITERAL PERIOD PERIOD NUMBER;

dec_list ::= dec_list SEMICOLUM declaration
                | declaration;

declaration     ::= var_list definition LESS LITERAL GREATER
                | var_list definition LESS LESS LITERAL GREATER GREATER;

var_list ::= var_list COMMA LITERAL
                | LITERAL;

definition      ::= COLUM
                | ELEMENT;

arit_expr       ::= expr PLUS expr
                | expr MINUS expr
                | expr TIMES expr
                | expr DIVIDE expr
                | expr MODULO expr
                | MINUS expr;

bool_expr       ::= expr AND expr
                | expr OR expr
                | NOT expr
                | expr EQUAL expr
                | expr NOT_EQUAL expr
                | expr LESS expr
                | expr LESS_EQUAL expr
                | expr GREATER expr
                | expr GREATER_EQUAL expr;
```

Dans certaines séries de production il y a aussi des vérifications qui sont effectuées sur les arguments de la production par exemple pour arit_expr ::= expr PLUS expr, les deux termes expr doivent être de type arithmétique pour que la règle soit accepter.

La grammaire prend chaque symbole de la pile fournit par le parseur et tente de d'activer la règle correspondante à la suite des symboles. Si le parseur est incapable d'activer une règle une exception est produite. Aussi si une des vérifications effectuées dans une règle ne passe pas une exception est aussi lancée.

La plus grande difficulté dans la construction du parseur fut de tenir en compte des variables (LITERAL). Le problème se résous à ceci, par exemple dans cet exemple X+1 on ne sait pas quelle est le type de la variable X, arithmétique ou booléienne. Ce problème a été réglé en utilisant un “HashMap” de paire variable, type et le principe suivant:

- Si la variable n'existe pas dans le vecteur:
  - La rajouter avec le type nécessaire pour que la règle de production soit valide.
- Si la variable existe vérifier sont type:
  - Si le type est valide selon la règle continuer
  - Si le type n'est pas valide avec la règle, lancer une exception.

Si le parseur réussit donc à construire l'arbre syntaxique, trouver toute les règles de production pour simplifier l'expression en l'axiome (expr), alors l'expression est acceptée.

Conclusion

Travail future

Puisque l’application se trouve à sa première version, elle pourrait être enrichie de nouvelles fonctionnalités. En voici quelques-unes :

- Spécification des fonctions auxiliaires : Ceci pourrait être fait à l’aide d’un quatrième onglet dans la fenêtre principale de l’application. On pourrait aussi ajouter une barre d’outils qui permettrait d’insérer ces fonctions auxiliaires.
- Sauvegarde automatique : Fonctionnalité qui sauvegarderait la spécification courante dans un fichier temporaire à un intervalle de temps fixe (qui lui aussi pourrait être spécifié par l’utilisateur).
- Spécification du nom des paramètres : Dans cette version, le nom des paramètres est généré automatiquement. L’onglet des fonctions d’équivalence pourrait permettre à l’utilisateur de spécifier le nom des paramètres.
- Assurance de l’exclusion mutuelle des conditions de fonctions : Ceci serait une fonction qui lierait les conditions spécifiées dans la table de conditions d’une fonction pour s’assurer que les conditions sont mutuellement exclusives.
- Assurance d’avoir couvert toutes les conditions de fonctions possibles : Ceci serait une fonction qui lierait les conditions spécifiées dans la table de conditions d’une fonction pour s’assurer que toutes les conditions possibles aient été spécifiées.
- Codage d’autres règles syntaxiques: Ajout de la clause “where” au scanneur et au parseur pour permettre à l’utilisateur de l’utiliser dans les expressions de la spécification du module. Exemple : x where x:<int>
- Définition multiple de variables : Permettrait de redéfinir le type de donnée d’une variable à l’intérieur d’un bloc de spécification dans une expression.
  Exemple : ∀x:<bool>; (x∧∀x:<int>;x<1)
- Surcharge des fonctions : Permettrait à l’utilisateur de surcharger des fonctions de la table syntaxique en spécifiant un différent nombre de paramètres ou des paramètres de types différents.

Bibliographies

Site du Langage Java:
http://java.sun.com/index.jsp

API du Language Java:
http://java.sun.com/j2se/1.5.0/docs/api/

Sérialisation dans Java:
http://java.sun.com/developer/technicalArticles/Programming/serialization/

Site d'Ant pour les "build":
http://ant.apache.org/

JUnit:
http://www.junit.org/index.htm

CUP générateur de parseur:
http://www2.cs.tum.edu/projects/cup/

JFlex générateur de scanneur:
http://jflex.de/

Référence sur les expressions régulières:
http://www.regular-expressions.info/

Jude UML:
http://jude.change-vision.com/jude-web/index.html

Référence sur UML:
http://www.javagen.com/docs/uml.html

Note: Ces liens ont tous été visités à l'automne 2006, il se peut qu'ils aient été déplacés ou modifiés.

Annexe

Diagrammes de classes

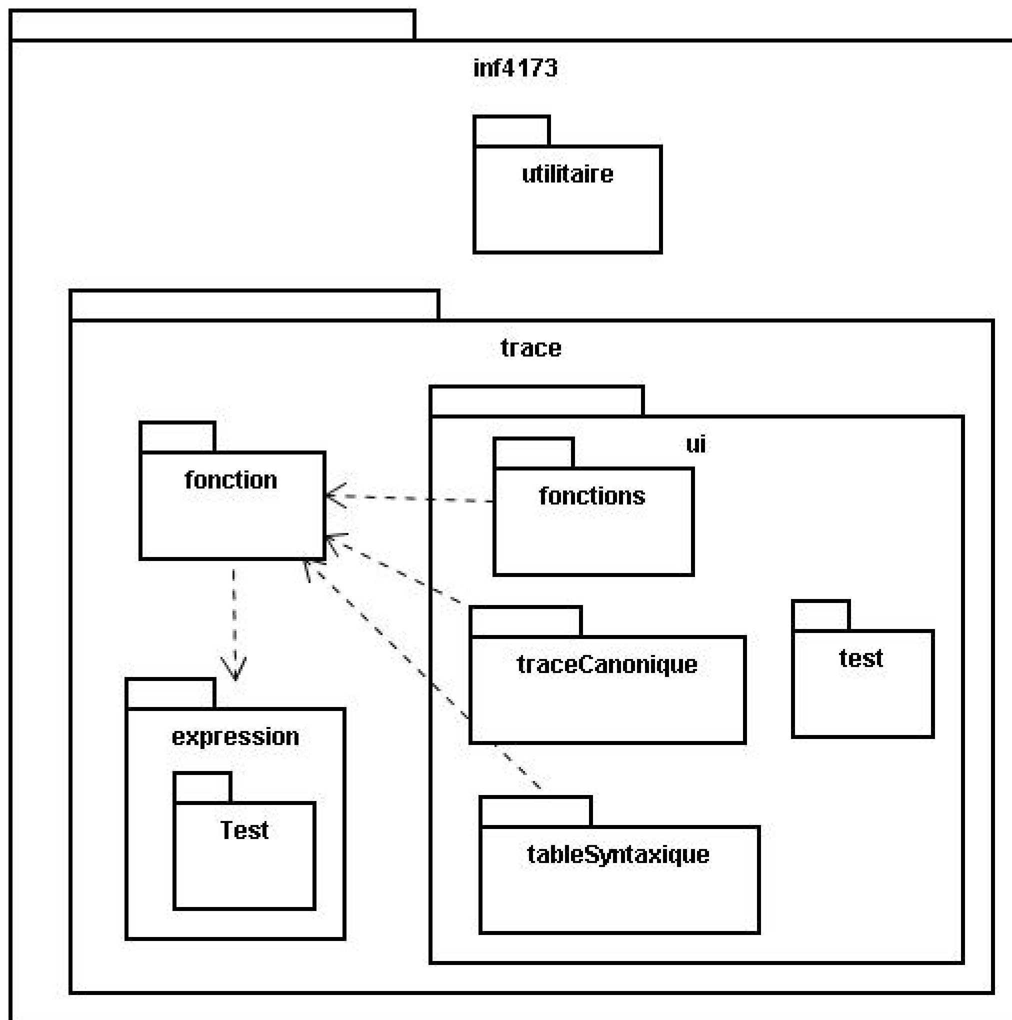


Figure 1: Diagramme de haut niveau des packages

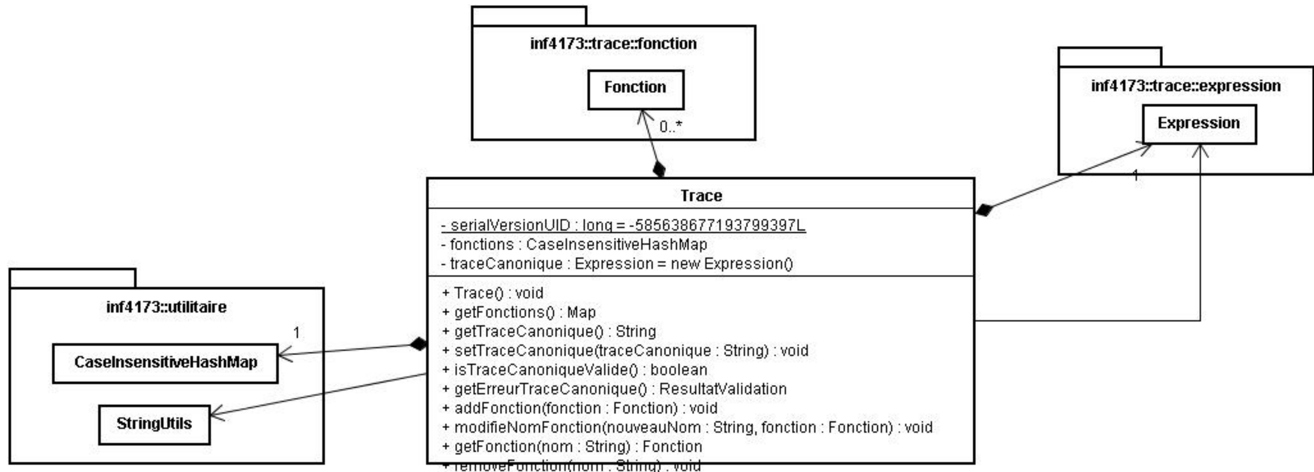
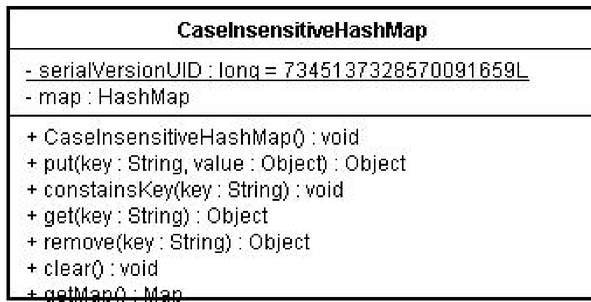


Figure 2 : Package inf4173.trace



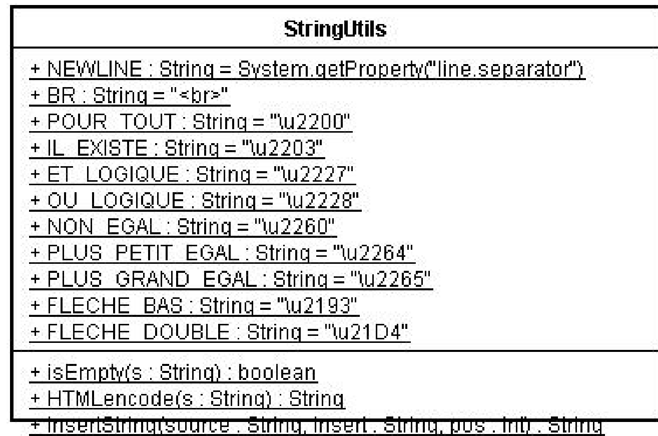


Figure 3 : Package inf4173.utilitaire

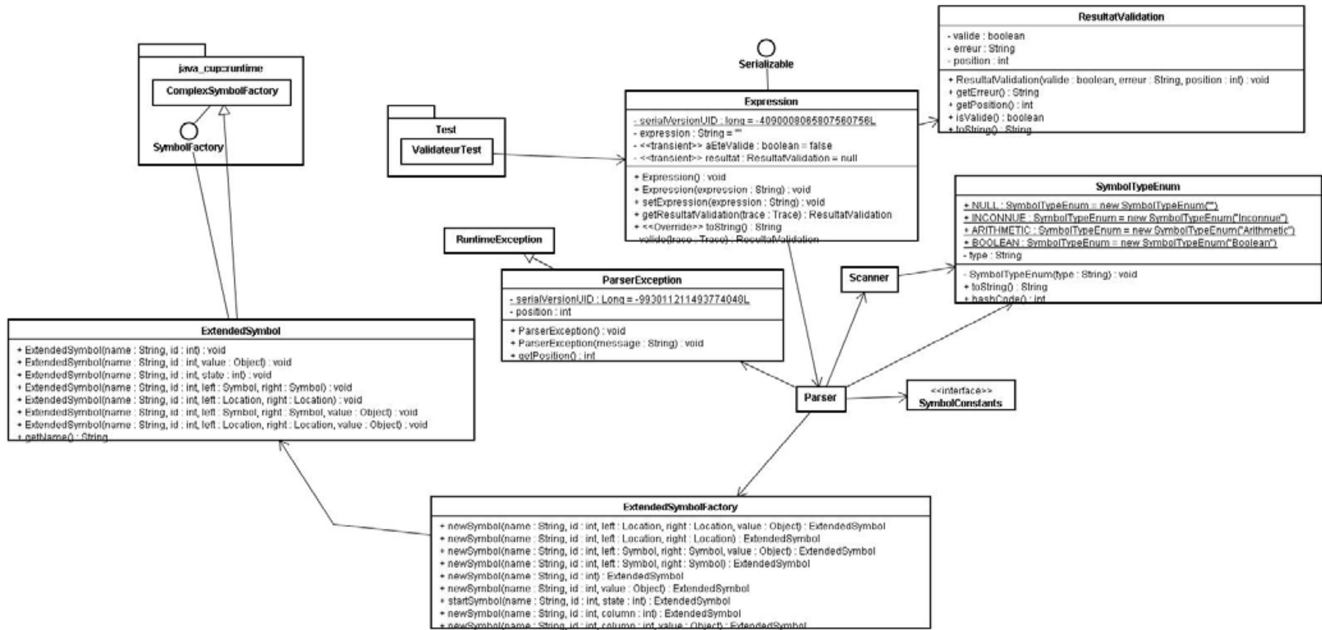


Figure 4 : Package inf4173.trace.expression

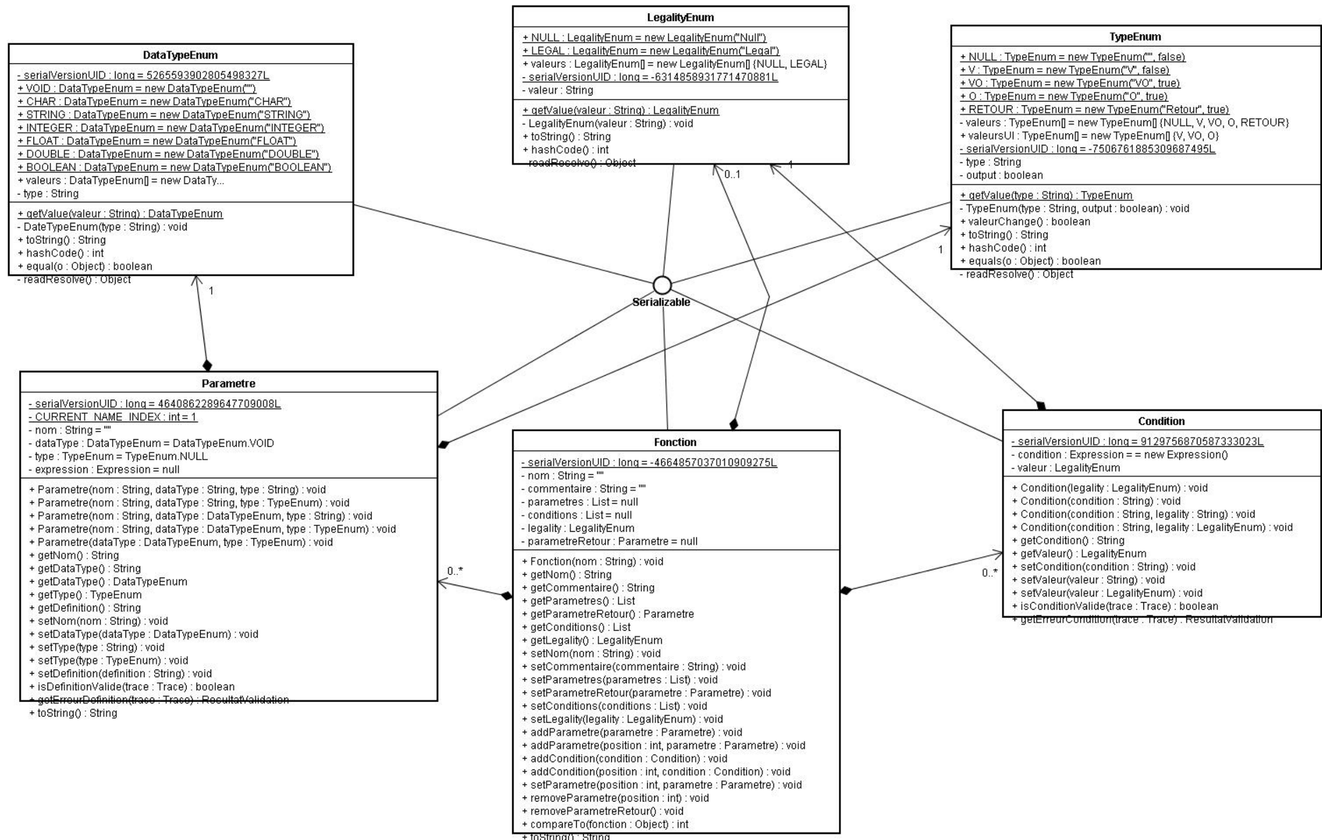


Figure 5 : Package inf4173.trace.fonction

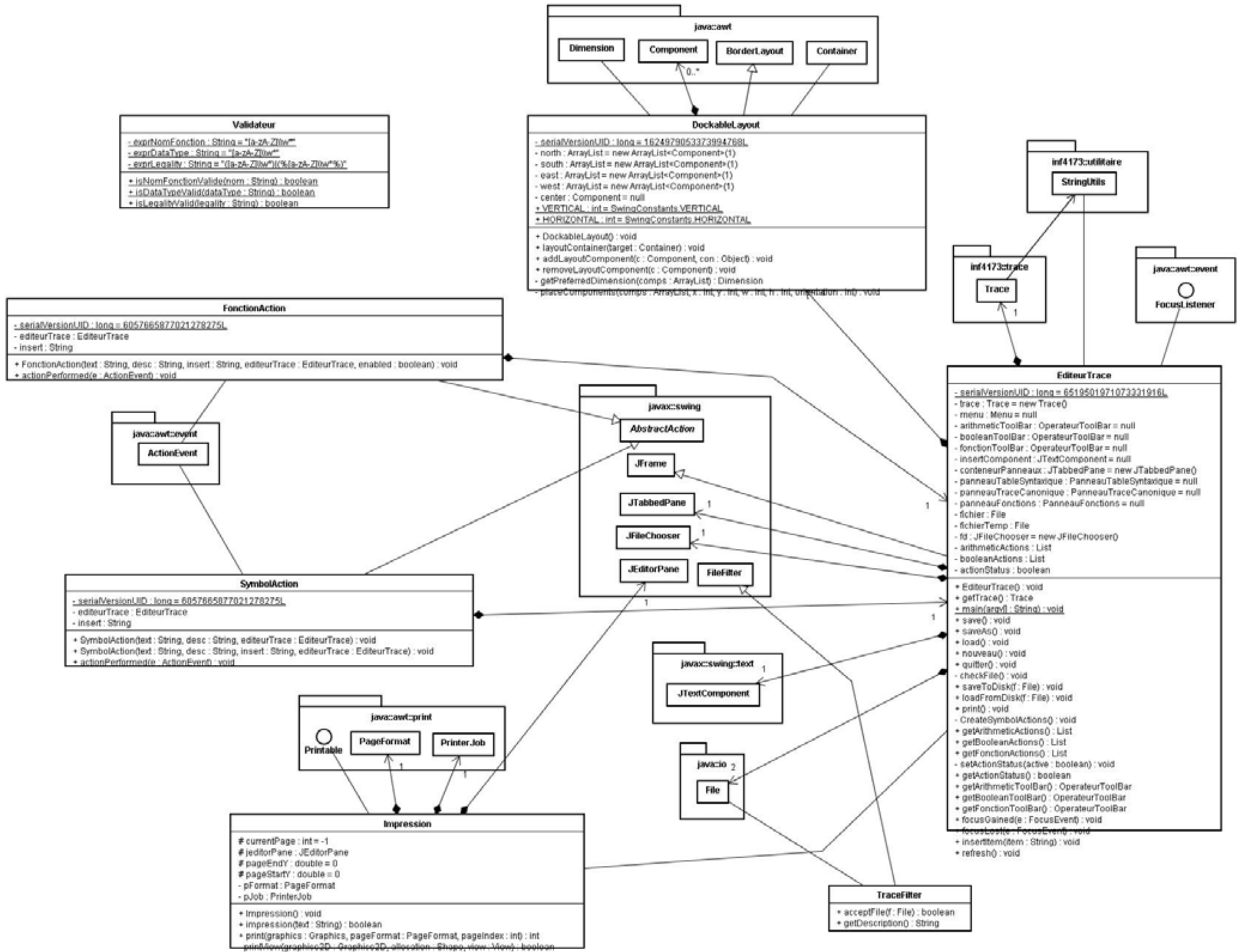


Figure 6 : Package inf4173.trace.ui

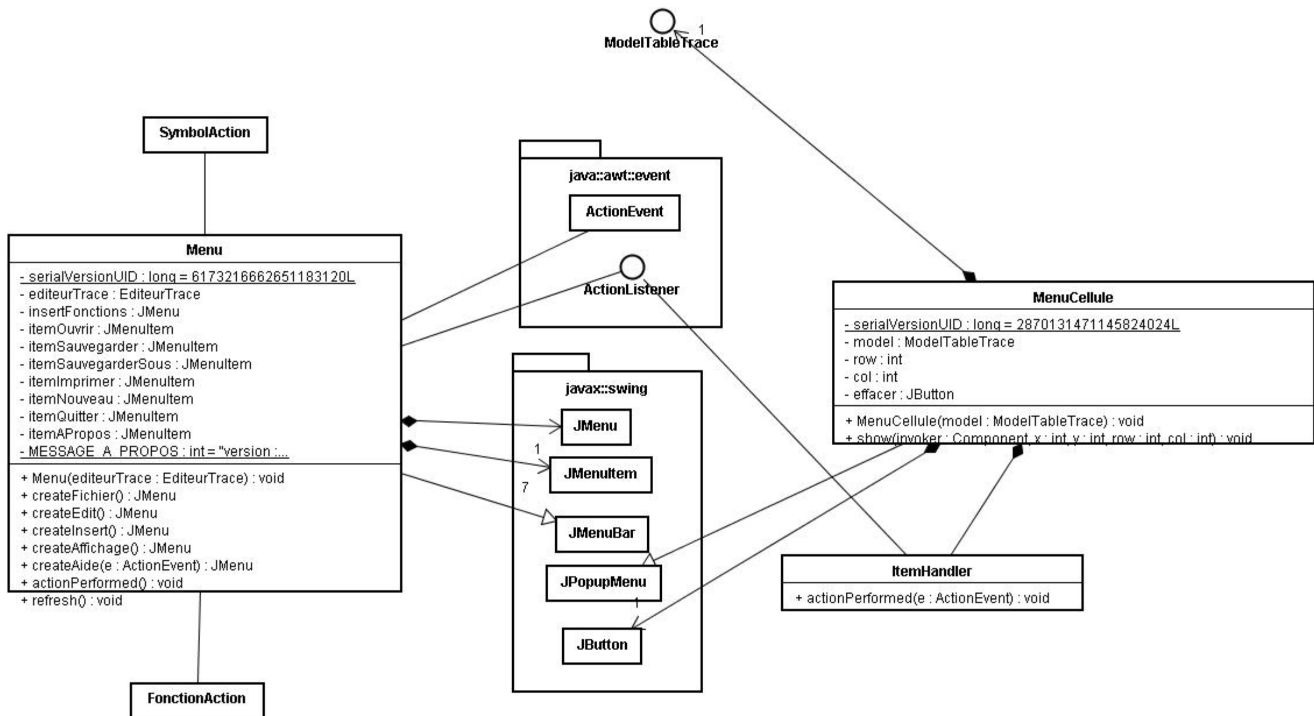


Figure 7 : Package inf4173.trace.ui (Menu)

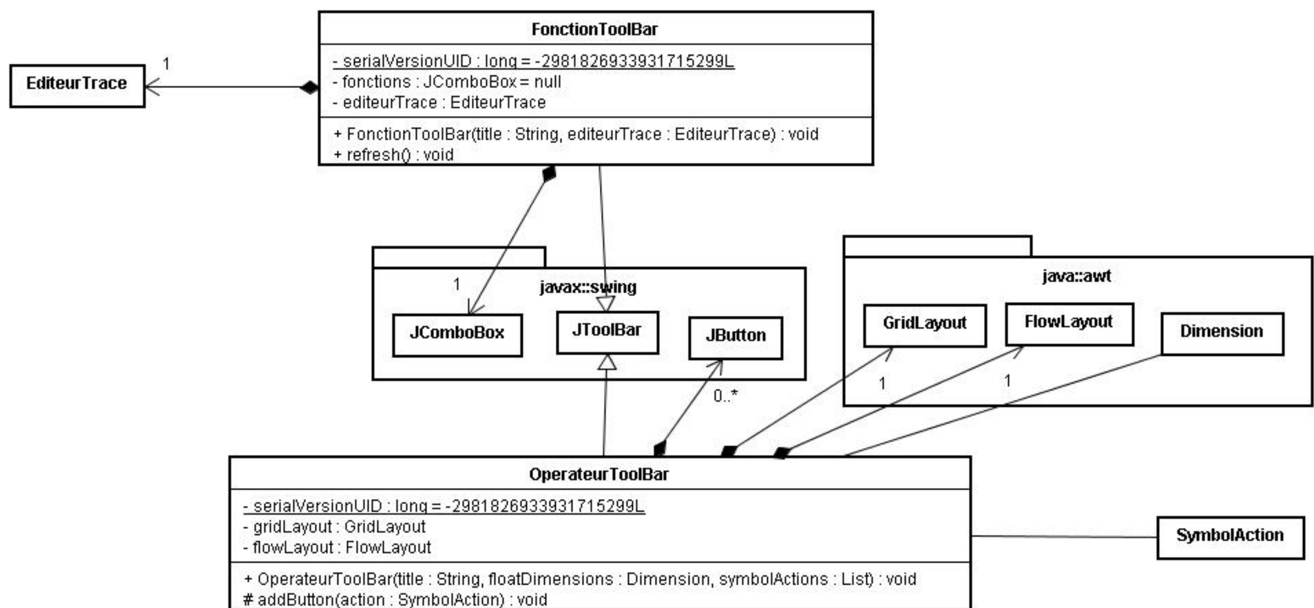


Figure 8 : Package inf4173.trace.ui (Toolbars)

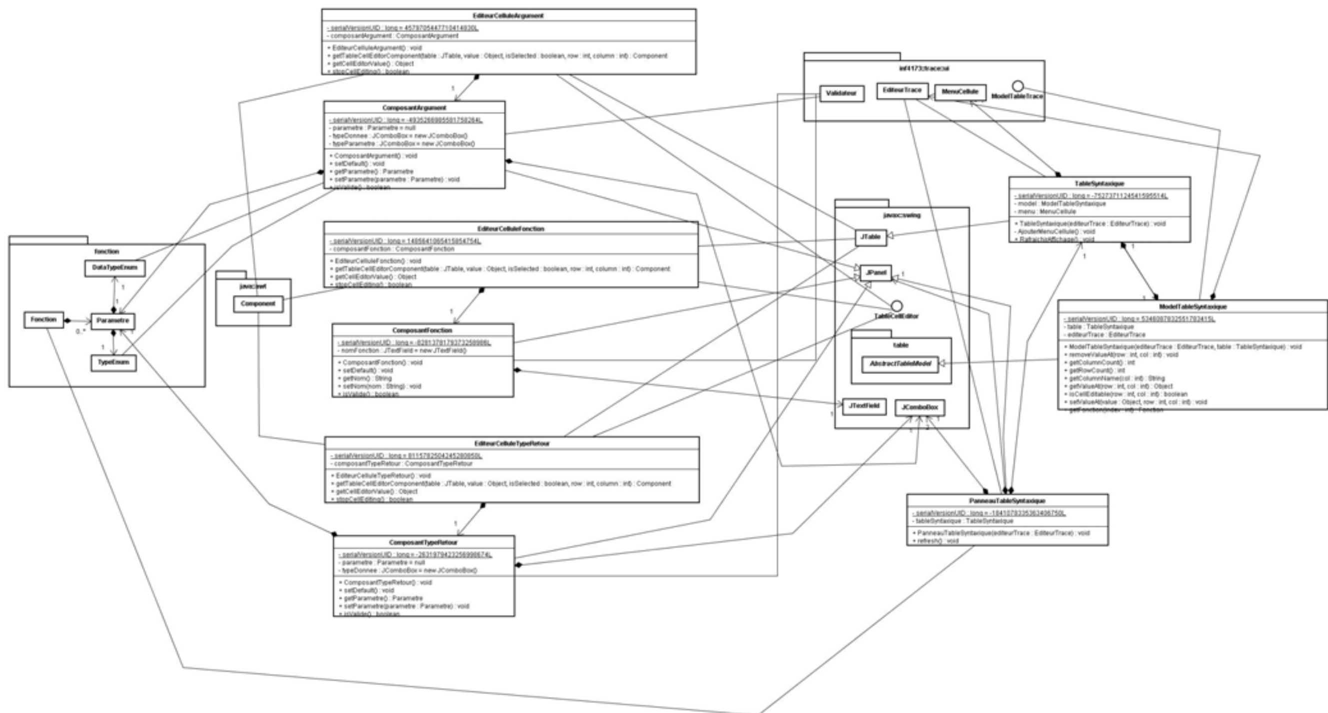


Figure 9 : Package inf4173.trace.ui.tableSyntaxique

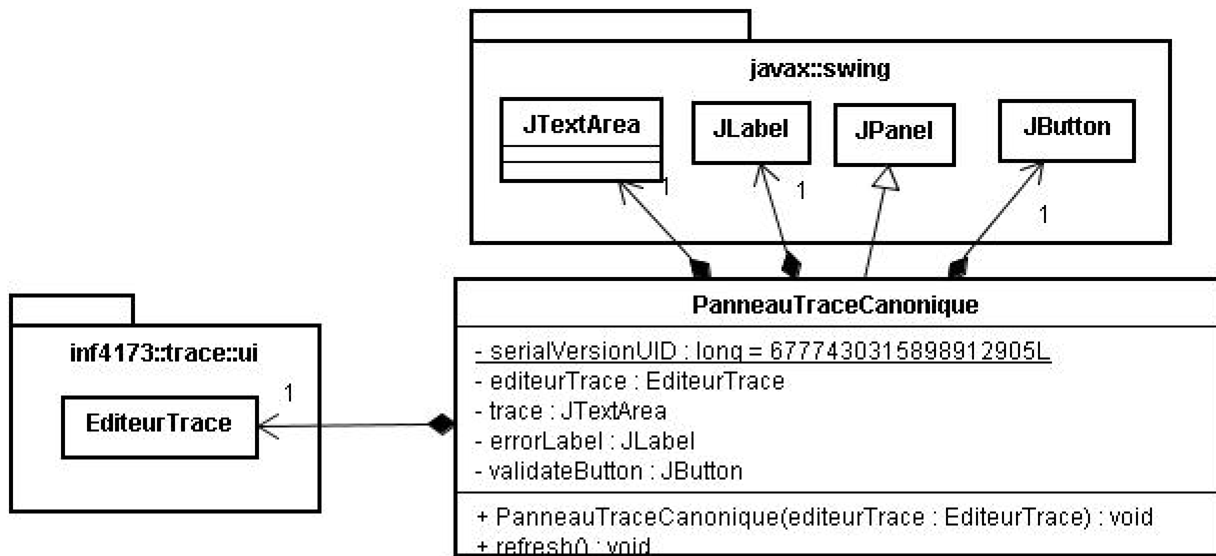


Figure 10 : Package inf4173.trace.ui.traceCanonique

Figure 11 : Package inf4173.trace.ui.fonctions